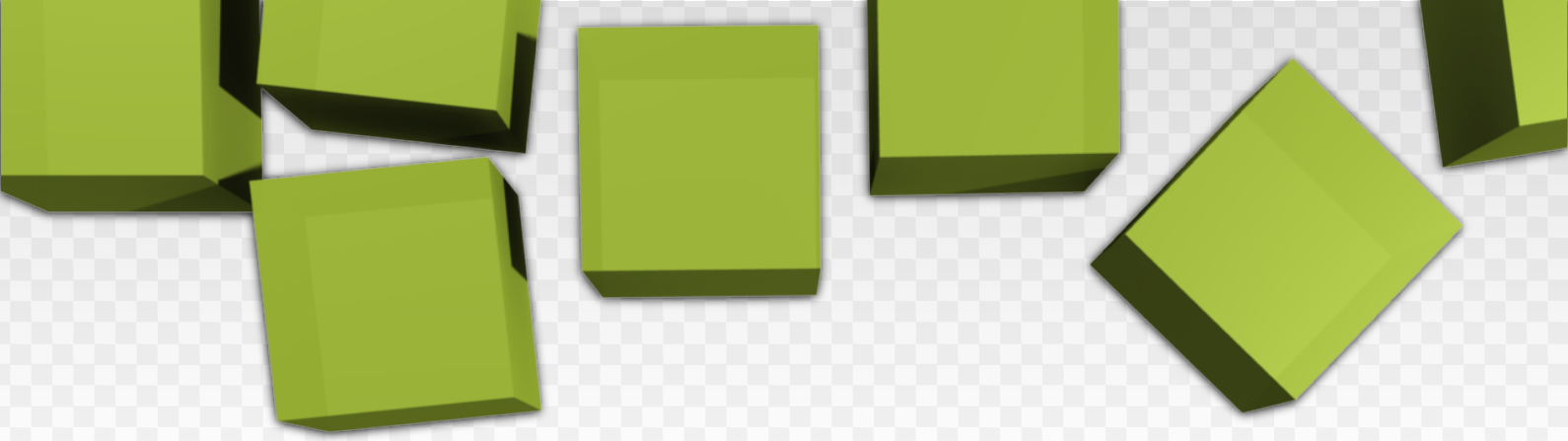


Introduction to Web Applications Development

Author:
C. Mateu

Coordinator:
J. Mas

Free Technology Academy

4

Introduction to web application development

Jordi Mas (coordinator)
Carles Mateu

PID_00148372

Universitat Oberta de Catalunya
www.uoc.edu

Jordi Mas

Software engineer at the free software company, Ximian, where he works on implementation of the free project, Mono. He volunteers with the development of the Abiword word processor and engineering of the Catalan versions of the Mozilla and Gnome project. He is also the general coordinator of Softcatalà. As a consultant, he has worked for companies such as Menta, Telépolis, Vodafone, Lotus, eresMas, Amena and Terra España.

Carles Mateu

Engineer in IT from the UOC. He is currently Director of the Information and Communication Systems Department of the UdL and Associate Professor of Networks and the Internet at the UdL.

First edition: February 2010

Av. Tibidabo, 39-43, 08035 Barcelona
Design: Manel Andreu
Publishing: Eureca Media, SL

Preface

Software has become a strategic societal resource in the last few decades. The emergence of Free Software, which has entered in major sectors of the ICT market, is drastically changing the economics of software development and usage. Free Software – sometimes also referred to as "Open Source" or "Libre Software" – can be used, studied, copied, modified and distributed freely. It offers the freedom to learn and to teach without engaging in dependencies on any single technology provider. These freedoms are considered a fundamental precondition for sustainable development and an inclusive information society.

Although there is a growing interest in free technologies (Free Software and Open Standards), still a limited number of people have sufficient knowledge and expertise in these fields. The FTA attempts to respond to this demand.

Introduction to the FTA
The Free Technology Academy (FTA) is a joint initiative from several educational institutes in various countries. It aims to contribute to a society that permits all users to study, participate and build upon existing knowledge without restrictions.

What does the FTA offer?
The Academy offers an online master level programme with course modules about Free Technologies. Learners can choose to enrol in an individual course or register for the whole programme. Tuition takes place online in the FTA virtual campus and is performed by teaching staff from the partner universities. Credits obtained in the FTA programme are recognised by these universities.

Who is behind the FTA?
The FTA was initiated in 2008 supported by the Life Long Learning Programme (LLP) of the European Commission, under the coordination of the Free Knowledge Institute and in partnership with three european universities: Open Universiteit Nederland (The Netherlands), Universitat Oberta de Catalunya (Spain) and University of Agder (Norway).

For who is the FTA?
The Free Technology Academy is specially oriented to IT professionals, educators, students and decision makers.

What about the licensing?
All learning materials used in and developed by the FTA are Open Educational Resources, published under copyleft free licenses that allow them to be freely used, modified and redistributed. Similarly, the software used in the FTA virtual campus is Free Software and is built upon an Open Standards framework.

Evolution of this book

The FTA has reused existing course materials from the Universitat Oberta de Catalunya and that had been developed together with LibreSoft staff from the Universidad Rey Juan Carlos. In 2008 this book was translated into English with the help of the SELF (Science, Education and Learning in Freedom) Project, supported by the European Commission's Sixth Framework Programme. In 2009, this material has been improved by the Free Technology Academy. Additionally the FTA has developed a study guide and learning activities which are available for learners enrolled in the FTA Campus.

Participation

Users of FTA learning materials are encouraged to provide feedback and make suggestions for improvement. A specific space for this feedback is set up on the FTA website. These inputs will be taken into account for next versions. Moreover, the FTA welcomes anyone to use and distribute this material as well as to make new versions and translations.

See for specific and updated information about the book, including translations and other formats: *http://ftacademy.org/materials/fsm/1*. For more information and enrolment in the FTA online course programme, please visit the Academy's website: *http://ftacademy.org/*.

I sincerely hope this course book helps you in your personal learning process and helps you to help others in theirs. I look forward to see you in the free knowledge and free technology movements!

Happy learning!

Wouter Tebbens
President of the Free Knowledge Institute
Director of the Free technology Academy

Acknowledgenments

The authors wish to thank the Fundació per a la Universitat Oberta de Catalunya (http://www.uoc.edu) for financing the first edition of this work under the framework of the International Master's degree in Free Software offered by this institution.

The current version of these materials in English has been extended with the funding of the Free Technology Academy (FTA) project. The FTA project has been funded with support from the European Commission (reference no. 142706- LLP-1-2008-1-NL-ERASMUS-EVC of the Lifelong Learning Programme). This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Contents

Module 6

Database access: JDBC

David Megías Jiménez, Jordi Mas and Carles Mateu

1. Introduction to databases
2. Controllers and addresses
3. Basic database access
4. Prepared statements and stored procedures
5. Transactions
6. Metadata
7. Practical: database access

Module 7

Web services

David Megías Jiménez, Jordi Mas and Carles Mateu

1. Introduction to web services
2. XML-RPC
3. SOAP
4. WSDL and UDDI
5. Security

Module 8

Use and maintenance

David Megías Jiménez, Jordi Mas and Carles Mateu

1. Configuring security options
2. Configuring load balancing
3. Configuring a *caching proxy* with Apache
4. Other Apache modules

Module 9

Monitoring and analysis

David Megías Jiménez, Jordi Mas and Carles Mateu

1. Analysis of HTTP server logs
2. Statistics and counter tools
3. Performance analysis

Annex

Introduction to web applications

Carles Mateu

PID_00148404

Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Introduction to the Internet

Internet, the network of networks, came about in the mid-1970s under the auspices of DARPA, the United States Defense Advanced Research Projects Agency. DARPA launched a research programme into techniques and technologies to connect several packet switching networks that would allow the computers connected in these networks to communicate with one another easily and transparently. These projects led to the birth of a data communication protocol called IP, the Internet Protocol, which allowed several computers to communicate through a network, the Internet, formed by the interconnection of several networks.

In the mid-1980s, the United States National Science Foundation created a network called NSFNET, which became the backbone of the Internet in conjunction with similar networks created by NASA (NSINet) and the US DoE or Department of Energy (ESNET). In Europe, most countries had national backbones (NORDUNET, RedIRIS, SWITCH, etc.) and a series of pan-European initiatives sprang up too (EARN and RARE). It was around this time that the first private Internet providers emerged, offering paid access to the Internet.

From this point on, due in part to the wide availability of implementations of the suite of TCP/IP protocols (consisting of all Internet protocols, rather than just TCP and IP), some of which were open source, the Internet began what would subsequently become one of its basic features, an exponential growth that only began to decline slightly in mid-2002.

The mid-1990s saw the Internet boom and it was around this time that the number of private Internet access providers rocketed, allowing millions to connect to the Internet, which was coming to be known as the Net, overtaking all existing communication networks (Compuserve, FidoNet/BBS etc). The turning point came with the emergence of free TCP/IP implementations (including those forming part of the operating system) and the popularisation and falling price of increasingly faster means of access (faster modems, ISDN, ADSL, cable, satellites). All of these changes led to a snowball effect whereby the more users that connected, the more costs fell, the more providers that emerged and the more attractive and economical the Internet became, which meant that more and more users began connecting, etc.

Nowadays, having an e-mail address, web access etc. is considered normal in many countries around the world and is not regarded as the latest thing. Businesses, institutions, governments, etc. are quickly migrating all of their services, applications, stores, etc. to a web environment that will allow

their customers and users access to all this from the Internet. Despite the slight slowdown in its growth rate, the Internet is set to become a universal communications service that allows universal communication.

2. The WWW as an Internet service

The WWW (World Wide Web) or Web as it is informally known, has, together with e-mail, become the warhorse of the Internet. The Web has evolved from an immense "library" of static pages into a service offering access to multiple features and functions, an infinite number of services, programs, stores, etc.

2.1. A brief history of the WWW

In 1989, while working at CERN, the European Organization for Nuclear Research, Tim Berners-Lee began to design a system for easy access to CERN's information. This system used hypertext to structure a network of links between documents. After obtaining approval to continue with the project, the first web browser was born, christened WorldWideWeb (without spaces).

By 1992, the system had extended beyond the walls of CERN and there were now considerably more "stable" servers: 26 Growth was now dramatic and by 1993 the Web was being mentioned in the New York Times. This was the year that Mosaic was launched, an X-Window/Unix browser that later became known as Netscape, a key factor in the popularisation of the Web. In 1994, the WWW Consortium was set up as the catalyst for the development of the prevailing standards on the Web (http://www.w3c.org). Its growth was now unstoppable and by the end of the 1990s, it had become the insignia service of the Internet, giving rise to the continuous growth of the online services that we know today.

2.2. Web basics

The amazing success of the Web is down to two basic features: HTTP protocol and HTML language. The first allows straightforward and easy implementation of a communications system so that any type of file can be easily sent, simplifying the operation of the server, allowing low-power servers to deal with thousands of requests and cutting deployment costs. The second feature provides an easy and straightforward mechanism for composing linked pages that is also highly efficient and very user-friendly.

2.2.1. HTTP

HTTP (Hypertext Transfer Protocol) is the basic protocol of the WWW. It is a straightforward, connection-oriented protocol without state. It is a connection-oriented protocol because it requires a communications protocol (TCP, Transmission Control Protocol) in connected mode, a protocol that establishes an end-to-end communication channel (between client and server) along which the bytes constituting the data to be transferred pass, in contrast to datagram or non-connection-oriented protocols, which divide data into small packages (datagrams) before sending them in different ways to the client from the server. The protocol does not maintain state, i.e. each data transfer is an independent connection separate from the previous one and there is no relationship between them. This is true to the point that, when we want to send a Web page, we need to send the HTML code of the text and the images it contains, since the initial HTTP specification, 1.0, opened and used as many connections as there were page components, transferring one component for each connection (the text of the page or each image).

Supplementary content

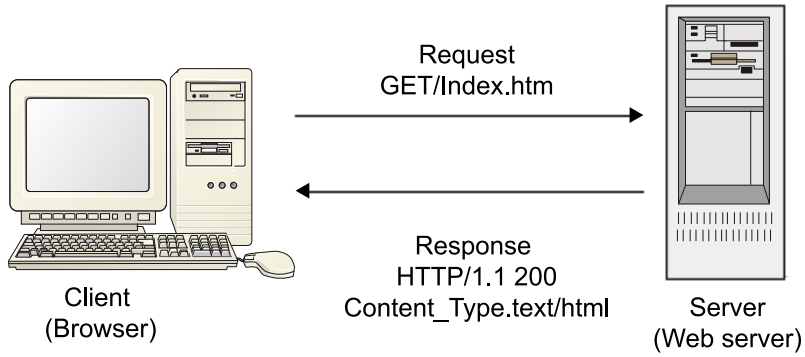
HTTP uses port 80 (equivalent in a way to the TCP service or connection identifier) for all default connections (we can use other ports besides 80).

There is a HTTP variant called HTTPS (S for secure) that uses the SSL (Secure Socket Layer) security protocol to encrypt and authenticate traffic between the client and the server. This is frequently used by e-commerce Web servers or for personal or confidential information.

Supplementary content

HTTPS uses port 443 by default.

The schematic operation of HTTP is as follows: the client sets up a TCP connection to the server, to the HTTP port (or that indicated in the address of the connection), it sends a HTTP resource request command (along with some informative headers) and the server responds through the same connection with the requested data and a series of informative headers.



<html>

...

The protocol also defines how to encrypt the passing of parameters between pages, tunnelling connections (for firewall systems), the existence of intermediate cache servers, etc.

The request for information directives defined by HTTP 1.1 (the version deemed stable and in use) are:

GET Request for resource.
POST Request for resource by passing parameters.
HEAD Request for data on resource.
PUT Creation or sending of resource.
DELETE Deletion of resource.
TRACE Echoes back the request just as it was received on the receiver for debugging.
OPTIONS Used to check server capacity.
CONNECT Reserved for use on intermediate servers that can operate as tunnels.

We will now look at some of these commands in detail as they are essential for the development of Web applications.

All resources to be served through HTTP must be referenced with a URL (Universal Resource Locator).

HTTP requests: GET and POST

In HTTP, requests can be made using one of two methods. If sending parameters with the request, GET will send them encrypted in the URL. The POST method will send parameters as part of the body of the request if sending them.

GET requests use the following format:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/4.5 [en]
Accept: image/gif, image/jpeg, text/html
Accept-language: en
Accept-Charset: iso-8859-1
```

We can see that this is made up of:

1) **Request line:** contains the requested resource.
2) **Request header:** contains additional information about the client.
3) **Request body:** in POST and PUT requests, among others, it contains additional information.

Request line

The request line contains the following elements:

1) **Method:** name of HTTP method called (GET, POST, etc.).
2) **Resource identifier:** URL (Uniform Resource Locator) of the requested resource.
3) **Protocol version:** protocol version requested for the response.

Request header

Contains additional information to help the server (or intermediate servers, proxies and caches) to process the request correctly. The information is provided as:

Identifier: value

Some of these identifiers, the most well-known and important being:

Host: name of requested server.
User-Agent: name of browser or program used to access the resource.
Accept: some text and image formats accepted by the client.
Accept-Language: languages supported (preferred) by the client, useful for automatically personalising the response.

Request parameters

A HTTP request can also contain parameters, for instance, as a response to a registration form, the selection of a product in an online store, etc. These parameters can be passed in two ways:

- As part of the request chain encrypted as part of the URL
- As extra request data

To encrypt parameters as part of the URL, they are added to the URL after the name of the resource, separated from the latter by the character **?**. The different parameters are separated from one another by the character **&**. Spaces are replaced by **+**. And special characters (those mentioned above **&**, **+** ¡ **?** and non-printing characters, etc.) are represented by **%xx** where **xx** represents the hexadecimal ASCII code of the character.

For example:

```
http://www.example.com/Index.jsp?name=Mr+Nobody&OK=l
```

In the HTTP request, this would end up as:

```
GET /index.jsp?name=Mr+Nobody&OK=l HTTP/1.0
Host: www.example.com
User-Agent: Mozilla/4.5 [en]
Accept: image/gif, image/jpeg, text/html
```

```
Accept-language: en
Accept-Charset: iso-8859-1
```

To pass the parameters as extra request data, they are sent to the server as the message body of the request. For example, the above request would look like this:

```
POST /index.jsp HTTP/1.0
Host: www.example.com
User-Agent: Mozilla/4.5 [en]
Accept: image/gif, image/jpeg, text/html
Accept-language: en
Accept-Charset: iso-8859-1

name=Mr+Nobody&OK=l
```

Note that to pass the parameters as the body of the request, the POST method rather than GET needs to be used, although POST requests can also carry parameters in the request line. Parameters passed as the body of the request are encrypted, as in the previous example, in the URL or they can use an encryption deriving from MIME [Multipurpose Internet Mail Extensions) format known as multipart encryption.

The previous request in multipart format would be:

```
POST /index.jsp HTTP/1.0
Host: www.example.com
User-Agent: Mozilla/4.5 [en]
Accept: image/gif, image/jpeg, text/html
Accept-language: en
Accept-Charset: iso-8859-1
Content-Type: multipart/form-data,
   delimiter="----RANDOM----"

----RANDOM----
Content-Disposition: form-data; name="name"
Mr Nobody
----RANDOM----
Content-Disposition: form-data; name="OK"
1

----RANDOM------
```

This encryption is exclusive to the POST method and used when sending files to the server.

HTTP responses

Responses in HTTP are very similar to requests. A W3C recommendation response to a request from a page would look something like this:

```
HTTP/1.1 200 OK
Date: Mon, 04 Aug 2003 15:19:10 GMT
Server: Apache/2.0.40 (Red Hat Linux)
Last-Modified: Tue, 25 Mar 2003 08:52:53 GMT
Accept-Ranges: bytes
Content-Length: 428
Connection: close  <HTML>
...
```

Here, we see that the first line responds with the version of the protocol used to send us the page followed by a return code and a return phrase. The return code can take one of the following values:

- 1xx Request received, still in process.
- 2xx Correct. Request processed correctly.
- 3xx Redirection. The request must be repeated or redirected.
- 4xx Client error. The request cannot be processed because it is incorrect, does not exist, etc.
- 5xx Server error. The server has failed trying to process the request, which is theoretically correct.

The return phrase will depend on the implementation but is only used to clarify the return code.

After the status, we find a series of control fields in the same format as the headers of the request telling us the contents (creation date, length, server version etc). The requested contents then follow.

2.2.2. HTML language

The other basic factor in the success of the WWW is HTML (Hypertext Markup Language). This is a markup language (marks are inserted in the text) allowing us to represent rich content and to reference other resources (images, etc.), links to other documents (the most common feature of the WWW), display forms for subsequent processing etc.

HTML is currently in version 4.01 and is starting to offer advanced features for creating pages with richer contents. A specification compatible with HTML, XHTML (Extensible Hypertext Markup Language) has also been created, which

is usually defined as a validatable XML version of HTML, providing us with an XML Schema that can be used to validate the document to check that it is formed properly, etc.

3. History of web applications

Originally, the Web was simply a collection of static pages, documents, etc. that could be consulted and/or downloaded.

The next step in its evolution was the inclusion of a method to make dynamic pages allowing the displayed contents to be dynamic (generated or calculated from request data). This method was known as CGI (Common Gateway Interface) and defined a mechanism by which information could be passed between the HTTP server and external programs. CGIs are still widely used because they are straightforward and most web servers support them. They also give us complete freedom in choosing the programming language to develop them.

The operating schema of CGIs had a weak point: every time we received a request, the web server launched a process to run the CGI program. In addition, because most CGIs were written in an interpreted language (PERL, Python etc.) or a language that required run-time environment (VisualBasic, Java, etc.), it represented a heavy load for the server machine, if the web had several CGI accesses, this led to serious problems.

Hence, alternatives to CGIs began to be developed to solve this serious performance issue. Two main solutions were devised. Firstly, systems were designed for executing modules that were more integrated with the server so as to prevent the latter from having to instantiate and execute a multitude of programs. The other solution was to equip the server with a programming language interpreter (RXML, PHP, VBScript, etc.) allowing us to include the pages in the code so that the server could execute them, thus cutting down response time.

It was then that the number of architectures and programming languages allowing us to develop web applications skyrocketed. All used one of the above two solutions but the most common and widespread were those that combined the two, i.e. an integrated programming language allowing the server to interpret commands that we "embed" in HTML pages and a system for executing programs more closely linked with the server that does not have the performance problems of CGIs.

During this course, we will look in more detail at perhaps the most successful and powerful of these approaches, the one used by Sun Microsystems in its Java system, integrated by two components: a language allowing us to embed interpretable code in HTML pages, which the server translates to executable

programs, JSP (Java Server Pages) and a programming mechanism closely linked to the server with a performance far superior to conventional CGIs, called Java Servlet.

Another of the more successful technologies widely used on the Internet is the programming language interpreted by the PHP server. This language allows us to embed HTML in programs, with syntax from C and PERL and which, with its ease of learning, simplicity and power, is becoming a very widespread tool in some developments.

Other Web application programming methods also have their market, including moc_perl for Apache, RXML for Roxen, etc., but many are closely related to a specific web server.

Bibliography

Goodman, D. (1998). *Dynamic HTML. The Definitive Reference*. O'Reilly.

Musciano, C.; Kennedy, B. (2000). *HTML & XHTML: The Definitive Guide*. O'Reilly.

Raggett, D.; Lam, J.; Alexander, I.; Kmiec, M. (1998). *Raggett on HTML 4*. Addison Wesley Longman Limited. Chapter 2 available online at:http://www.w3.org/People/Raggett/book4/ch02.html.

Rosenfeld, L.; Morville, P. (1998). *Information Architecture for the World Wide Web*. O'Reilly.

World Wide Web (W3) Consortium (2003).*http://www.w3.org/Consortium/*. World Wide Web Consortium.

Annex

PID_00148406

Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Annex B. GNU Free Documentation License

B0.1. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of *copyleft* which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.2. Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The *Document* below refers to any such manual or work. Any member of the public is a licensee, and is addressed as *you* You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A *Modified Version* of the Document means any work containing the Document or aportion of it, either copied verbatim, or with modifications and/or translated into another language.

A *Secondary Section* is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The *Invariant Sections* are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The *Cover Texts* are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A *Transparent* copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not *Transparent* is called *Opaque*.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The *Title Page* means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such,*Title Page* means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section *Entitled XYZ* means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as *Acknowledgements*, *Dedications*, *Endorsements*, or *History*. To *Preserve the Title* of such a section when you modify the Document means that it remains a section *Entitled XYZ* according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

B.3. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.4. Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible.

You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.5. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled *History*. Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled *History* in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the *History* section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled *Acknowledgements* or *Dedications* Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled *Endorsements* Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled *Endorsements* or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as

invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled *Endorsements*, provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.6. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled *History* in the various original documents, forming one section Entitled *History*; likewise combine any sections Entitled *Acknowledgements*, and any sections Entitled *Dedications*. You must delete all sections Entitled *Endorsements*.

B.7. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

B.8. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an *aggregate* if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

B.9. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled *Acknowledgements*, *Dedications*, or *History*, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

B.10. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

B.11. Future revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Seehttp://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License or *any later version* applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

B.12. Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the *with...Texts*. line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Server installation

Carles Mateu

PID_00148400

Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Basic web server concepts

A web server is a program that deals with and responds to the diverse requests performed by browsers, providing the requested resources through HTTP or HTTPS protocol (the secure, encrypted and authenticated version of HTTP). A basic web server has a very straightforward schema of operation that executes the following loop infinitely:

1) It waits for requests on the assigned TCP port (the W3C recommendation port for HTTP is 80).
2) It receives a request.
3) It looks for the resource in the request string.
4) It sends the resource by the same connection through which it received the request.
5) It returns to step 2.

A web server following the above steps would meet the basic requirements of HTTP servers but could only serve static files.

The above schema is the basis for the design and building of all existing HTTP server programs, which vary only in the type of request (static pages, CGI, Servlets etc.) that they can serve depending on whether they are multi-process, multi-threaded, etc. We will now look in detail at some of the main features of web servers, which obviously expand on the above schema.

1.1. Static file service

All web servers must at least be able to serve the static files located on a specific part of the disc. One essential requirement is to be able to specify which part of the disc will be served. We do not recommend having the server force you to use a specific directory (although you can set up a default one).

Most web servers also allow you to add other directories to be served, specifying the point of the virtual "file system" on the server where they will be located.

For example, we can have the following situation:

Disc directory	Web directory
/home/apache/html /home/company/docs /home/joseph/report	/ /docs /report-2003

In this case, the server should translate the following web addresses as:

URL	Disc file
/index.html /docs/manuals/product.pdf /company/who.html /report-2003/index.html	/home/apache/html/index.html /home/company/docs/manuals/product.pdf/ /home/apache/html/company/who.html /home/joseph/report/index.html

Some web servers also allow us to specify security directives (for instance, the addresses, users, etc. for which a directory will be visible). Others allow us to specify which files will be considered the directory index.

1.2. Security and authentication

Most modern web servers allow us to control security and user authentication from the server program.

The simplest method of control is with the use of .htaccess files. This security system is derived from one of the first web servers (NCSA httpd) and involves placing a file called .htaccess in any web content directory to be served. In this file, we indicate which users, machines, etc. have access to the files and subdirectories of the directory in which the file is located. As the NCSA server was the most widespread server for a long time, most modern servers allow use of .htaccess files respecting the syntax of the NCSA server.

Others allow us to specify service rules for directories and files in the web server configuration, specifying in the latter the users, machines, etc. that can access the indicated resource.

As for authentication (validation of the username and password provided by the client), web servers offer a wide variety of features. At the very least, most allow us to provide the web server with a file containing the usernames and passwords to validate those sent by the client. In all events, it is common for servers to provide gateways allowing us to delegate the tasks of authentication and validation to different software (such as RADIUS, LDAP etc). If we use an operating system like Linux, which has an authentication infrastructure like PAM (Pluggable Authentication Modules), we can use this feature as a way to authenticate the web server. This enables us to use the many methods available in PAM to authenticate against diverse security systems.

1.3. Dynamic content

One of the most important aspects of the chosen web server is the level of support offered for serving dynamic content. Since most served web content is generated dynamically rather than coming from static pages, and this is a spiralling trend, the web server's support for dynamic content is one of the most critical factors to take into account when making your choice.

Most web servers offer CGI support (remember that CGIs are the oldest and most straightforward method of generating dynamic content). Many offer support for certain programming languages (basically interpreted), such as PHP, JSP, ASP, Pike, etc. We strongly recommend that the web server you use provides support for one of these languages (the most widespread being PHP), without taking into account JSP, which usually requires external web server software to work (such as a servlet container). There are many products available in this area but one basic consideration to bear in mind when choosing a server programming language is whether you want a very W3C standardised language so that your application will not need to depend on a specific web server or architecture, or whether portability is not a priority and, in contrast, the features of a given programming language are.

1.4. Virtual servers

One feature that is fast gaining supporters and users, particularly among Internet service providers and domain hosting companies, is the ability of some web servers to provide multiple domains with a single IP address, discriminating between the various hosted domains by the name of the domain sent in the header of the HTTP request. This feature allows for a more rational and economical administration of IP addresses, a resource in short supply.

If we require several server names (perhaps because we offer hosting or for some other reason), we need to make sure that the chosen web server provides these features and that the virtual server support allows us to use a different configuration for each server (directories, users, security etc). Ideally, each server will behave as though it were a different computer.

1.5. Extra features

Web servers offer many extra features to set themselves apart from the competition. Some are very useful and may influence our choice of web server. However, be aware that if you use any of these characteristics or they become essential for you, you could be forced to use a certain web server even though you may wish to change at some point in the future.

Some of the additional features of open source web servers include the following.

Spelling (Apache), this feature of Apache is used to define an error page for resources not found. It suggests similar names of resources to that requested by users in case they made a typing error.

Status (Apache), displays a Web page generated by the server displaying its operating status, response level, etc.

RXML Tags (Roxen), adds certain tags to HTML (HTML commands), improved for programming and generating dynamic content.

SQL Tags (Roxen), adds Roxen extended HTML (RXML) commands for access to SQL databases from the HTML pages.

Graphics (Roxen), adds Roxen extended HTML (RXML) commands to generate graphics, titles, etc., thus omitting the need for graphic design work.

bfnsgd (AOLServer), **mod_gd** (Apache), enables graphics to be produced from text and True Type fonts.

mod_mp3 (Apache) **ICECAST, MPEG** (Roxen), allows us to convert the web server into a music server (with streaming etc).

Throttle (Roxen), **mod_throttle** (Apache), offers means for limiting HTTP service speed, whether by the user, virtual server, etc.

nsxml (AOLServer), **tDOM** (AOLServer), **mod_xslt** (Apache), allows us to transform XML files using XSL.

Kill Frame (Roxen), sends a code with each served web page to stop the web from turning into a frameinside another web page.

1.6. Acting as representatives

Some web servers can be used as intermediate servers (proxy servers). Intermediate servers can be used for a range of purposes:

- As browsing accelerators for our users (use as proxy cache).
- As front-end accelerators for a web server. Using several web servers to replicate access to a master server (reverse proxy or HTTP server acceleration).
- As frontals for a server or protocol.

Some web servers can be used as intermediate servers for some of the above uses. Nonetheless, for the first two (browser or front-end accelerators), there are much more efficient specific free software programs, such as Squid (http://www.squid-cache.org/) which is considered one of the best proxy products available.

There are modules for diverse web servers that can be used as front-ends for other servers specialising in another type of service. For instance, Tomcat is an execution engine for servlets and JSP, and incorporates a small HTTP server to deal with static content requests and to redirect the rest to the servlet execution engine (web application development mechanisms, servlets and JSPs), but besides including a web server, Apache is the web server par excellence for use with Tomcat. Thus, there is an Apache module that links up with Tomcat (this module is called **mod_jk2**).

1.7. Additional protocols

Besides dealing with and serving HTTP (and HTTPS) requests, some servers can deal with and serve requests from other protocols or protocols implemented on HTTP. Some of these can become basic requirements of our system. Hence, their existence on the web server can be essential.

2. Apache server

Apache is a robust, free software web server implemented through a collaborative effort that offers equivalent features and functionality to commercial servers. The project is supervised and led by a group of volunteers from all over the world who use the Internet and web to communicate, plan and develop the server and its related documentation. These volunteers are known as Apache Group. In addition to Apache Group, hundreds of people have contributed to the project with code, ideas and documentation.

2.1. The birth of Apache

In February 1995, the most popular Internet web server was a public domain server developed at NCSA (National Center for Supercomputing Applications of the University of Illinois). However, when Rob McCool (the main developer of the server) left NCSA in 1994, the program's development was reduced to virtually nothing. Development then passed into the hands of people in charge of websites who gradually made improvements to their servers. A group of these individuals, using e-mail as the basic tool for their coordination, agreed to share these improvements (in the form of patches). Two of these developers, Brian Behlendorf and Cliff Skolnick, set up a mailing list, a space in which to share information and a server in California where the main developers could work. At the start of the following year, eight programmers formed what would become known as the Apache Group.

Using the NCSA 1.3 server as a basis for their work, they added all published error corrections and the most valuable improvements that they came across. They tested the result on their own servers and published what would be the first official version of the Apache server (0.6.2, in April 1995). Coincidentally, around the same time, NCSA resumed development of the NCSA server.

At this point in time, the development of Apache followed two parallel lines: one by the group of developers working on 0.6.2 to produce the 0.7 series, incorporate improvements, etc., and another where the code was completely rewritten to create a new modular architecture. In July 1995, the existing improvements for Apache 0.7 were migrated to this new architecture, which was made public as Apache 0.8.

On 1 December 1995, Apache 1.0 appeared, which included documentation and a number of improvements in the form of embedded modules. Shortly afterwards, Apache surpassed the NCSA server as the most widely used on the Internet, a position that it has maintained to this day. In 1999, the members of

the Apache Group founded the Apache Software Foundation, which provides legal and financial support to the development of the Apache server and the offshoots of this project.

2.2. Installing Apache

There are two main ways to install Apache: we can either compile the source code or we can install it from a binary package for our operating system.

2.2.1. Compiling from source

To compile Apache from source code, we must first obtain the latest version from the Apache site (http://httpd.apache.org). After downloading, you will need to follow these steps:

Decompress the file you have just downloaded, which will create a directory in which the server sources will be located.

Once inside this directory, the steps are as follows:

- Configure the code for compilation, for which you will need to execute:

```
$ ./configure
```

There are a number of parameters for adjusting the compilation of Apache. The most common important of these are:

Parameter	Meaning
`--prefix`	Directory where you wish to install Apache
`--enable-modules`	Modules to enable
`=LIST-MODULES`	Shared modules to be enabled
`--enable-mods-shared`	Dynamic cache
`=LIST-MODULES`	Dynamic cache on disc
`--enable-cache`	Cache module in memory
`--enable-disk-cache`	Automatic MIME detection
`--enable-mem-cache`	Monitoring of user session
`--enable-mime-magic`	Apache-proxy module
`--enable-usertrack`	Apache-proxy module to CONNECT
`--enable-proxy`	Apache-proxy module for FTP
`--enable-proxy-connect`	HTTP Apache-proxy module
`--enable-proxy-ftp`	SSL/TLS support (mod_ssl)
`--enable-proxy-http`	HTTP protocol handling
`--enable-ssl`	WebDAV protocol handling
`--enable-http`	Optimised CGI support
`--enable-dav`	CGI support
`--disable-cgid`	CGI support
`--enable-cgi`	Optimised CGI support
`--disable-cgi`	Virtual host support
`--enable-cgid`	
`--enable-vhost-alias`	

After configuring the source code, if no errors have been detected, it can now be compiled. To do so, execute:

```
$ make
```

Note that, at the very least, GNU Make and GNU CC are required to compile Apache.

After compiling, we can install it in the directory designated as the destination in the previous configuration with `configure`. This step is carried out using one of the objectives already defined for `make`. Specifically, we will use:

```
$ make install
```

Once installed in its location, in the bin subdirectory of the installation directory (the one we specified with `prefix`), we will find a program called `apachectl`, which we can use to control the server. To start it:

```
$cd <installation directory>/bin
$./apachectl start
```

To stop it:

```
$cd <installation directory>/bin
$./apachectl stop
```

2.2.2. Installation with binary packages

Most free software operating systems, particularly Linux distributions, include Apache server. However, it is often necessary to install Apache (either because we did not install it previously, we need a new version or because we need to reinstall it due to problems with a file).

Instructions for installing Apache on some of the most well known Linux distributions now follow.

Redhat/Fedora

Redhat and Fedora distributions have included Apache server for some time now, so the installation process is very straightforward.

From the appropriate server (either redhat.com or fedora.us), download the Apache binary package (in RPM format). Check that you are downloading the latest version for your distribution because both Redhat and Fedora publish updates to fix bugs or problems. Once you have the package, install it with:

```
rpm -ihv httpd-x.x.x.rpm
```

If it is already installed, you can upgrade with the command:

```
rpm -Uhv httpd-x.x.x.rpm
```

For Fedora, since this distribution uses an `apt` repository, Apache can be updated or installed using:

```
apt-get install httpd
```

You will also need to install any additional modules, such as:

- mod_auth_*
- mod_python
- mod_jk2
- mod_perl
- mod_ssl
- php
- etc.

Debian

Installing Apache on Debian is very easy. You simply need to execute the following command:

```
apt-get install apache
```

which will install Apache or, if it is already installed, update to the latest version.

2.3. Configuring Apache

After installing the server, you will need to configure it. By default, Apache comes with a minimum configuration to boot the server on the default TCP service port (port 80) and serves all files from the folder specified by the configuration directive `DocumentRoot`. Apache's configuration file is called `httpd.conf`, and is found in the conf subdirectory of the installation directory. The `httpd.conf` file is an ASCII file containing Apache's configuration directives.

2.3.1. Configuration file structure

The `httpd.conf` file is divided into three basic sections, although the directives of each section may seem mixed up and disorganised. These sections are:

- Global parameters
- Operating directives
- Virtual hosts

Some parameters are general for the server while others can be configured independently for all directories and/or files or for a specific virtual server. In these cases, the parameters are located in sections indicating the scope of application of the parameter.

The most important sections are:

<Directory>: the parameters located in this section will only be applied to the specified directory and its subdirectories.

<DirectoryMatch>: like Directory, but accepts regular expressions in the name of the directory.

<Files>: the configuration parameters control access to the files through their name.

<FilesMatch>: as for Files, but accepts regular expressions in the name of the file.

<Location>: controls file access through the URL.

<LocationMatch>: as for Location, but accepts regular expressions in the name of the file.

<VirtualHost>: the parameters only apply to the requests directed to this host (name of server or IP address or TCP port).

<Proxy>: the parameters only apply to the proxy requests (it therefore requires mod_proxy to be installed) matching the URL specification.

<ProxyMatch>: like Proxy, but accepts regular expressions in the specified URL.

<IfDefine>: applied if a specific parameter is defined in the command line (with the -D option) when booting the server.

<IfModule>: the parameters apply if the specified module is loaded (with `LoadModule`).

If there is a conflict between parameter specifications, the order of precedence is as follows:

1) `<Directory>` and `.htaccess`
2) `<DirectoryMatch>` and `<Directory>`
3) `<Files>` and `<FilesMatch>`
4) `<Location>` and `<LocationMatch>`

For `<VirtualHost>`, these directives are always applied after applying the general directives, so a `VirtualHost` can always overwrite the default configuration.

A configuration example would be:

```
<Directory /home/*/public_html>
   Options Indexes
</Directory>
<FilesMatch \.(?i:gif jpe?g png)$>
   Order allow,deny
   Deny from all
</FilesMatch>
```

2.3.2. Global configuration directives

Some configuration directives are never applied to any of the above sections (directories, etc.); they are directives that affect all web servers. The main ones are:

`ServerRoot`: specifies the location of the root directory in which the web server is located. From this directory, we can find the configuration files, etc. If the server is correctly installed, this should never be changed.

`KeepAlive`: specifies whether persistent connections will be used to deal with all requests from a user with the same TCP connection.

`Listen`: specifies the port where requests will be dealt with. By default, TCP port 80 is used. We can also specify which IP addresses will be used (if the server has more than one); by default all of those available are used.

`LoadModule`: with `LoadModule`, we can load the additional Apache modules on the server. The syntax is:

```
LoadModule module filemodule
```

We must have installed `mod_so` to be able to use it.

2.3.3. Main directives

There are some directives that are generally in the main configuration section, rather than those mentioned above (some of these cannot be in any section and must be in the main one). These are:

`ServerAdmin`: used to specify the e-mail address of the administrator. This address can appear as a contact address in error messages to allow users to report an error to the administrator. It cannot be inside any section.

`ServerName`: specifies the name and TCP port that the server uses to identify itself. These can be determined automatically but it is preferable to specify them. If the server has no DNS name, it is best to enter the IP address. It cannot be contained in a section. The syntax is:

`ServerName nameaddress:port` as in:

```
ServerName www.uoc.edu:80
ServerName 192.168.1.1:80
```

`DocumentRoot`: the root directory from which documents are served. By default, this is the htdocs directory, located in the Apache installation folder. It cannot be contained within any section except for `VirtualHost`. It has a `<Directory>` section in which the configuration parameters of this directory are set.

`Directorylndex`: specifies the file served by default for each directory if none are specified in the URL. By default, this is index.html. So, if we were to type www.uoc.edu in our browser, the server would send www.uoc.edu/index.html by default. More than one file may be specified and the order in which this name is indicated will determine the serving priority. The directive can be located either inside or outside any section.

`AccessFileName`: specifies the name of the configuration file if other than .htaccess. For this configuration to work, the AllowOverride directive must have the correct value. It cannot be inside any section. The default filename is `.htaccess`.

`ErrorDocument`: this directive establishes the server configuration in the event of an error. Four different configurations can be set:

- Display an error text
- Redirect to a file in the same directory
- Redirect to a file on our server
- Redirect to a file not on our server

The directive syntax is `ErrorDocument errorcode action`.

This directive can be located either in a section or in the global configuration, for example:

```
ErrorDocument 404 /notfound.html.
```

In a file is not found, the file notfound.html will be displayed.

Alias: the `Alias` and `AliasMatch` directives are used to define access to directories outside `DocumentRoot`. The syntax is as follows: `Alias url directory`

For example:

```
Alias /docs /home/documents
```

This will have a request served to http://www.uoc.edu/docs/manual from /home/documents/manual.

`UserDir`: this directive is used to tell Apache that a subdirectory of the working directory of the system users serves to store their personal page.

For example:

```
public UserDir
```

This will make the page stored in the user directory `test`, in the public subdirectory, accessible as:

```
http://www.uoc.edu/~test/index.html
```

2.3.4. Section directives

The configuration of most location sections (`Directory`, `Location`, etc.) includes a series of directives allowing us to control access to their contents. These directives are supplied by the module `mod_access`.

`Allow`: allows us to specify who is authorised to access the resource. We can specify IP addresses, computer names, parts of the name or address and even variables of the request. We can use the keyword `all` to indicate all clients.

`Deny`: allows us to specify who is not allowed to access the resource. The same options are available as for `Allow`.

`Order`: allows us to fine-tune the operation of the directives `Allow` and `Deny`. We have two options:

- `Allow`, `Deny`. Access is denied by default and only clients that meet the specifications of Allow and do not meet those of Deny are given access.

- `Deny, Allow`. Access is allowed by default and only clients that do not meet the specifications of Deny and do meet those of Allow are given access.

2.3.5. Virtual servers

Apache supports the serving of a number of websites with a single server. For this, it offers facilities for the creation of virtual domains based on diverse IP addresses or IP names.

Apache was one of the first servers to support virtual servers without IP, based on name instead. This considerably simplifies server administration and generates significant savings in IP addresses, which are normally in short supply. Virtual name servers are totally transparent for the client with the only possible exception of very old browsers, which do not send the `Host:` header with requests.

Virtual by IP address servers

To deal with several virtual servers, each with its own IP address, we need to use the configuration section called `VirtualHost`. In this section, we define each of the servers with its own configuration and IP address. An example of this would be:

```
<VirtualHost 192.168.1.1>
   ServerAdmin webmaster@uoc.edu
   DocumentRoot /web/uoc
   ServerName www.uoc.edu
   ErrorLog  /web/logs/uoc_error_log
   TransferLog /web/logs/uoc_access_log
</VirtualHost>
<VirtualHost 192.168.254.254>
   ServerAdmin webmaster@asociados.uoc.edu
   DocumentRoot /web/asociados
   ServerName asociados.uoc.edu
   ErrorLog /web/logs/asociados_error_log
   TransferLog /web/logs/asociados_access_log
</VirtualHost>
```

As we can see, this example defines two web servers, each with a different IP and name. Each has its own `DocumentRoot`, etc.

To use virtual IP servers, the server system must have the different IP addresses to be served configured in the operating system.

Virtual name servers

To deal with a number of servers all using the same IP address, we need to use the section called `Virtual Host`, which will allow us to define the parameters of each server. If our needs are the same as those in the example of virtual IP address servers with a single address, we should use the following configuration:

```
NameVirtualHost *:80
   <VirtualHost *:80>
      ServerAdmin webmaster@uoc.edu
      ServerName www.uoc.edu
      DocumentRoot  /web/uoc
      ErrorLog /web/logs/uoc_error_log
      TransferLog  /web/logs/uoc_access_log
   </VirtualHost>
   <VirtualHost *:80>
      ServerAdmin webmaster@uoc.edu
      ServerName asociados.uoc.edu
      DocumentRoot /web/asociados
      ErrorLog  /web/logs/asociados_error_log
      TransferLog /web/logs/asociados_access_log
   </VirtualHost>
```

We can use an IP address in place of * to assign, for example, a group of virtual name servers to this IP and another group to another.

We require a special use of name server directives when our server has two IP addresses but we have assigned the same name to both, for instance, when we have an intranet and an Internet connection with the same name. In this case we can serve the same contents as follows:

```
NameVirtualHost 192.168.1.1
NameVirtualHost 172.20.30.40
<VirtualHost 192.168.1.1 172.20.30.40>
   DocumentRoot /www/server1
   ServerName server.uoc.edu
   ServerAlias server
</VirtualHost>
```

This configuration can be used to serve the same Web page to the intranet and the Internet. Note the use of an alias for the server so that we do not have to use domains in the intranet.

There is also a default virtual server specification `_default_` for requests not served by another.

```
<VirtualHost _default_>
   DocumentRoot /www/default
</VirtualHost>
```

We can use `_default_` with a port number to specify different default servers for each port.

Apache also allows much more complex configurations of virtual servers, which is particularly useful for mass servers, etc. You will find an excellent reference guide on the Apache project website, along with useful advice and configuration recipes.

3. Other free software web servers

There are many free software HTTP servers, the majority of which have been eclipsed by the fame of Apache. Some of these have features making them very interesting.

3.1. AOLServer

AOLserver is the free software web server developed by AOL (America Online, the world's leading Internet provider). AOL uses AOLserver as the main web server for one of the web environments with the biggest traffic and Internet use. AOLserver is a multi-threaded TCL-based web server with many features for use in large-scale environments or dynamic websites. All AOL domains and web servers, more than two hundred, which give support to thousands of users, millions of connections, etc., use AOLserver.

AOLserver has a wide user base, thanks in particular to its integration with OpenACS, a very powerful free software content management system, initially developed by a company called ArsDigita and subsequently released under the GPL. The AOLserver-OpenACS tandem forms the infrastructure for complex and powerful web projects such as dotLRN (a virtual open source university campus).

3.2. Roxen and Caudium

Roxen is a web server published under GNU licence by a group of Swedish developers that later set up the company Roxen Internet Services. The Roxen server (previously Spinner and Spider) has always attracted attention for the many functionalities it offers to users. This server, developed in Pike programming language, offers hundreds of modules to users, allowing us to easily develop very rich, dynamic websites, etc. with no tools other than the Roxen server. The main features of Roxen are:

- Cross-platform, can run on a multitude of platforms: Windows, Linux, Solaris, MAC OS X, etc.
- Free software.
- A very rich and user-friendly web-based administration interface.
- Integrated graphic support that, with just a few RXML tags (Roxen HTML extension), allows the generation of images, titles, graphics, etc.

- Access to integrated databases, allows access to PostgreSQL, Oracle, MySQL, etc.

- Integrated MySQL database.

- Server programming with RXML, Java, Perl, PHP and CGIs.

- Excellent cryptographic support.

- Modular architecture allowing server extensions to be uploaded and downloaded when in operation.

- Platform independence for modules developed by the user.

In mid-2000, following the appearance of Roxen version 2.0, which ended the latter's compatibility with previous versions, particularly 1.3 (the most widely used), a group of developers, including some of the founders of Roxen, began a new project based on Roxen version 1.3 with the aim of developing a web server that maintained compatibility with the latter. This web server is called Caudium. At the present time, both Roxen and Caudium have a promising future, good relations (their developers try to maintain compatibility between the APIs of the two systems) and a loyal user base.

Roxen is one of the few examples of an excellent product that has always featured among the fastest and most stable web servers with the most features and facilities but which has not achieved success because it was always eclipsed by Apache.

3.3. thttpd

thttpd is an extremely small, very fast, portable and secure HTTP server. It offers the same features as conventional servers such as Apache but its performance is far superior under extreme loads.

Its use as a general-purpose web server is rather less widespread, usually being limited instead to acting as a rapid server of static content, often supporting Apache servers for serving static binary content such as images, etc., leaving the dynamic or more complex pages for the Apache server. As an auxiliary of Apache for serving static content, it has managed to reduce the load of the main server to a hundredth of its original load.

3.4. Jetty

Jetty is a web server written entirely in Java that also incorporates a servlets container. It is small and high-performance, making it one of the most preferred for developing embedded products that require a HTTP server. Although Jetty servers are rarely found operating in isolation, we do often come across them as web servers embedded in products. For example:

- Integrated with application servers such as JBoss and Jonas.
- Integrated into the JTXA project as the basis for HTTP transport.
- Integrated into products such as IBM Tivoli, Sonic MQ and Cisco SESM as a HTTP server.
- On most demo CDs in books on Java, servlets, XML, etc.
- Running on multiple embedded systems and pocket PCs.

4. Practical: installing a web server

4.1. Exercise

2-1 Download the Apache server code from the Internet and install it in a subdirectory of your user directory. Make sure you install the most recent version and that you have correctly installed the following modules:

- mod_access
- mod_cgi

2-2 Configure the server you have installed to respond to HTTP requests on port 1234.

2-3 Configure the web server to serve the documents located in the web subdirectory of the user's working directory.

2-4 Configure the web server to run CGI programs from the `cgi` directory of the user's working directory.

4.2. Solution

2-1 After obtaining the Apache source code, you need to decompress it:

```
   [carlesm@bofh m2]$ tar xvzf httpd-2.0.48.tar.gz
httpd-2.0.48/
httpd-2.0.48/os/
httpd-2.0.48/os/os2/
httpd-2.0.48/os/os2/os.h
httpd-2.0.48/os/os2/core.mk
httpd-2.0.48/os/os2/config.m4
httpd-2.0.48/os/os2/Makefile.in
httpd-2.0.48/os/os2/core_header.def
....
httpd-2.0.48/include/ap_release.h
httpd-2.0.48/include/.indent.pro
httpd-2.0.48/include/util_cfgtree.h
httpd-2.0.48/acconfig.h
   [carlesm@bofh m2]$
```

Once you have the source code in your directory, you can configure it for compilation. First of all, you need to tell Apache where to install it. In this case, we chose the `apache` subdirectory of our working directory.

You also need to make sure that the required modules have been included.

```
[carlesm@bofh m2]$ cd httpd-2.0.48
[carlesm@bofh httpd-2.0.48]$ ./configure \
          --prefix=/home/carlesm/apache \
          --enable-cgi
checking for chosen layout... Apache
checking for working mkdir -p... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
....
creating srclib/pcre/Makefile
creating test/Makefile
config.status: creating docs/conf/httpd-std.conf
config.status: creating docs/conf/ssl-std.conf
config.status: creating include/ap_config_layout.h
config.status: creating support/apxs
config.status: creating support/apachectl
config.status: creating support/dbmmanage
config.status: creating support/envvars-std
config.status: creating support/log_server_status
config.status: creating support/logresolve.pl
config.status: creating support/phf_abuse_log.cgi
config.status: creating support/split-logfile
config.status: creating build/rules.mk
config.status: creating include/ap_config_auto.h
config.status: executing default commands
[carlesm@bofh httpd-2.0.48]$
```

Then comes the moment to start compiling:

```
   [carlesm@bofh httpd-2.0.48]$ make
Making all in srclib
make[1]: Entering directory '/srcs/httpd-2.0.48/srclib'
Making all in apr
make[2]: Entering directory '/srcs/httpd-2.0.48/srclib/apr'
Making all in strings ....
make[1]: Leaving directory '/srcs/httpd-2.0.48'
```

If compilation is successful, you will be able to install Apache in your chosen directory:

```
   [carlesm@bofh httpd-2.0.48]$ make install
```

```
Making install in srclib
make[1]: Entering directory '/srcs/httpd-2.0.48/srclib'
Making install in apr
make[2]: Entering directory '/srcs/httpd-2.0.48/srclib/apr'
Making all in strings
....
mkdir /home/carlesm/apache/man
mkdir /home/carlesm/apache/man/man1
mkdir /home/carlesm/apache/man/man8
mkdir /home/carlesm/apache/manual
Installing build system files
make[1]: Leaving directory '/srcs/httpd-2.0.48'
   [carlesm@bofh httpd-2.0.48]$ cd  /home/carlesm/apache/
   [carlesm@bofh apache]$ ls
bin      build      cgi-bin      conf      error      htdocs
icons    include    lib          logs      man        manual
modules
   [carlesm@bofh apache]$
```

You must then configure Apache with the requested parameters. To do this, edit the `/home/carlesm/apache/conf/httpd.conf` file and change the following parameters:

```
Listen 1234
ServerAdmin admin@uoc.edu
DocumentRoot "/home/carlesm/web"
<Directory "/home/carlesm/web">
   Options Indexes FollowSymLinks
   AllowOverride None
   Order allow,deny
   Allow from all
</Directory>
ScriptAlias /cgi-bin/ "/home/carlesm/cgi/"
<Directory "/home/carlesm/cgi">
   AllowOverride None
   Options None
   Order allow,deny
   Allow from all
</Directory>
```

Bibliography

Laurie, Ben; Laurie, Peter (2002). *Apache: The Definitive Guide, 3rd Edition*. O'Reilly.

Bowen, Rich; Lopez Ridruejo, Daniel; Liska, Allan (2002). *Apache Administrator's Handbook*. SAMS.

Wainwright, Peter (2002). *Professional Apache 2.0*. Wrox Press.

Redding, Loren E. (2001). *Linux Complete*. Sybex.

Web page design

Carles Mateu

PID_00148397

Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

HTML (HyperText Markup Language) is used to create documents with a hypertext structure. A hypertext document contains information that is cross-referenced with other documents, allowing us to switch from the first document to the cross-referenced one from the same application being used to view it. HTML can also be used to create multimedia documents, i.e. those containing information that is not merely textual. For example,

- Images
- Video
- Sound
- Active subprograms (plug-ins, applets)

HTML is not the only language available for creating hypertext documents; there are languages that came before and after HTML (SGML, XML, etc.), but HTML has become the W3C recommendation language for creating content for the Internet.

1. Basic HTML

HTML documents are created as plain text documents (with no special formatting) in which all text formatting is specified using textual marks (called tags) that delimit the content affected by the tag (start and end tags are used).

These tags are textual marks that begin with the character <, followed by the name of the tag and any additional attributes, and end with the character >. So, initial tags look like this:

```
<TAG>
```

End tags start with the character <, followed by the character /, followed by the name of the tag and the character >. So, end tags look like this:

```
</TAG>
```

Tags are case-insensitive. Examples of HTML tags include:

```
<title>Name of document</title>
<P>Example of the use of tags to mark text</P>
<B>Bold<I>Italics</I>Bold</B>
```

Tag attributes, which indicate additional tag parameters, are included in the start tag as follows:

```
<TAG ATTRIBUTE ATTRIBUTE...>
```

The form of these attributes is either the name of the attribute or the name of the attribute followed by =, followed by the value we want to assign it (generally inside inverted commas). For example:

```
<A HREF="http://www.w3.org">Link</A>
<IMG SRC="image.jpg" BORDER=0 ALT="NAME">
```

In some cases, HTML can omit the end tag if it does not need to surround the text that it affects (as is the case of `IMG`). Another important point to note is that if the WWW client we use (the browser we are using) does not understand a tag, it will be ignored along with all of the text affected by it.

1.1. Structure of HTML documents

All HTML documents have more or less the same structure. The whole document needs to be contained within a `HTML` tag and is split into two: the header, contained in a `HEAD` tag and the body of the document (containing the document information), which is contained within a tag called `BODY`. The header contains some definitions of the document: its title, extra formatting marks, keywords, etc.

One example might be:

```
<HTML>
   <HEAD>
      <title>Document title</TITLE>
   </HEAD>
   <BODY>
      Text of document
   </BODY>
</HTML>
```

If we open a document with these contents in a browser, we will see that the text inside the `TITLE` tag is not displayed in the document; instead the browser displays it in the title bar of the window.

1.1.1. Comments

In HTML, we can enter comments on the page with the tags `<!--` and `-->`. The content inside these two marks is ignored by the browser and is not displayed to the user.

1.2. Blocks of text

There are several types of blocks of text in HTML:

- Paragraphs
- Line breaks
- Quoted blocks
- Divisions
- Pre-formatted text
- Centred text

1.2.1. Paragraphs

The <P> tag is used to separate paragraphs. Since HTML ignores line breaks entered in the original file and the entire text is continuous for HTML, we need a mechanism to indicate the start and end of paragraphs; this mechanism is provided by <P> and </P>.

The P tag can also have an attribute, ALIGN, indicating the alignment of the text in the paragraph. This can be one of the following values:

LEFT, aligned to the left; this is the default behaviour.
RIGHT, aligned to the right.
CENTER, centred text.

The end of paragraph mark, </P>, is optional in W3C recommendation HTML and can be omitted. If this is the case, the browser will take a new <P> to indicate the end of the previous paragraph.

1.2.2. Line breaks

The <BR> tag indicates a line break. It can be used as an initial mark and does not require an end tag. BR does not modify the parameters specified for the paragraph in which we are located at this time.

1.2.3. Horizontal rules

HTML has a tag for including a horizontal rule on our page (a line drawn from one side of the page to the other) with a variable width. This tag, HR, is used to separate blocks of text. This element only has an initial label but comes with several attributes for adapting its appearance:

- NOSHADE: eliminates the shadow effect of the bar.
- WIDTH: defines the length of the line in relation to the page.
- SIZE: defines the thickness of the line.

1.2.4. Quoted paragraphs

HTML has an element called the BLOCKQUOTE that allows us to represent paragraphs quoted literally from another text. These are generally indented or extended to the left and have a paragraph break before and after the quoted paragraph. We should avoid using BLOCKQUOTE to indent text, reserving it for literal quotations because the browser may represent this in other ways, e.g. by not indenting.

1.2.5. Dividing text into blocks

The <DIV> element is used to divide text into blocks by inserting a single line between the blocks like BR, although it can have the same attributes as P, i.e. we can define the alignment of the text for each DIV block.

The alignments supported by DIV with the ALIGN parameter are:

- LEFT, aligned to the left; this is the default behaviour.
- RIGHT, aligned to the right.
- CENTER, centred text.

1.2.6. Pre-formatted text

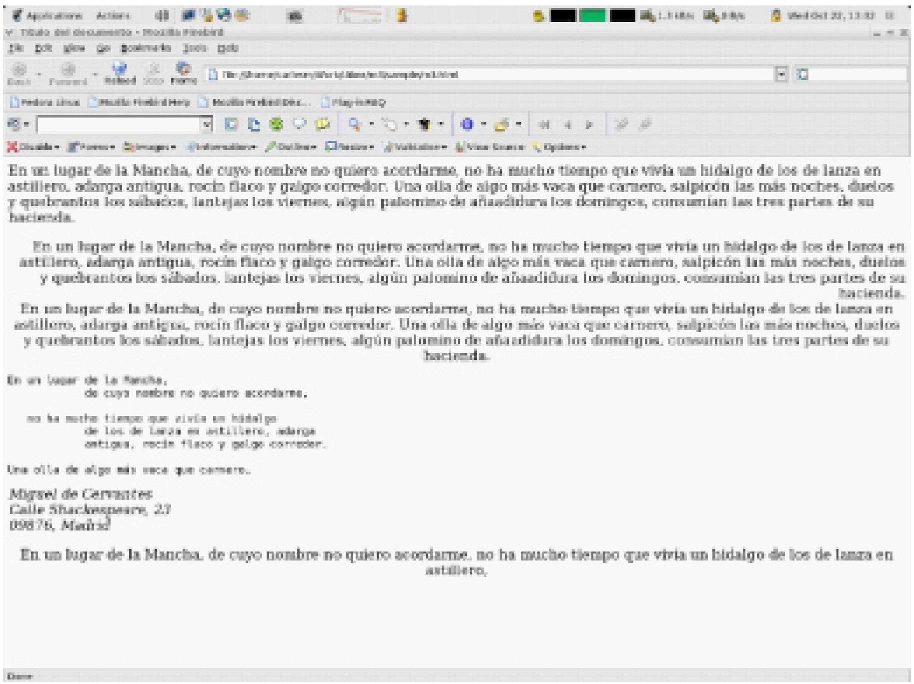
The text inserted between the <PRE> and </PRE> tags will be displayed by the browser respecting the format of the line breaks and spaces used to enter it. Browsers generally display this text with a fixed-width typeface similar to that of a typewriter.

We can see some of these tags in the following example:

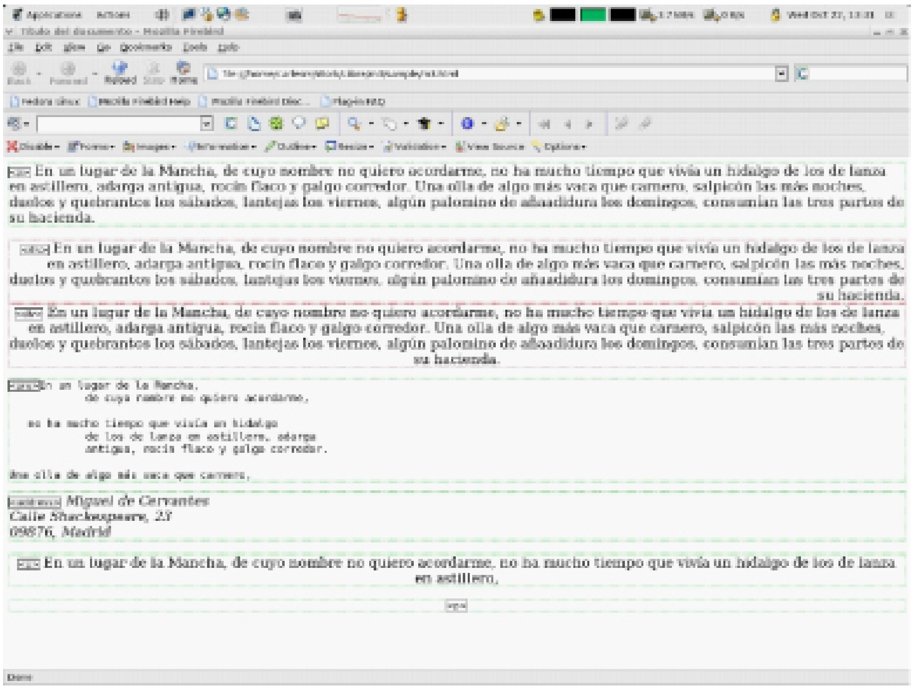
```
<HTML>
   <HEAD>
      <TITLE>Document title</TITLE>
   </HEAD>
   <BODY>
<P ALIGN=LEFT>
In a village of La Mancha, the name of which I have no desire to call to mind, there lived not
long since one of those gentlemen that keep a lance in the lance-rack, an old buckler, a lean
hack, and a greyhound for coursing. An olla of rather more beef than mutton, a salad on most
nights, scraps on Saturdays, lentils on Fridays, and a pigeon or so extra on Sundays, made away
with three-quarters of his income.
</P>
<DIV ALIGN=RIGHT>
In a village of La Mancha, the name of which I have no desire to call to mind, there lived not
long since one of those gentlemen that keep a lance in the lance-rack, an old buckler, a lean
hack, and a greyhound for coursing. An olla of rather more beef than mutton, a salad on most
nights, scraps on Saturdays, lentils on Fridays, and a pigeon or so extra on Sundays, made away
with three-quarters of his income.
</DIV>
<DIV ALIGN=CENTER>
In a village of La Mancha, the name of which I have no desire to call to mind, there lived not
long since one of those gentlemen that keep a lance in the lance-rack, an old buckler, a lean
hack, and a greyhound for coursing. An olla of rather more beef than mutton, a salad on most
nights, scraps on Saturdays, lentils on Fridays, and a pigeon or so extra on Sundays, made away
with three-quarters of his income.
</DIV>
```

```
<PRE>
In a village of La Mancha,
               the name of which I have no desire to call to mind,
   there lived not long since one of those gentlemen
         that keep a lance in the lance-rack, an old buckler,
         a lean hack, and a greyhound for coursing.
An olla of rather more beef than mutton,
</PRE>
<ADDRESS>
   Miguel de Cervantes<BR>
   Shakespeare Street, 23<BR>
   09876, Madrid<BR>
</ADDRESS>
   <CENTER>
      <P>
In a village of La Mancha, the name of which I have no desire to call to mind, there lived not
long since one of those gentlemen that keep a lance in the lance-rack,
      </P>
   </CENTER>
</BODY>
</HTML>
```

This HTML code will be displayed as follows:



One of the utilities provided by some free software browsers like Mozilla or Firebird is that they show the block elements that make up a web page; in this case, our example would be seen as follows:



1.3. Logical tags

HTML also has a group of tags for formatting the text not as we wish to show it but by giving the format based on the semantics of this block of text, which allows the browser to display the text in the most appropriate manner.

These tags are:

- <CITE>: literal quotation from a text or document.
- <ADDRESS>: address.
- <SAMP>: example of code or result.
- <CODE>: program code.
- <KBD>: data that needs to be typed in.
- <VAR>: variable definition.
- <DFN>: definition of text or a word (there is little browser support for this option).

These tags will be formatted differently according to the browser and how we have configured it. This example shows how they look in the Mozilla browser:

The C Programming Language, Ritchie, Dennis; Kernigahn, Ritchie, AT&T Bell Labs

Our address is:

10, Downing Street, London.

The files ending with the extension .jpg are image files.

```
printf(”Hello World\n”);
```

Once you have entered the system, type startx to boot…

We will define the variable *neigh* to save…

A *Distributed-CSP* is a problem the solution of the…

© 2003, Carles Mateu

The code that produced this result is:

```
<HTML>
   <HEAD>
      <TITLE>Document title</TITLE>
   </HEAD>
   <BODY>
<P><CITE>The C Programming Language</CITE>,
Ritchie, Dennis; Kernighan, Ritchie. AT&T Bell Labs.
<P> Our address is:
<ADDRESS>
10, Downing Street.
London.
</ADDRESS>
<P>
Files ending in the extension
<SAMP>jpg</SAMP> are image files.
<P>
<CODE>printf("Hello World\n");</CODE>
<P>After entering, type  <KBD>startx</KBD> to boot...
<P>We will define the variable <VAR>neigh</VAR> to save...
<P>A <DFN>Distributed-CSP</DFN> is a problem, of...
<P><CITE>© 2003, Carles Mateu</CITE>
   </BODY>
</HTML>
```

1.4. Fonts

HTML includes tags for changing attributes of our texts such as font and colour. HTML also has certain special tags called character entities that allow us to enter special characters such as the copyright symbol, accents etc. where these are not supported by our keyboard, text editor, character set, etc.

1.4.1. Headers

There is an element called <Hx> that we can use to define the parts of our text that need to be considered as headers (section, chapter, etc.) The tag assigns a larger text size to the characters (based on x, as we shall see), uses a bold typeface for the header and inserts a paragraph break after this header.

The header size (or level or index of importance of the latter) can vary from 1 to 6, so there are six possible tags: H1, H2, H3, H4, H5 and H6.

1.4.2. Font

HTML has a tag for dealing with typefaces. This tag, FONT, is obsolete in HTML 4.01, so you should avoid using it and try to use style sheets (CSS) instead. FONT this is used to specify:

- Measurements, with the SIZE attribute
- Colours, with the COLOR attribute
- Typefaces, with the FACE attribute

Be cautious about using this tag to specify typefaces because your client may not have this particular typeface installed and the page will not be viewed as you had planned.

The attributes supported by FONT, used to define font characteristics, are:

- SIZE: character size, with values from 1 to 7 or relative values (-7 to +7).
- COLOR: colour of the characters.
- FACE: typeface to use; you can indicate more than one, separated by commas.

The SIZE attribute defines the size of font in relation to the default document size, which is defined using BASEFONT. BASEFONT has just one parameter, SIZE, used to set the base size for the document.

1.4.3. Font styles

HTML has a set of tags that can be used to define different letter styles for the text inside the tags. The available tags are:

B (bold).
I (italics).
U (underlined).
STRIKE (strikethrough).
SUP (superscript).
SUB (subscript).
BLINK (blinking).

TT (teletype).
BIG (big).
SMALL (small).

Besides these physical typefaces, there are also some logical typefaces, which browsers may prefer to represent in another way:

EM (emphasised).
STRONG (highlighted).
CODE (program code).
CITE (quotation).
KBD (keyboard entry).
SAMP (example).
VAR (program variable).

Some of these logical styles also introduce a paragraph style, which we saw earlier.

With HTML, we can mix different styles such as bold and italics, etc. In this case, the corresponding HTML tags are nested:

```
<B><I>Bold and italics</I></B>
```

We can see how these typefaces and colours look on the next page:

Heading H1
Heading H2
Heading H3
Heading H4
Heading H5
Heading H6

Letter size

1 2 3 4 5 6 7 6 5 4 3 2 1

Colours LETTER COLOURS

Bold
Italics
Underlined
~~Strikethrough~~
A **Superscript**
B **Subscript**

Typewriter (Teletype)
Large text
Small text

The HTML code that produced this result is:

```
<HTML>
```

```
   <HEAD>
      <TITLE>Document title</TITLE>
   </HEAD>
<BODY>

<h1>Header H1</h1>
<h2>Header H2</h2>
<h3>Header H3</h3>
<h4>Header H4</h4>
<h5>Header H5</h5>
<h6>Header H6</h6>

<b>Font size</b> <BR>
<font SIZE=1>1</font> <font SIZE=2>2</font>
<font SIZE=3>3</font> <font SIZE=4>4</font>
<font SIZE=5>5</font> <font SIZE=6>6</font>
<font SIZE=7>7</font> <font SIZE=6>6</font>
<font SIZE=5>5</font> <font SIZE=4>4</font>
<font SIZE=3>3</font> <font SIZE=2>2</font>
<font SIZE=1>1</font>
<P>
<B>Colours</b>
<font COLOR=#800000>C</font><font COLOR=#000080>O</font>
<font COLOR=#000080>L</font><font COLOR=#008000>O</font>
<font COLOR=#00FFFF>R</font><font COLOR=#FF0000>E</font>
<font COLOR=#C0C0C0>S</font> . <font COLOR=#800080>D</font>
<font COLOR=#008080>E</font> . <font COLOR=#FF0000>L</font>
<font COLOR=#808080>E</font><font COLOR=#FF00FF>T</font>
<font COLOR=#00FF00>R</font><font COLOR=#808000>A</font>
<font COLOR=#FFFF00>S </font>

<P> <b>Bold</b> <br> <i>Italics</i> <br> <u>Underlined</u><br>
<strike>Strikethrough</strike> <br> A<sup>Superscript</sup> <br>
B<sub>Subscript</sub><br>  <blink>Blinking</blink> <br>
<tt>Typewriter(Teletype)</tt>  <BR> <big>Big
text</big>  <br> <small>Small text</small>

</BODY>
</HTML>
```

1.4.4. Character entities

HTML has a series of special codes called character entities, used to type characters that cannot be entered with the keyboard, such as accents, circumflexes, special symbols, etc. We can also use special character entities to type any character from the ISO-Latin1 character table.

Code	Result
á, Á, é, É,...	á,Á,é,É,...
¿	¿
¡	¡
º	º
ª	ª
™ or ™	Trademark symbol
©	Copyright symbol
®	Registered symbol
	(non-breaking space)
<	<
>	>
&	&
"	"

1.5. Links

One of the key features of the Web that has had the greatest impact on its success is its hypertextual nature, i.e. the possibility of intuitively and transparently linking documents that may be located on different servers. Links can be made to images, audio, video, etc. as well as to web pages.

We can create links using a tag called A and its set of attributes, `NAME`, `HREF`, `TARGET`, affording us total control over link creation in documents.

There are four types of main link:

- Links within a page
- Links to other pages on our system
- Links to pages from other system
- Links to documents consulted through other protocols (e-mail, etc.)

1.5.1. Links

To create a link, we need to use the A tag with the attribute `HREF`. The value of this attribute will be the destination of the link:

```
<A HREF="destination">Text or image</A>
```

The contents of the tag are given special consideration and displayed differently by the browser (generally by underlining). When we click on this text, we will be taken to the destination indicated by the value of the `HREF` attribute, which must be a URL.

1.5.2. Destinations

A destination is a URL address indicating a service we wish to obtain or a resource we wish to access. The format for URLs is as follows:

service://user:password@server:port/resourcepath

Several services can be indicated in the URL and these will be accepted by most browsers:

http: indicates the web page transfer service and is in everyday use.
https: indicates a secure and encrypted HTTP service.
ftp: indicates that we need to use the file transfer protocol, FTP. If we do not enter a username and password, anonymous transfer will be attempted. If this fails, we will be asked for the username and password.
mailto: indicates that an e-mail should be sent to the specified address.
news: access to the USENET news service.

Examples of URLs include:

```
http://www.uoc.edu
https://www.personales.co/usuarios/carles/indice.html
ftp://user:secret@ftp.cesca.es/pub/linux
mailto:destination@e.mail.co
news://noticias.uoc.edu/es.comp.os.linux
```

Destinations within a page

One of the possibilities of HTML is that of jumping to destinations within the same page. To do this, we need to define the destinations on the page, called anchors with a name. To do so, we can use the NAME attribute of the A tag. For example:

```
<A NAME="anchor">
```

Once we have defined the anchors in our documents, we can either browse through or go directly to them. To browse these anchors, we will use a URL extension such as:

```
<A HREF="http://www.uoc.edu/manual.html\#anchor">Link</A>
```

If we create this link on the same page, we can abbreviate the address to:

```
<A HREF="\#anchor">Link</A>
```

1.6. Lists

In HTML we can define three main types of lists and numberings:

- Unordered lists
- Ordered (numbered) lists
- Lists of definitions

1.6.1. Unordered lists

To enter unordered lists, we can use the <UL> tag to indicate the start of the list, the </UL> tag to indicate the end of the list and <LI> to indicate each of the items in the list.

We can also use the TYPE attribute to indicate the marker to use to highlight the various items: DISC, CIRCLE, SQUARE.

All of the items must be entered between <LI> and </LI>.

1.6.2. Ordered (numbered) lists

Ordered lists are used in a very similar way to unordered lists. This time, we can use the <OL> tag to indicate the start of the list, the </OL> tag to indicate the end of the list and <LI> to indicate each of the items in the list.

We can also use the TYPE attribute to indicate the marker to use to number the various items:

TYPE=1 Numbers (the default option).
TYPE=A Upper-case letters.
TYPE=a Lower-case letters.
TYPE=I Upper-case Roman numerals.
TYPE=i Lower-case Roman numerals.

We can also use the START attribute to indicate the point at which line numbering should begin. The TYPE attribute can be used in the individual items too.

All of the items must be entered between <LI> and </LI>.

1.6.3. Lists of definitions

A list of definitions is a non-numbered list that allows us to give a description or definition of each element. The descriptive lists are formed with the tags: <DL> and </DL> to define the list, <DT> to indicate the term to be defined and DD to indicate the definition.

For DL, we can use the COMPACT attribute, which tells the browser to display the list in the most compact way possible by putting the term and its definition on the same line.

The different examples of HTML lists can be seen in this diagram:

- First element
- Second element
- Third element

1. First element
2. Second element
C. Third element

ASCII
: 7-bit character set. Only 127 characters are allowed.

EPS
: Encapsulated PostScript format

PNG
: Portable Network Graphics, highly efficient graphic format.

The HTML code that produced this result is:

```
<HTML>
   <HEAD>
   <TITLE>Document title</TITLE>
   </HEAD>
<BODY>
<UL>
   <LI>First element
   <LI>Second element
   <LI>Third element
</UL>
<P>

<OL>
   <LI>First element
   <LI>Second element
   <LI TYPE=A>Third element
</OL>
<P>

<dl compact>
   <dt>ASCII  <dd>
7-bit character set.
Only 127 characters allowed.
   <dt>EPS <dd>
Encapsulated PostScript Format.
   <dt>PNG<dd> Portable Network Graphics,
   high efficiency graphics format.
</dl>

</BODY>
</HTML>
```

1.7. Images

A single tag is used to include graphics and images on our pages: <IMG>.

<IMG> has several attributes for specifying the image file to use, its measurements, etc.

The attribute for specifying the image to display is SRC. With this tag, we can specify a URL for the image file that will be requested from the server by the browser in order to display it.

The images referenced with the SRC attribute can be located in any directory on the server, on other servers, etc. The value we enter for SRC must be a URL.

We can also use the ALT attribute to assign an alternative text to the image if the browser cannot show it. In this case, the browser will display this alternative text to the user.

In addition, there are several attributes allowing us to specify the image measurements, width and height, WIDTH and HEIGHT. If these are not specified, the browser will display the image at the size of the actual image file. If we specify the measurements, the browser resizes the image to suit. Using image measurement parameters allows the browser to leave the space taken up by the image and display the rest of the page while the images are loading.

Images are commonly used as buttons for links. In this case, the browser will generally add a border to distinguish it from the rest of the text. You can prevent this effect by adding a further attribute, BORDER, which is used to specify the thickness of this border. To remove it, change the value to zero.

IMAGE DOES NOT EXIST

The HTML code that produced this screen is:

```
<HTML>
   <HEAD>
      <TITLE>Document title</TITLE>
   </HEAD>
<BODY>

<IMG SRC="logo.gif"> <P>
```

```
<IMG SRC="nologo.gif" ALT="IMAGE DOES NOT EXIST"><P>

</BODY>
</HTML>
```

1.8. Tables

HTML has a group of tags that can be used to enter text in table form. The tags for this feature are:

- TABLE: marks the start and end of the table.
- TR: marks the start and end of a row.
- TH: marks the start and end of a header cell.
- TD: marks the start and end of a cell.
- CAPTION: used to insert titles in tables.

The code for a simple table might be:

```
<TABLE>
   <TR><TH>Header 1</TH>...<TH>Header n</TH></TR>
   <TR><TD>Cell 1.1</TD>...<TD>Cell n</TD></TR>

   ...

   <TR><TD>Cell 1.1</TD>...<TD>Cell n</TD></TR>
   <CAPTION>Title</CAPTION>
</TABLE>
```

As we can see, the table is enclosed by TABLE tags. Each table row needs to be contained between the <TR> and </TR> tags. We have two options for displaying cells in individual rows: we can either enclose them in <TH> tags or in <TD> tags. The difference is that the first option selects a bold typeface and centres the column.

1.8.1. The <TABLE> tag.

The TABLE tag has some attributes that can be used to specify the exact format to give to the table.

- BORDER: indicates the size of the cell borders.
- CELLSPACING: indicates the size in points of the space between cells.
- CELLPADDING: indicates the distance in points between the contents of a cell and its borders.
- WIDTH: specifies the width of the table. This can be in points or in relation to the percentage of the total available width. For example, 100% indicates the entire browser window.

- ALIGN: aligns the table in relation to the page, to the left (LEFT), right (RIGHT) or middle (CENTER).
- BGCOLOR: specifies the background colour of the table.

1.8.2. The <TR> tag.

The TR tag can be used to enter the rows making up the table. TR has the following attributes:

- ALIGN: aligns the content of the cells in a row horizontally to the left (LEFT), right (RIGHT) or middle (CENTER).

- VALIGN: aligns the content of the cells in a row vertically along the top (TOP), bottom (BOTTOM) or middle (MIDDLE).

- BGCOLOR: specifies the background colour of the row.

1.8.3. The <TD> and <TH> tags

The TD and TH tags are used to add the cells that will make up the row where they are located. The main difference between the two is that TH horizontally centres the cell contents and displays them in bold. Both tags can have the following attributes:

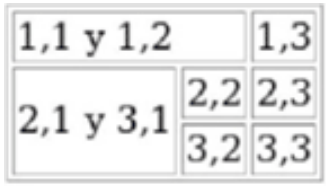
- ALIGN: aligns the content of the cells in a row horizontally to the left (LEFT), right (RIGHT) or middle (CENTER).
- VALIGN: aligns the content of the cells in a row vertically along the top (TOP), bottom (BOTTOM) or middle (MIDDLE).
- BGCOLOR: specifies the background colour of the cell.
- WIDTH: specifies the width of the cell in points or as a percentage; in the latter case, remember that this is the width of the table rather than the window.
- NOWRAP: stops the line inside cells from being divided by spaces.
- COLSPAN: indicates how many cells to the right including the current one will be merged to form a single one. If COLSPAN is zero, all cells to the right will be merged.
- ROWSPAN: indicates the number of column cells below the current one will be merged with the latter.

1.8.4. The <CAPTION> tag.

This is used to add a centred legend or title above or below a table. It has just one attribute:

`ALIGN`: this indicates where the `CAPTION` tag will be located in relation to the table. The possible values are: `TOP`, places it above the table, and `BOTTOM`, which places it below.

Two HTML tables can be seen in the image:



1,1 y 1,2		1,3
2,1 y 3,1	2,2	2,3
	3,2	3,3

Simple Table

UOC	April	May	June	July
Vehicles	22	23	3	29
Visitors	1234	1537	7	1930
Income	11000	13000	-500	60930

The HTML code that produced this result is:

```
<HTML>
   <HEAD>
      <TITLE>Document title</TITLE>
   </HEAD>
   <BODY>

   <TABLE BORDER=1>
      <TR>
         <TD COLSPAN=2>1.1 and 1.2</TD>
         <TD>1.3</TD>
      </TR>
      <TR>
         <TD ROWSPAN=2>2.1 and 3.1</TD>
         <TD>2.2</TD>
         <TD>2.3</TD>
      </TR>
      <TR>
         <TD>3.2</TD>
         <TD>3.3</TD>
      </TR>
      <CAPTION ALIGN=bottom>Simple Table</CAPTION>
   </TABLE>

   <HR>
```

```
<TABLE BORDER=0 CELLSPACING=0 BGCOLOR=#0000FF>
<TR><TD>
<TABLE BORDER=0 CELLSPACING=1 CELLPADDING=2
   WIDTH=400 BGCOLOR=#FFFFFF>
   <TR>
      <TH><IMG SRC="logo.gif"></TH>
      <TH>April</TH>
      <TH>May</TH>
      <TH>June</TH>
      <TH>July</TH>
   </TR>
   <TR>
      <TD BGCOLOR=#A0A0A0>Vehicles</TD>
      <TD>22</TD>
      <TD>23</TD>
      <TD>3</TD>
      <TD>29</TD>
   </TR>
   <TR>
      <TD BGCOLOR=#A0A0A0>Visitors</TD>
      <TD>1234</TD>
      <TD>1537</TD>
      <TD BGCOLOR=#FFa0a0>7</TD>
      <TD>1930</TD>
   </TR>
   <TR>
      <TD BGCOLOR=#A0A0A0>Income</TD>
      <TD>11000</TD>
      <TD>13000</TD>
      <TD BGCOLOR=#FF4040>-500</TD>
      <TD BGCOLOR=#a0a0FF>60930</TD>
   </TR>
</TABLE>
</TD></TR>
</TABLE>
</BODY>
</HTML>
```

1.9. Forms

Forms are HTML elements used to collect user information. A variety of form elements are available, allowing for rich and efficient interaction with users. In all events, forms do not process the information entered by users. We will need to process this ourselves later through other means (CGI, JSP, Servlets etc).

One way to create a form is as follows:

```
<FORM ACTION="url process" METHOD="POST">
...
Elements
..
</FORM>
```

The FORM tag provides us with certain attributes:

- ACTION: this attribute is used to specify the URL where the data that the user types into the form will be sent. An e-mail address can be used as the URL, for example:

```
mailto:address@e.mail
```

or we can enter a HTTP URL (the most common method for sending data to CGI programs):

```
http://www.uoc.edu/proceso.cgi
```

- METHOD: the method specifies the way in which the data is sent. We are offered two options: GET and POST. We will look at these options in detail later when we discuss CGI programming.

- ENCTYPE: specifies the type of encoding used. It is generally only used when the form result is sent by e-mail and changes the encoding to text/plain.

- NAME: used to assign a name to the form, which will be necessary later for using with JavaScript.

1.9.1. Form elements

HTML provides us with a wide variety of input elements for forms. These can be used to carry out a range of functions, including typing in text and sending files.

The <INPUT> Elements

The INPUT element is perhaps the most widely known and used of form elements and is used as an input field. There are different types of INPUT element, depending on the value of the TYPE attribute:

- TYPE=RADIO: allows us to choose from a range of options but only one from those of the same name.

- TYPE=RESET: clears the entire form.

- TYPE=TEXT: allows the user to enter a line of text.

- TYPE=PASSWORD: allows the user to enter a line of text, displaying characters such as "*" instead of the text. This is generally used where passwords must be typed in.

- TYPE=CHECKBOX: allows us to choose from one or more options.

- TYPE=SUBMIT: accepts the data entered in the form and carries out the specified action.

- TYPE=HIDDEN: text field not visible to the user. Used to store values.

The INPUT element also has some optional attributes:

- NAME: names the field. This is important for subsequent processing with our programs.
- VALUE: assigns an initial value to the field.
- SIZE: size of fields, where applicable. TEXT and PASSWORD.
- MAXLENGTH: maximum length allowed for user input TEXT and PASSWORD fields).
- CHECKED: for RADIO or CHECKBOX, indicates whether they are marked or unmarked by default.

The SELECT Elements

The SELECT element is used to select one or more of the available options. An example of a SELECT element would be:

```
<SELECT name="destination">
   <option> Africa
   <option> Antarctica
   <option> America
   <option> Asia
   <option> Europe
   <option> Oceania
</SELECT>
```

The attributes of the SELECT element are:

- SIZE: the on-screen size of the SELECT element If 1, only one option will be displayed and SELECT will operate as a drop-down list. If greater than 1, the user will be presented with a selection list.

- MULTIPLE: users can choose more than one option if this is selected.

The OPTION element has two attributes:

- VALUE: the value that will be assigned to the variable when this option is selected.
- SELECTED: this option will be selected by default.

The TEXTAREA element

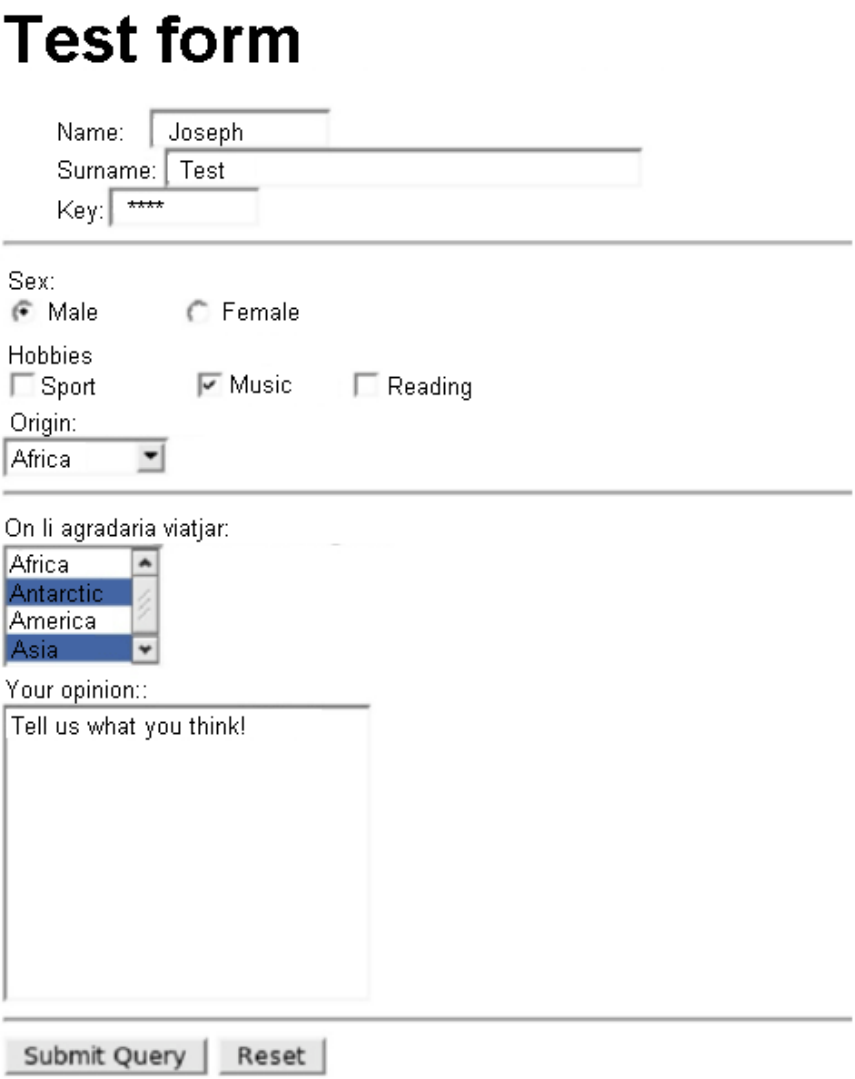
The TEXTAREA element is used to obtain multiple-line text elements from the user. The format is as follows:

```
<TEXTAREA name="comments" cols=30 rows=6>
Enter comments about the page
</TEXTAREA>
```

Note that the contents enclosed by <TEXTAREA> and </TEXTAREA> are considered to be the initial value of the field. The attributes for TEXTAREA are:

- ROWS: the rows that will be taken up by the text box.
- COLS: the columns that will be taken up by the text box.

We will now look at an example of this basic form, built with the above elements.



The HTML code that produced this result is:

```
<HTML>
   <HEAD>
      <TITLE>Document title</TITLE>
   </HEAD>
<BODY>
<H1>Form test</H1>

<FORM METHOD=GET>
Name: <INPUT TYPE=TEXT NAME=NAME SIZE=10><BR>
Surname: <INPUT TYPE=TEXT NAME=SURNAME SIZE=30><BR>
Password: <INPUT TYPE=PASSWORD NAME=PASS SIZE=8><BR>
<HR>
Gender: <BR>
<INPUT TYPE="RADIO" NAME="Gender">Male
<INPUT TYPE="RADIO" NAME="SEXO">Female
<BR>
```

```
Hobbies:<BR>
<INPUT TYPE="CHECKBOX" NAME="SPORT">Sport
<INPUT TYPE="CHECKBOX" NAME="MUSICA">Music
<INPUT TYPE="CHECKBOX" NAME="LECTURA">Reading <BR>
Origin:<BR>
<SELECT name="ORIGIN">
   <option> Africa
   <option> Antarctica
   <option> America
   <option> Asia
   <option> Europe
   <option> Oceania
</SELECT>
<HR>
Where would you like to travel:<BR>
<SELECT name="destination" MULTIPLE SIZE=4>
   <option> Africa
   <option> Antarctica
   <option> America
   <option> Asia
   <option> Europe
   <option> Oceania
</SELECT>
<BR>
Your opinion:
<BR>
<TEXTAREA COLS=25 ROWS=10 NAME="YOUR OPINION">
Tell us what you think!
</TEXTAREA>
<HR>
<INPUT TYPE=SUBMIT> <INPUT TYPE=RESET>
</FORM>
</BODY>
</HTML>
```

2. Advanced HTML

2.1. Style sheets

Style sheets are a mechanism for separating the format for representing and presenting contents. This is done by associating presentation attributes to each HTML tag or its subclasses.

For example, if we want all of the paragraphs in our document (defined by <P></P>) to have a red background and yellow text, we would use the following definition:

```
<STYLE TYPE="text/css"> P {color: red; background:yellow;} </STYLE>
```

To indicate which styles we need to use on a page, the STYLE tag can be used to specify them in situ, while the LINK tag allows us to indicate an external file containing our styles.

The STYLE tag must be located in the page header. The TYPE setting is used to indicate the syntax we will use to define the styles, which, in our case, will be text/css. The LINK tag, used to define an external style sheet, looks like this:

```
<LINK REL="stylesheet" HREF="miweb.css" TYPE="text/css">
```

In fact, use of the LINK tag is highly recommended when defining the style sheets associated with a page as this will facilitate maintenance because all the styles of a site will be concentrated into a single file instead of being repeated on each page.

2.1.1. Style sheet format

As we saw in the previous examples, the format of style sheets is as follows:

```
<element>{<format>}
```

For example:

```
P {color: red; background:yellow;}
```

Please,note that CSS syntax is case sensitive.

This syntax will allow us to define the format we would like for the paragraphs in our website. There is an extension for this syntax used to define a style that will only be applied to parts of the document. Specifically, it allows us to define `classes` of elements to which the style will be applied. For example, to define a paragraph class that we will call highlighted:

```
P.highlighted {color: red; background:yellow;}
```

We can then use the `CLASS` attribute that HTML 4.0 added to HTML to define the class of each paragraph:

```
<P CLASS="highlighted">A highlighted paragraph</P>
<P>A normal paragraph</P>
<P CLASS="highlighted">Another highlighted one</P>
```

There is also a method for assigning a style to individual paragraphs, thus offering more granularity to the class concept. For this, we need to define the style of an individual HTML element with CSS using the following syntax:

```
#paragraph1 {color: green; background:yellow;}
```

We can then assign this identity to an HTML element using the `ID` attribute:

```
<p CLASS="destacado">A highlighted paragraph</P>
<P>A normal paragraph</P>
<P CLASS="highlighted" ID="paragraph1">Another highlighted one but but the colour here
is assigned by its identity</P>
```

2.1.2. The `SPAN` and `DIV` tags

Earlier, we saw how to assign styles to HTML elements (paragraphs, etc.), but we sometimes need to assign styles to sections of text or content that do not form part of an HTML block. For example, we may want to define a style that would allow us to mark specific words of text (to indicate changes, for instance). Obviously, we cannot define a new HTML tag as the existing browsers, which would not be familiar with our tag, would ignore this content. The solution comes in the form of the `DIV` tag, which we saw earlier, and `SPAN` tag.

If we want to mark a section of content as belonging to a specific class in order to define a style for it or to assign individual identification to it, we will need to wrap this content inside `SPAN` or `DIV`. The difference between them is that `DIV` ensures that there is a line break at the start and end of the section. This allows us to define blocks of text without having to enclose them in tags that would modify their format (such as `P`).

For example, we could define:

```
all.unsure{color: red; } all.revised { color:blue;  }
```

and then use it in our HTML document:

```
<P>This <SPAN CLASS="unsure">long</SPAN> paragraph must be
reviewed by the <SPAN CLASS="reviewed">CEO</SPAN>
</P>
```

2.1.3. More important properties

We will now look at the more important properties that can be defined using CSS. Given the incompatibilities between different browsers, we recommend testing your pages with different browsers and different versions to make sure that they display properly.

Typeface properties

The properties allowing us to define the appearance (typeface) of the text are:

- `font-family`: font (which can be generic from among: `serif`, `cursive`, `sans-serif`, `fantasy` and `monospace`). We can specify single fonts or a list of fonts, whether generic or otherwise, separated by commas. Be aware when specifying fonts that they may not be installed on the computer of the user visiting your page.

- `font-size`: size of the font. `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large` and the numerical relative or absolute size values.

- `font-weight`: thickness of the font. The possible values are: `normal`, `bold`, `bolder`, `lighter` and numerical values from 100 to 900 (where 900 is the thickest bold font).

- `font-style`: style of font. We can use `normal`, `italic`, `italic small caps`, `oblique`, `oblique small caps` and `small caps`.

2.1.4. Text properties

There is also a group of properties used to alter the text on the page and its format.

- `line height`: line spacing as a numerical or percentage value.

- `text decoration`: decoration of the text: `none`, `underline`, `overline`, `line-through` and `blink`.

- `vertical-align`: vertical alignment of the text. This can be: `baseline` (normal), `sub` (subscript), `super` (superscript), `top`, `text-top`, `middle`, `bottom`, `text-bottom` or a percentage.

- `text-transform`: text modification: `capitalize` (initial in upper case), `uppercase` (converts the text to upper case), `lowercase` (converts it to lower case) or `none`.

- `text-align`: horizontal alignment of the text: `left`, `right`, `center` or `justify`

- `text-indent`: indentation of the first line of text in absolute or percentage values.

2.1.5. Block properties

The following properties affect blocks of text (paragraphs etc).

- `margin-top`, `margin-right`, `margin-bottom`, `margin-left`: minimum distance between a block and the adjacent elements. Possible values: size, percentage or `auto`.

- `padding-top`, `padding-right`, `padding-bottom`, `padding-left`: fills in the space between the border and contents of the block. Possible values: size as an absolute value, percentage or `auto`.

- `border-top-width`, `border-right-width`, `border-bottom-width`, `border-left-width`: width of the block border in numerical values.

- `border-style`: style of the block border. `none`, `solid` or `3D`.

- `border-color`: colour of the block border.

- `width`, `height`: measurements of the block. Values as a percentage, absolute values or `auto`.

- `float`: justification of a block's content. Values: `left`, `right` or `none`.

- `clear`: the other elements are positioned in relation to the current one. Possible values: `left`, `right`, `both` or `none`.

2.1.6. Other properties

There are other style sheet properties that can be used to change other aspects:

- `color`: text colour.
- `background`: background colour or image. Values, a colour or a URL of the image file.

```
background: url(nicebackground.gif);
```

URLs in CSS

In CSS, the format for URLs is as follows: url(address)

- `display`: decides whether or not an element has a block character. This can be: `inline` (such as `<I>` or `<B>`), `block` such as `<P>`, `list` such as `<LI>` or `none`, which disables the element.

- `list-style`: style of marker of an element of a list (allowing us to use graphics as markers). Possible values: `disc`, `circle`, `square`, `decimal`, `lower-roman`, `upper-roman`, `lower-alpha`, `upper-alpha`, `none` or a URL of an image.

- `white-space`: indicates how blank spaces should be treated, whether as usual or whether they should be respected as in the block `<PRE>`. Values: `normal` and `pre`.

2.2. Layers

HTML 4.0 introduced a new concept to increase our control over the positioning of elements on our pages. We can now define layers as pages embedded within other pages.

We can specify the attributes of these layers (position, visibility, etc.) using style sheets, just like other HTML elements. Layers, which can be controlled with programming languages like JavaScript, are the basis of what we now know as dynamic HTML. Unfortunately, the implementations of the different browsers are incompatible between each other, so we either need to use huge volumes of program code to cover all possibilities or to limit ourselves to using only the common minimums. One of the few options we have to make layers work in the majority of browsers is to define them using CSS style sheets.

The following example shows how to add a layer that we will call `thelayer` using the `ID attribute`.

```
<STYLE TYPE="text/css">
   #thelayer  {position:absolute; top:50px; left:50px;}
</STYLE>
```

In this example, `thelayer` would be placed 50 points from the upper left-hand corner of the page. To define the layer, in this case, we will use a SPAN tag.

```
<SPAN ID="thelayer">
   ...
   Content of the layer
   ...
</SPAN>
```

In this example, we positioned the previous layer in a specific position on the page; we can also place layers in relative positions in relation to the position that the text would occupy on the page where they are written.

The definition for this is as follows:

```
<STYLE TYPE="text/css"> #flotlayer  {position: relative; left: 20px; top: lOOpx;} </STYLE>
```

There are several specific layer properties that can be easily modified:

- `left`, `top`: these are used to indicate the position of the upper left-hand corner of the layer in relation to the layer where it is located. The whole document is considered a layer.

- `height`, `width`: indicate the height and width of the layer.

- `clip`: allows us to define a clipped area inside the layer.

- `z-index`: indicates the depth in the stack of layers. The greater the `z-index`, the shallower the depth and the greater the visibility (they will be superimposed on those with smaller `z-indexes`). By default, the `z-index` is assigned by order of definition in the HTML file.

- `visibility`: specifies whether the layer should be visible or hidden. The possible values are `visible`, `hidden` or `inherit` (inherits the visibility of the parent layer).

- `background-image`: Image that will be used as the background of the layer.

- `background-color`, `layer-background-color`: defines the background colour of the layer for Internet Explorer and Mozilla/Netscape, respectively.

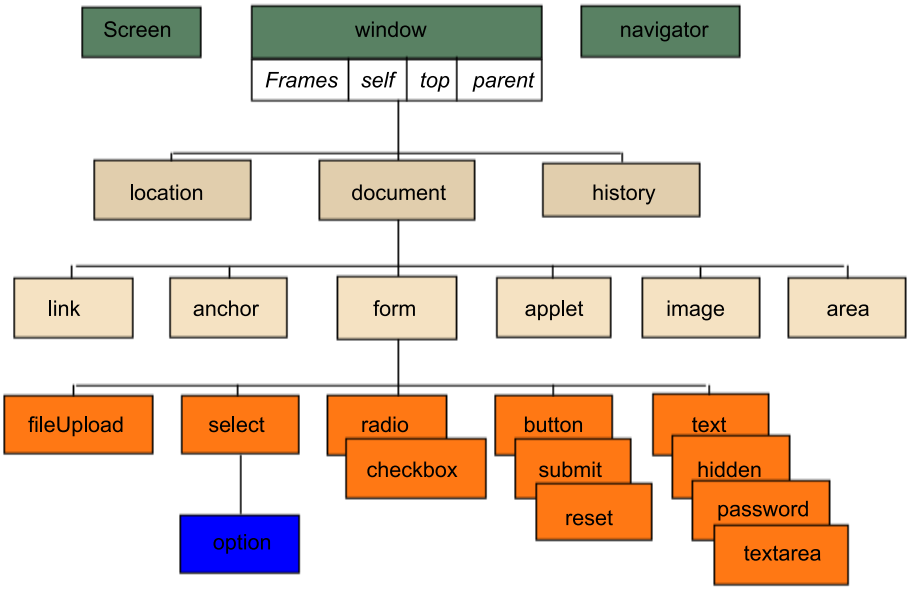
3. Dynamic HTML

Dynamic HTML (DHTML) is not a recommendation defined by the W3C; it is a marketing term used by Netscape and Microsoft to refer to new Web technologies as a whole. These include:

- HTML, particularly HTML 4.0
- Style sheets (CSS)
- JavaScript

These technologies are generally known as DHTML, especially where they work together to add to the user's web experience. Among other things, this combination of technologies offers much richer and more complex graphic user interfaces, the possibility of controlling forms more efficiently (JavaScript code is executed on the client, resulting in enhanced performance), etc.

One of the key features of DHTML is DOM (Document Object Model), which defines a hierarchy of objects accessible through JavaScript (a tree in fact) representing each and every element in the HTML document. The tree used in DOM is as follows:



We will now look at an example of how to define a form in HTML that uses controls directed by JavaScript and DOM to handle a TEXTAREA element.

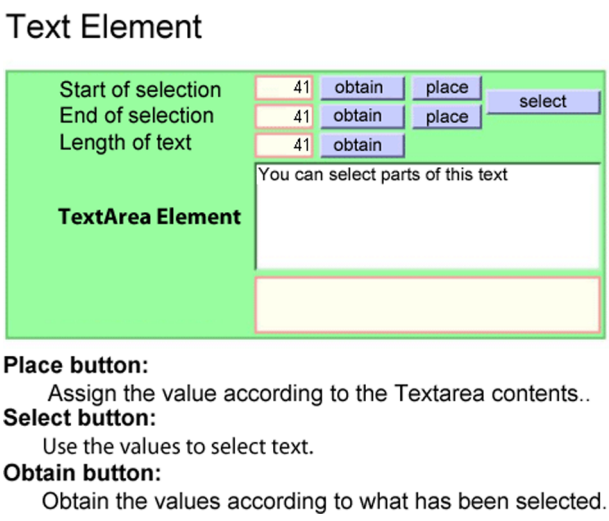
```
<html>
   <head>
      <meta http-equiv="Content-Type"
       content="text/html; charset=ISO-8859-1">
```

```
        <title>Textarea</title>
        <meta name="Author" content="ShinSoft">
        <meta http-equiv="Content-Script-Type" content="text/javascript">
        <meta http-equiv="Content-Style-Type" content="text/css">
        <style type="text/css">
        <!--
table{ background-color:#99ff99; border:2px solid #66cc66; }
td textarea { background-color:#ffffff; border:2px inset #66cc66; width:100 %; }
td input[type="button"]{ background-color:#ccccff; border:2px outset #9999ff; }
td input[type="text"] { background-color:#ffffee; border:2px solid #ff9999; text-align:right; }
[readonly]{ color:#999966; }
dt { font-weight:bold; font-family:fantasy; }
#t { background-color:#ffffee; border:2px solid #ff9999;  }
   -->
   </style>

<script language="JavaScript">
<!--
function notSupported(){ alert('No browser support.'); }
function setSel(){
   var f=document.f;
   var t=f.ta;
   if(t.setSelectionRange){
      var start=parseInt(f.start.value);
      var end =parseInt(f.end .value);
      t.setSelectionRange(start,end);
      t.focus();
      f.t.value = t.value.substr(t.selectionStart, t.selectionEnd-t.selectionStart);
   } else notSupported();
} function setProp(id){
   var f=document.f;
   var t=f.ta;
   if(id==0) t.selectionStart = parseInt(f.start.value);
   if(id==1) t.selectionEnd = parseInt(f.end .value);
   f.t.value = t.value.substr(t.selectionStart,
    t.selectionEnd-t.selectionStart);
   t.focus();
}
function getProp(id){
   var f=document.f; var t=f.ta; if(id==0) f.start.value = t.selectionStart;
   if(id==1)
      f.end.value = t.selectionEnd;
   if(id==2)
      f.txl.value = t.textLength;
   f.t.value = t.value.substr(t.selectionStart,
    t.selectionEnd-t.selectionStart);
   t.focus();
```

```
   }
function init(){
   var f=document.f;
   var t=f.ta;
   if(t.setSelectionRange){
      f.start.value = t.selectionStart
      f.end .value = t.selectionEnd;
      f.txl .value = t.textLength;
   } else notSupported();
}
// -->
</script>
</head>
<body bgcolor="#ffffff" text="#000000"
   link="#cc6666" alink="#ff0000" vlink="cc6666"
    onload="init();">
<h2>Textarea Element</h2>
<form name="f">
<table border=0 cellspacing=1>
   <tr>
   <th>Start of selection</th>
   <td>
      <input type="text" name="start" size=4 value="0">
      <input type="button" value="obtain" onClick="getProp(0);">
      <input type="button" value="place" onClick="setProp(0);">
   </td>
   <td rowspan=2>
      <input type="button" value="Select" onClick="setSel();">
   </td>
   </tr>
   <tr>
   <th>End of selection</th>
   <td>
      <input type="text" name="end" size=4 value="1">
      <input type="button" value="obtain" onClick="getProp(1);">
      <input type="button" value="place" onClick="setProp(1);">
   </td>
   </tr>
   <tr>
   <th>Length of text</th>
   <td>
      <input type="text" name="txl" id="txl" size=4 value="" readonly>
      <input type="button" value="obtain" onClick="getProp(2);">
   </td>
   </tr>
   <tr>
   <th>TextArea Element</th>
```

```
    <td colspan=2>
    <textarea name="ta" id="ta" cols=30 rows=5>
    We can select parts of this text
    </textarea></td></tr>
    <tr>
       <th>selected string</th>
       <td colspan=2>
       <textarea name="t" id="t" readonly></textarea>
       </td>
    </tr>
</table>
</form>

<dl>
<dt>Place button:</dt>
<dd>Assign the value according to the Textarea contents.</dd>
<dt>Select button:</dt>
<dd>Use the values to select text.</dd>

<dt>Obtain button:</dt>
<dd>Obtain the values according to what has been selected.</dd>
</dl>

</body></html>
```

The result that will appear in the browser is as follows:



4. JavaScript

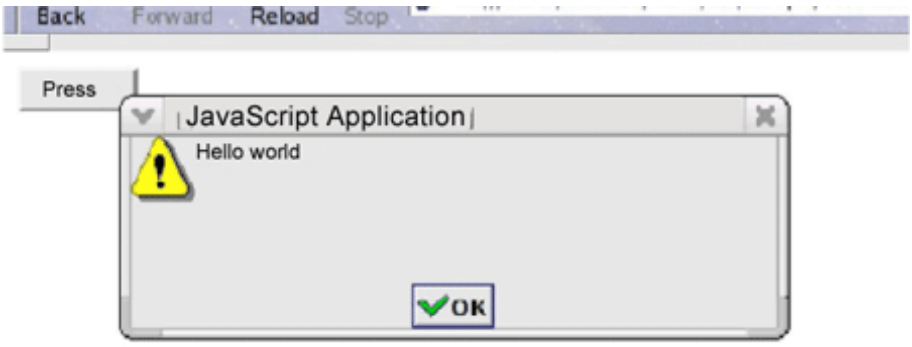
JavaScript is an interpreted programming language (a script language). Although there are interpreters that do not depend on a given browser, it is a script language usually linked to web pages. JavaScript and Java are two different programming languages with very different philosophies. The only thing they have in common is their syntax, since Netscape based its design of JavaScript on the syntax of Java.

4.1. First basic program

As is now the norm when demonstrating programming languages, our first contact with JavaScript will be to create our first program displaying the typical "Hello world" message. Since JavaScript is a language that is generally linked to a web page, the following code will be a HTML file that we will need to display in a browser.

```
<HTML>
   <HEAD>
      <SCRIPT LANGUAGE="Javascript">
         function Greeting()
         {
            alert("Hello world");
         }
      </SCRIPT>
   </HEAD>
   <BODY>
      <FORM>
         <INPUT TYPE="button" NAME="Button"
               VALUE="Press" onClick="Greeting()">
      </FORM>
   </BODY>
</HTML>
```

This JavaScript program paints a button on our screen; this button opens a window with a message when we click on it. The program looks like this:



We will now discuss the above code so that we can introduce the diverse elements of JavaScript.

As we can see, the JavaScript code is wrapped by `<SCRIPT>` tags. These tags can appear at whichever point in the document we wish; they do not have to appear in the header.

Browsers that do not offer JavaScript support will ignore the content of the tags. Optionally, we may require our users to have a specific version of JavaScript if we use tags like:

```
<SCRIPT LANGUAGE="Javascriptl.1">
...
</SCRIPT>
```

An easy way to use the `<SCRIPT>` tags is to put them in the page header, as this makes for a more legible HTML code.

The `<SCRIPT>` tag contains the JavaScript code. In this case, it only has one function, but it could have more.

Our code:

```
function Greeting() { alert("Hello world"); }
```

defines a function called `Greeting`. As we can see, in contrast to Java, this function does not belong to an object. Despite being object-oriented, JavaScript allows functions to exist outside of objects (like C++).

We will see that the only code contained by this function is a call to function, `alert` (a window `object method`).

The following block of JavaScript code is inside the HTML definition of the form.

```
<FORM>
   <INPUT TYPE="button" NAME="Button"
```

```
    VALUE="Press" onClick="Greeting()">
</FORM>
```

In this case, the JavaScript code declares an event manager, specifically for the `onClick` event, for the object button. An event is an occurrence of something (in this case a mouse click by the user). When the JavaScript event takes place, it executes the code indicated in the `onClick` event manager. This code is a call to a function, `Greeting`.

4.2. Basic elements of JavaScript

JavaScript sentences end in ; (like C and Java) and can be grouped into blocks delimited by { and }.

Another point to bear in mind is that symbols (names of variables, functions, etc.) are case-sensitive.

4.2.1. Comments

There are two options for adding comments to the program:

```
// Single-line comment

/*
   comment that takes
   up several lines
*/
```

As you can see, the format of comments is identical to Java.

4.2.2. Literals

JavaScript follows the same mechanism as Java and C for defining literals, i.e. it has the following literals:

- **Integers** 123
- **Real** 0.034
- **Boolean** true, false
- **Strings** "Text string"

JavaScript also offers vector support:

```
seasons = ["Autumn"," Winter"," Spring"," Summer"];
```

Special characters

Like Java, JavaScript uses certain character sequences for inserting special characters in our string constants.

Within these strings, we can indicate several special characters with special meanings. The most widely-used are:

Character	Meaning
`\n`	New line
`\t`	Tab
`\'`	Inverted comma
`\"`	Quotation marks
`\\`	Backslash
`\xxx`	The ASCII number (Latin-1 code) of the hexadecimal character

4.3. Data types and variables

In JavaScript, data types are dynamically assigned as we assign values to the different variables. These can be:

- character strings
- integers
- real
- Boolean
- vectors
- matrices
- references
- objects

4.3.1. Variables

In JavaScript, the names of variables begin with an alphabetical character or the character '_', and may be formed by alphanumeric characters and the character '_'.

There is no need for an explicit declaration of variables as they are global. If you want a local variable, however, you will need to declare it using the reserved word `var` and do so in the body of a function. In a variable declaration with `var` we can declare several variables by separating their names with, .

The variables will take the data type from the type of data object we assign to them.

4.3.2. References

JavaScript eliminates pointers to memory from the language, but maintains the use of references. References work in a very similar way to pointers to memory, except that they skip out memory management tasks for programmers, which make pointers so prone to errors in other languages.

JavaScript allows references to objects and to functions. This ability to reference functions will be very useful when using functions that hide differences between browsers.

```
function onlyExplorer()
{
   ...
}
function onlyMozilla()
{
   ...
}

function all()
{
   var function;
   if(browserMozilla)
      function=onlyMozilla;
   else
      function=onlyExplorer;

   function();
}
```

4.3.3. Vectors

Javascript has a type of data for handling collections of data. The elements in these `arrays` can vary.

As we can see in the code below, JavaScript `arrays` are objects (`array` type) whose index of access may be a non-numerical value for which we should not initially declare the measurements. We do not have an n-dimensional type of vector, so we can use vectors of vectors for this.

```
//we dimension a vector of 20
vector elements = new Array(20);

//the vector grows to house the 30
myWonderfulVector elements  [30] = "contained";
```

```
//we dimension a capitals
vector = new Array ();

//we can use strings as capital indices
vector["France"] = "Paris";
```

4.3.4. Operators

Javascript has the same operators as Java and C, and behaves in the same way as these languages usually do:

- **Arithmetical operators**: the usual arithmetical operators are available (+, -, *, /, %, etc.), as well as increment (+ +) and decrement (-) operators.

- **Comparison operators**: we can use the following:
  - Equality ==
  - Inequality !=
  - Strict equality ===
  - Strict inequality !==
  - Less than <
  - Greater than >
  - Less than or equal to <=
  - Greater than or equal to <=

- **Logical operators**: JavaScript has the following logical operators:
  - Not !
  - and &&
  - or ||

- **Object operators**: for object handling we also have:
  - Create an object `new`
  - Delete an object `delete`
  - Reference to the current object `this`

4.4. Control structures

Like all programming languages, JavaScript has some control structures.

4.4.1. Conditional forks

JavaScript offers the two best-known control structures:

```
if (condition)
   <code>
else
   <code>
```

```
switch(value)
{
   case valuetest1:
      <code>
      break;
   case valuetest2:
      <code>
      break;
   ...
   default:
      <code>
}
```

4.4.2. Loops

There are three loops, `while` and `do`, `while,` and the `for loop`.

```
while(condition)
 <code>

do
 {
 <code>
 } while(condition);

for(start; condition; increase)
 <code>
```

4.4.3. Object handling structures

There are two very specific structures for object handling. Firstly, we have the `for..in` loop, which allows us to cycle through the properties of an object (generally in vectors):

```
for (<variable> in <object)
 <code>
```

Secondly, we have `with`, which is very convenient when dealing with multiple properties of a single object. We can write:

```
with (object)
{
   property1 = ...
   property2 = ...
}
```

Instead of:

```
object.property1=...
object.property2=...
```

4.5. Functions

JavaScript incorporates the necessary constructions for defining functions. The syntax is as follows:

```
function name(argument1, argument2,..., argument n)
{
   code
}
```

The parameters are passed by value.

4.6. Objects

In JavaScript, an object is a data structure that contains both variables (object properties) and functions for handling the object (methods). The object-oriented programming model used by Javascript is a lot simpler than that of Java or C++. JavaScript does not distinguish between objects and object instances.

The mechanism for accessing the properties or methods of an object is as follows:

```
object.property
value=object.method(parameter1, parameter2, ...)
```

4.6.1. Defining objects in JavaScript

To define an object in JavaScript, we must first define a special function whose purpose is to build the object. We need to assign the same name to this function, called constructor, as we did to the Object.

```
function MyObject(attr1, attr2)
{
   this.attr1=attr1;
   this.attr2=attr2;
}
```

From now on we can create objects of the type `MyObject`

```
object=new MyObject(....)
```

```
object.attr1=a;
```

To add methods to an object we must first define these methods as a normal function:

```
function Method1(attr1, attr2)
{
   //code
   // we have the object in this
}
```

To assign this method to an object method, type:

```
object.method1=Method1;
```

From now on, we can enter the following:

```
object.method1(....);
```

4.6.2. Inheritance

In object-oriented programming, inheritance allows us to create new objects with the methods and properties of objects that are called parents. This allows us to create derived objects, thus moving from generic implementations to increasingly specific implementations.

The syntax for creating an object derived from another, such as a `ChildObject` derived from a `ParentObject` that had two arguments (`arg1` and `arg2`) will be:

```
function ChildObject(arg1, arg2, arg3)
{
   this.base=ParentObject;
   this.base(arg1,arg2);
}
```

At this point, we can obtain access through a `ChildObject` to both the methods and properties of the child and parent objects.

4.6.3. Predefined objects

The existing JavaScript implementations incorporate a series of predefined objects:

- **Arrays** Vectors.
- **Date** For storing and handling dates.
- **Math** Mathematical methods and constants.

- **Number** Some constants.
- **String** String handling.
- **RegExp** Regular expression handling.
- **Boolean** Boolean values.
- **Function** Functions.

4.7. Events

One of the most important aspects of JavaScript is its browser interaction. For this, it incorporates a series of events triggered just as the user carries out an action on the web page.

Event	Description
onLoad	Page loading finishes. Available in: <BODY>
onUnLoad	A page is left. Available in: <BODY>
onMouseOver	The mouse is hovered over. Available in: <A>, <AREA>,..
onMouseOut	The mouse stops hovering over an element.
onSubmit	A form is sent. Available in: <FORM>
onClick	An element is clicked. Available in: <INPUT>
onBlur	The cursor is lost. Available in: <INPUT>, <TEXTAREA>
onChange	Content is changed. Available in: <INPUT>, <TEXTAREA>
onFocus	The cursor is found. Available in: <INPUT>, <TEXTAREA>
onSelect	Text is selected. Available in: <INPUT TYPE="text">, <TEXTAREA>

There are two mechanisms for indicating the function to handle an event:

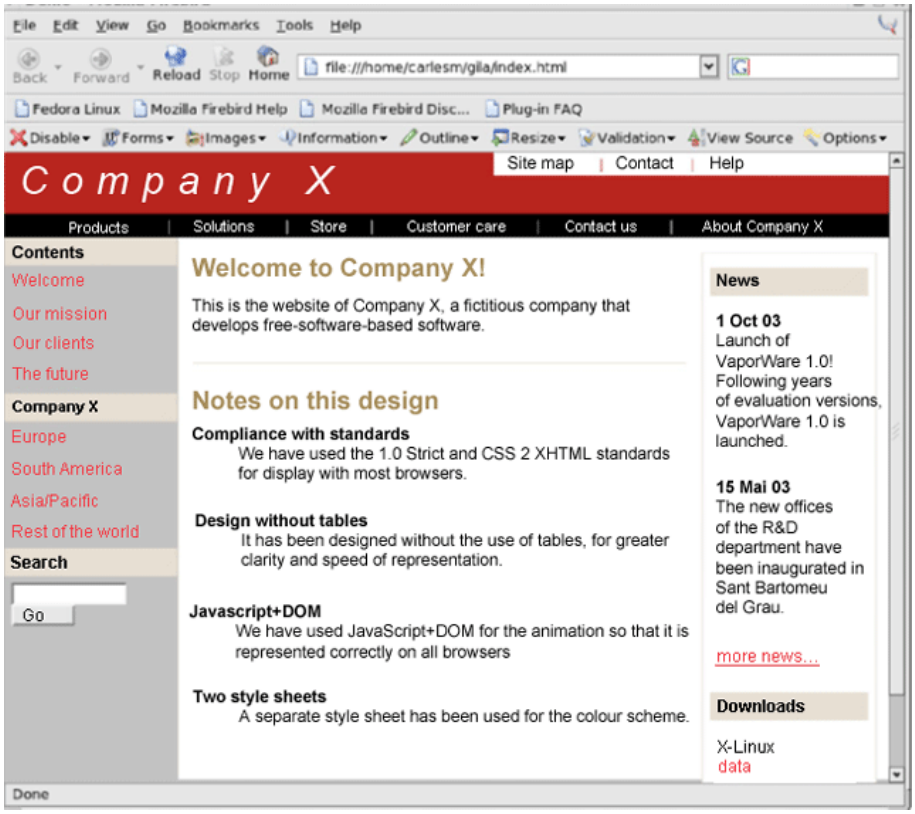
```
<HTML>
<HEAD>
   <SCRIPT LANGUAGE="Javascript">
      function Alarm() {
         alert("Hello World");
      }
   </SCRIPT>
</HEAD>
<BODY onLoad="Greeting()">
...
</BODY>
</HTML>
```

```
<HTML>
<HEAD>
   <SCRIPT LANGUAGE="Javascript">
```

```
      function Greeting() {
         alert("Hello World");
         }
      window.onload = Greeting;
   </SCRIPT>
</HEAD>
<BODY>
...
</BODY>
</HTML>
```

5. Practical: creating a complex web page using the techniques described.

We will now create a web page using all of the techniques seen up to this point. The result will be a page like this:

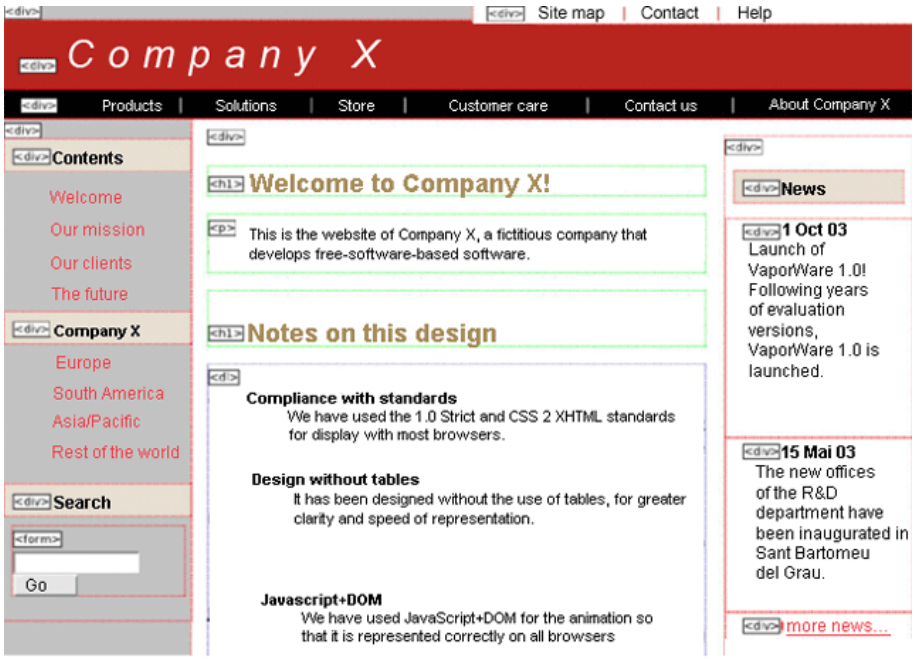


To build this page, we are not going to use any tables or frames). We are only going to use separators like `DIV`, `P`, etc. and CSS positioning.

We will also incorporate an animation effect in the title of the business, for which we will not use any components that are not DHTML. This will also allow the animation to work properly in Mozilla/Netscape browsers, such as Opera, and in Internet Explorer, etc.

Another of the principles for the design of this page will be that we will use two different files to maintain the style sheets, one containing the colours and the other containing the styles per se. This will make it easier to make any changes in style, etc.

We will now show the same page as before, indicating the type of element in each of the blocks that make up the page:



This is the code for the style sheet indicating the format, which, in our example, is called `style.css`

```
/* ###### Body ###### */

Body {
   font-family: verdana, tahoma, helvetica, arial, sans-serif;
   font-size: 94%;
   margin: 0;
}

h1, h2, h3 {
   font-family: arial, verdana, tahoma, sans-serif;
}
h1 {
   font-size: 164%;
   font-weight: bold;
   font-style: italic;
   padding-top: lem;
   border-top-style: solid;
   border-top-width: 1px;
} P {
   padding-bottom: lex;
} img {
   border: none;
}

code {
font-family: "lucida console", monospace;
font-size: 95%; }
```

```
dt {
   font-weight: bold;
}
dd {
   padding-bottom: 1.5em;
}
#textBody {
   text-align: justify;
   line-height: 1.5em;
   margin-left: 12em;
   padding: 0.5ex 14em 1em 1em;
   border-left-style: solid;
   border-left-width: 1px;
}

#textBody a {
   /* colours.css  */
}
#textBody a:hover {
   text-decoration: none;
}

/* ###### Header ###### */

#header{
   height: 4em;
   padding: 0.25em 2.5mm 0 4mm;
}

.headerTitle {
   font-size: 252%;
   text-decoration: none;
   font-weight: bold;
   font-style: italic;
   line-height: 1.5em;
   }

.headerTitle span {
   font-weight: normal;
}

.headerLinks {
   font-size: 87%;
   padding: 0.5ex lOem 0.5ex 1em;
   position: absolute; right: 0; top: 0;
}
```

```
.headerLinks * {
   text-decoration: none;
   padding: 0 2ex 0 lex;
}

.headerLinks a:hover {
   text-decoration: underline;
}

.menuBar {
   text-align: center;
   padding: 0.5ex 0;
}

.menuBar * {
   text-decoration: none;
   font-weight: bold;
   padding: 0 2ex 0 lex;
}

.menuBar a:hover {
   /* colours.css */
}

/* ###### Left ###### */
.leftBar {
   font-size: 95%;
   width: 12.65em;
   float: left;
   clear: left;
}

.leftBar a, .leftBar span {
   text-decoration: none;
   font-weight: bold;
   line-height: 2em;
   padding: 0.75ex lex; display: block;
}
[class ="leftBar"] a, [class ="leftBar"] span {
 line-height: 1.5em;
}

.leftBar a:hover {
   /* colors.css */
}

.leftBar .leftBarTitle {
```

```
   font-weight: bold;
   padding: 0.75ex lex;
}

.leftBar .textBar {
   font-weight: normal;
   padding: lex 0.75ex lex lex;
}

.leftBar .thePage{
/* colors.css */
}

/* ###### Right ###### */

.rightBar {
   font-size: 95%;
   width: 12.65em;
   margin: 2ex 0.8ex 0 0;
   float: right;
   clear: right;
   border-style: solid;
   border-width: 1px;
}
[class ="rightBar"] {
   margin-right: 1.5ex;
}

.rightBar a {
   font-weight: bold;
}

.rightBar a:hover {
   text-decoration: none;
}

.rightBar .leftBarTitle {
   font-weight: bold;
   margin: 1em 1ex;
   padding: 0.75ex 1ex;
}

.rightBar .textBar {
   font-weight: normal;
   line-height: 1.5em;
   padding: 0 3ex 1em 2ex;
```

```
}
```

We will now show `colors.css`:

```
/* ###### Text colours ###### */

#textBody a, .rightBar a
{ color: #ff2020; }

.rightBar a
{ color: #ff2020; }

h1, .rightBar span
{ color:  #a68c53; }

.headerTitle, .headerLinks *, .leftBar
.leftBarTitle, .leftBar .thePage,
.rightBar
.leftBarTitle
{ color: black; }

.menuBar a:hover, .leftBar a:hover
{ color: black; }

.leftBar a:hover
{ color: black; }

.headerTitle span, .menuBar, .menuBar *
{ color: white; }

.headerLinks
{ color: #b82619; }

/* ###### Background colours ###### */

body
{ background-color: #c3c2c2; }

#textBody, .headerLinks, .menuBar a:hover,
.rightBar
{ background-color: white;  }

#header
{ background-color: #b82619; }

.menuBar
{ background-color: black; }
```

```
.leftBar .leftBarTitle, .rightBar
.leftBarTitle
{ background-color: #e6dfcf; }

/* ###### Border colours ###### */

h1, #textBody, .rightBar
{ border-color:  #e6dfcf; }
```

We will now reveal the code for the website of *Company X*, our fictitious company:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
   <head>
      <meta http-equiv="content-type" content="application/xhtml+xml; charset=iso8859-1" />
      <meta name="author" content="haran" />
      <meta name="generator" content="Windows Notepad" />
      <link rel="stylesheet" type="text/css" href="style.css" />
      <link rel="stylesheet" type="text/css" href="colours.css"/>

      <title>Demo</title>
   <style type="text/css"> .textoexp {
      font-family: Arial;
      color: black;
      font-size: 30pt;
      text-align: left;
      letter-spacing: -20px;
   } </style>
   <script type="text/javascript"> function
   expandText(start, end, step, speed){
      if (start < end){
      document.getElementById("textoexpand").style.letterSpacing = start+"px";
         start=start+step;
         setTimeout("expandText("+start+","+end+","+step+","+speed+")",speed);
      }
   }
   </script>
   </head>
   <body onLoad="expandText(-20,30,1,5);">
   <div id="top"></div>

   <!-- ###### Header ###### -->
   <div id="header">
```

```
      <a href="./index.html" class="headerTitle" title="Homepage">
      <span class="textexp" id="textexpand">Company X</span></a>
      <div class="headerLinks">
         <a href="./index.html">Site Map</a>
         <a href="./index.html">Contact</a>
         <a href="./index.html">Help</a>
      </div>
   </div>

   <div class="menuBar">
      <a href="./index.html">Products</a>
      <a href="./index.html">Solutions</a>
      <a href="./index.html">Store</a>
      <a href="./index.html">Customer car</a>
      <a href="./index.html">Contact us</a>
      <a href="./index.html">About Company X</a>
   </div>

   <!-- ###### Left ###### -->

   <div class="leftBar">
      <div class="leftBarTitle">Content</div>
      <a href="#wel">Welcome</a>
      <a href="#mission">Our mission</a>
      <a href="#achievements">Our clients</a>
      <a href="#future">The future</a>
      <div class="leftBarTitle">Company X</div>
      <a href="index.html">Europe</a>
      <a href="index.html">South America</a>
      <a href="index.html">Asia/Pacific</a>
      <a href="index.html">Rest of the world</a>
      <div class="leftBarTitle">Search</div>
      <span class="textBar"><form method="GET">
      <Input type=text size=13 name="text"><input type=submit
      name="Go" value="Go"></form></span>
   </div>

   <!-- ###### Right ###### -->

   <div class="rightBar">
      <div class="leftBarTitle">News</div>
      <div class="textBar"><strong>1 Oct 03</strong><br />
       Launch of VaporWare 1.0! Following years of evaluation versions,
       VaporWare 1.0 is launched.
      </div>
   </div>
   <div class="textBar"><strong>15 May 03</strong><br />
```

```
     The new offices of the R&amp;amp;D department have been inaugurated
     in Sant Bartomeu del Grau. </div>
   <div class="textBar"><a href="./index.html">more news...</a></div>
   <div class="leftBarTitle">Downloads</div>
   <div class="textBar"><strong>X-Linux</strong><br/>
      <a href="./index.html">data  </a>  ;   <a href="./index.html">download</a></div>
   <div class="textBar"><strong>VaporWare</strong><br  />
      <a href="./index.html">data  </a>  ;   <a href="./index.html">download</a></div>
   </div>

   <!-- ###### Text ###### -->
   <div id="textBody">
      <h1 id="Welcome"
      style="border-top: none;
      padding-top: 0;">Welcome to Company X!</h1>
      <p>This is the website of Company X, a fictitious company
      that develops software based on free software.
      <h1 id="Notes">Notes on this design</h1>
      <dl>
         <dt>Compliance with standards</dt>
         <dd>We have used the 1.0 Strict and CSS 2 XHTML standards
         for display with most browsers.</dd>
         <dt>Design without tables</dt>
         <dd>It has been designed without the use of tables, for
         greater clarity and speed of rendering.</dd>
         <dt>Javascript+DOM</dt>
         <dd>We have used JavaScript+DOM for the animation so that
         it is rendered correctly on all browsers.</dd>
         <dt>Two style sheets</dt> <dd>A separate style sheet has
         been used for the colour scheme.</dd>
      </dl>
   </div>
</body>
</html>
```

Bibliography

Musciano, Chuck; Kennedy, Bill (2000). *HTML & XHTML: The Definitive Guide. 4th Edition*. O'Reilly.

Meyer, Eric A. (2000). *Cascading Style Sheets: The Definitive Guide*. O'Reilly.

Flanagan, David (2001). *JavaScript: The Definitive Guide*. O'Reilly.

Goodman, Danny (2002). *Dynamic HTML: The Definitive Reference*. O'Reilly.

Text structured format: XML

Carles Mateu

PID_00148402

Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Introduction to XML

XML is the abbreviation of Extensible Markup Language. It is a World Wide Web Consortium recommendation (`http://www.w3.org` is a basic reference for XML), whose original aim was to rise to the challenge of the electronic publication of documents on a large scale. XML is becoming increasingly important in the exchange of a wide variety of information on the Web and in other contexts.

XML derives from a markup language called SGML (an ISO standard, specifically ISO-8879) and it is a subset of SGML whose aim is to be served, received and processed on the Web like HTML. XML was designed for simplicity of implementation and interoperability with SGML and HTML, and for use in the design of data-based applications.

XML was developed in 1996 by a group under the auspices of W3C, originally known as the SGML working group, with the following goals, as set down in the standard definition document:

1) XML shall be straightforwardly usable over the Internet.
2) XML shall support a wide variety of applications.
3) XML shall be compatible with SGML.
4) It shall be easy to write programs which process XML documents.
5) The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6) XML documents should be human-legible and reasonably clear.
7) The XML design should be prepared quickly.
8) The design of XML shall be formal and concise.
9) XML documents shall be easy to create.
10) Brevity in XML markup is of minimal importance.

Another area of the recommendation refers to other related standards that, together with the definition of XML, are absolutely essential for understanding XML:

This specification, together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 1766 for language identification tags, ISO 639 for language name codes, and ISO 3166 for country name codes), provides all the information necessary to understand XML Version 1.0 and construct computer programs to process it.

You may wonder why W3C saw the need to develop a new language for the Web when we already had HTML. By 1996, HTML had already revealed some of its most serious deficiencies:

- HTML was optimised to be easy to learn, not to be easy to process:
  - A single set of tags (regardless of applications).
  - Predefined semantics for each tag.
  - Predefined data structures.

- HTML sacrifices power for ease of use.
- HTML is fine for straightforward applications but not so good for complex applications:
  - Complex data sets.
  - Data that needs to be handled in different ways.
  - Data for controlling programs.
  - No possibility of formal validation.

In the light of this, the W3C developed a new language (XML) providing:

Extensibility: new tags and attributes can be defined as needed.
Structure: any type of hierarchically organised data can be modelled.
Validity: data can be automatically validated (structurally).
Media independence: the same content can be published on a range of media.

XML can be characterised as follows:

XML is a simplified version of SGML that is very easy to implement. It is a metalanguage, rather than a language, designed for defining an unlimited number of languages for specific purposes which can be processed using the same tools regardless of the purpose for which they were built.

As this definition indicates, XML is not a language, it is a metalanguage that allows us to define a multitude of languages for specific purposes. The following chapters will explain how to define these languages and how to

define rules for the structural validation of the languages. These definitions and even the definition of *programs* to translate and transform XML files are in XML, which gives us a clue as to the power of XML. Many languages have been defined in XML; the configuration files of some of the Web's most widely-used programs (Tomcat, Roxen, etc.) are defined with XML and many data files and documents, among others, also use it for their definition. The most well-known languages defined with XML include:

- **SVG** (Scalable Vector Graphics)
- **DocBook XML** (Docbook-XML)
- **XMI** (XML Metadata Interface Format)
- **WML** (WAP Markup Language)
- **MathML** (Mathematical Markup Language)
- **XHTML** (XML HyperText Markup Language)

Here is an example of an XML document that we have defined:

```
<?xml version="l.0" encoding="iso-8859-l"?>
   <library>
      <book language="English">
         <title>The Hobbit</title>
         <author>J. R. R. Tolkien</author>
         <publisher>Allen and Unwin</publisher>
      </book>
      <book language="Spanish">
         <title>El Quijote</title>
         <author>Miguel de Cervantes</author>
         <publisher>Alfaguara</publisher>
      </book>
   </library>
```

This sample document contains some of the features of XML already mentioned, such as a hierarchical structure, legibility, designed for a specific use (the storage of books in our library). The example also demonstrates the differences between XML and HTML that justified the introduction of XML. We will now write a similar document in HTML:

```
<HTML>
   <HEAD>
      <title>Library</TITLE>
   </HEAD>
   <BODY>
      <Hl>The Hobbit</Hl>
      <P><B>Author:</B>J. R. R. Tolkien</P>
      <B>Publisher</B>Allen and Unwin</P>

      <H1>El Quijote</Hl>
```

```
        <P><B>Author:</B>Miguel de Cervantes</P>
        <P><B>Publisher</B>Alfaguara</P>
    </BODY>
</HTML>
```

As we can see, HTML offers us ways of representing the display format (<B>, <I> etc. represent the format: bold, italic etc.), but these do not give us any information on the semantics of the data they contain. In XML, we have tags that tell us about the semantics of the data: <author> indicates that the data contained corresponds to the author etc. No information is given on the display format; we are not told how the name of the author will be visualised on screen (bold, italics, etc). XML has other tools for specifying this and for changing the format of representation according to where it is to be displayed (to adapt it to a specific web browser, mobile telephone etc).

2. XML

An XML object (or XML document) is defined as a document formed by tags and values that meets the XML specification and is well formed.

XML specification

The XML specification is available from the W3C website: http://www.w3c.org

We will begin our study of XML with a sample XML document to see how it is made. To do so, we will design an XML format to represent kitchen recipes. We will save our recipes in this XML format (generally called XML application), which we call RecipeXML.

```
<?xml version="1.0"?>
   <Recipe>
      <Name>Spanish omelette</Name>
      <Description>
      The traditional, typical Spanish omelette,
      just like our mothers make it.
      </Description>
      <Ingredients>
         <Ingredient>
            <Quantity unit="piece">3</Quantity>
            <Item>Potatoes</Item>
         </Ingredient>
         <Ingredient>
            <Quantity unit="piece">2</Quantity>
            <item>Eggs</Item> </Ingredient>
         <Ingredient>
            <Quantity unit="litre">0.1</Quantity>
            <Item>Oil</Item>
         </Ingredient>
      </Ingredients>
      <Instructions>
         <Step>
         Peel and slice the potatoes
         </Step>
         <Step>
         Add some oil to a frying pan
         </Step>
         <!-- And continue in this way... -->
      </Instructions>
   </Recipe>
```

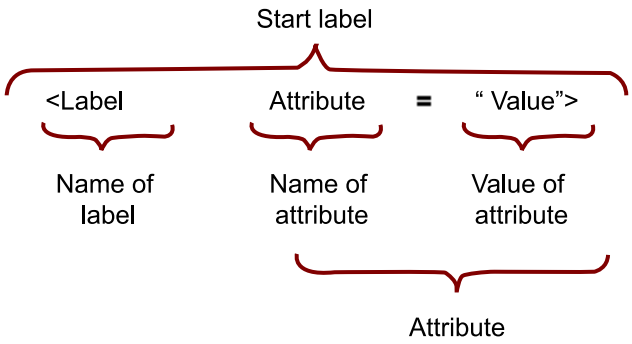
Recipe written in RecipeXML

This document is an XML document. All *well-formed* XML documents must begin with a line like this:

```
<?xml version="1.0"?>
```

, telling us the version of the XML specification used and that it is an XML document. It must also be made up exclusively of XML tags organised hierarchically. We can quickly see that XML does not store how the information should be represented or displayed; instead, it stores the semantics of this. Our document shows how this information is organised:

Recipes are *made up of a list of ingredients and instructions. The list of ingredients is a series of ingredients each with its name, quantity etc.*



As in our recipe, all XML tags follow the format shown.

2.1. Well-formed document

The well-formed concept is taken from mathematics, in which it is plausible to write an expression containing mathematical symbols such as:

$$\mathbf{1)}1(--5(+=)4 < 3$$

which, although formed by mathematical symbols, means nothing because it does not follow the rules and conventions of writing mathematical expressions. This mathematical expression is not *well formed*.

In XML, a well-formed document must follow these rules:

All tags closed: in HTML, we can *be* quite lax about syntactical rules, leaving tags (like <B>, for example) open for the whole of a document or indiscriminately using tags like <P> without closing them with their corresponding </P>. XML does not permit such laxness. All open tags

must have their corresponding closing tags. This is because tags in XML represent hierarchical information showing how the different elements relate to one another. If we do not close the tags, we create ambiguities in this representation that will inhibit automatic processing.

Tags cannot overlap: a tag opened inside another tag must be closed before the tag that contains it is closed. The example:

```
<Book>Platero y Yo<Author>J. R.Jiménez</Book></Author>
```

is not well formed because **Author** is not closed inside Book, where it should be. The correct sentence should read:

```
<Book>Platero y Yo<Author>J. R. Jiménez</Author></Book>
```

In other words, the structure of the document must be strictly hierarchical.

The values of attributes must appear inside quotation marks: unlike HTML, where we can indicate attributes without quotation marks, for example:

```
<IMAGE SRC=img.jpg SIZE=10>
```

in XML, all attributes must be enclosed by quotation marks. The above attributes would therefore be written as:

```
<IMAGE SRC="img.jpg" SIZE="10">.
```

The characters <, > and " are always represented by character entities: to represent these characters (in the text, rather than as tag marks), we must always use special character entities: **<**, **>** and **"**. These characters are specific to XML.

2.2. Well-formed is equivalent to analysable

The importance of whether or not a document is well formed in XML lies in the fact that a well-formed document can be subject to syntactic analysis or parsing (i.e. it can be automatically processed). There are many parsers (analysers) in many programming languages enabling us to work with XML documents. XML parsers can detect structural errors in XML documents (i.e. whether they are well formed) and report them to the program. This feature is very important for programmers because it releases them from the task of having to detect errors by assigning it to a program (the parser), which does it automatically.

Some parsers go beyond simply detecting whether the document is well formed and can even detect whether it is *valid*, which means that the structure, position and number of tags is correct and makes sense. Let's take the following excerpt from our recipe document:

```
<Ingredient>
   <Quantity unit="piece">3</Quantity>
   <Quantity unit="litre">4</Quantity>
   <Item>Potatoes</Item>
</Ingredient>
```

This XML is well formed and meets all the criteria for this but it does not make sense. What does it mean? We have potatoes in our recipe, but how many do we need? In this case, the problem is that we have a well-formed XML document but it is useless because it makes no sense. We need some sort of way to ensure that our document makes sense. In this case, we need to specify that each ingredient will only have a quantity type tag, that this will have an optional attribute and that it will not contain nested tags. To do this, XML has two document structure specification languages, **XML Schema** and **DTD**, which we will see later.

2.3. Namespaces

XML is a standard designed for the sharing of information. What happens when we take information from two different sources and combine it in XML to send it to somebody else? We may have some sort of conflict arising from coincidences in the names of tags.

Imagine this scenario: an Internet provider saves all of its data in XML. The sales division stores the home address of clients in a field called `<address>`. The client helpdesk uses `<address>` to store the client's e-mail address and, lastly, the network control centre uses `<address>` to store the IP address of the client's computer. If we combine the information from the three departments of the company in a single file, we could end up with:

```
<client>
   ...
   <address>Royal Street</address>
   ...
   <address>sales@client.com</address>
   ...
   <address>192.168.168.192</address>
   ...
</client>
```

Clearly, we would have a problem in this case because we would be unable to work out the meaning of `<address>` in each case.

Therefore, in 1999, the W3C defined an XML extension called namespaces, which can resolve many conflicts and ambiguities of this nature.

2.3.1. Using namespaces

Namespaces are a prefix added to XML tags to indicate the context of the tag in question. In the above example, we could define:

`<network:address>`: for use by the network control centre.
`<help:address>`: for use by the client helpdesk.
`<sales:address>`: for use by the sales department.

Our combined document would thus look like this:

```
<client>
   ...
   <sales:address>Royal Street</sales:address>
   ...
   <help:address>sales@client.com</help:address>
   ...
   <network:address>192.168.168.192</network:address>
   ...
</client>
```

In order to use a namespace in a document we must first declare it. This can be done in the root element of the document as shown:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<clientportfolio
   xmlns:sales="http://www.company.com/sales"
   xmlns:help="http://www.company.com/help"
   xmlns:network="http://www.company.com/network">
   <client>
      ...
      <sales:address>Royal Street</sales:address>
      ...
      <help:address>sales@client.com</help:address>
      ...
      <network:address>192.168.168.192</network:address>
      ...
   </client>
</clientportfolio>
```

The definition has `xmlns` attributes(XML namespace) where we indicate the prefix that we will use for the namespace and a URI (Uniform Resource Identifier) that will act as the unique namespace identifier.

3. Validation: DTD and XML Schema

As we have seen, XML allows us to check a document's form automatically, but without additional information it is impossible to check the validity of this form based on the document. As a result, W3C has developed XML standards to validate documents with a formal specification of how they should be. These standards are called DTD and XSchema.

DTD is an old standard derived from SGML that has some major weaknesses, the biggest being that it is not written in XML. XSchema, on the other hand, is a relatively modern, very powerful and extensible standard that is written entirely in XML.

3.1. DTD

DTD (**Document Type Definition**) is a standard allowing us to define a *grammar* that our XML documents must follow in order to be considered valid. A DTD definition for n XML documents specifies which elements can exist in an XML document, the attributes that these can have and which elements can or must be contained within other elements and their order.

XML parsers that can validate documents with DTDs read these documents and the associated DTD. If the XML document does not meet the requirements set down in the DTD, the parsers report the error and do not validate the document.

With DTDs, we define our XML dialect (remember that we define which tags we will use in our documents, the meaning we give to them etc). This ability to define a specific XML dialect is what gives the latter its extensible character. Although the DTD standard should have been replaced by XML Schema, it is still commonplace, easier to use and more compact than XML Schema. In addition, most users do not need the improvements introduced by XML Schema. A variety of XML dialects have been defined with DTD that are widely used on the Internet, such as RDF for semantic web, MathML for mathematical documents, XML/EDI for electronic data interchange in business, VoiceXML for applications operated by voice or which use voice, WML to represent documents for browsers on mobile devices such as telephones, etc.

Here is a possible DTD for our sample recipe that will define the way in which our recipes should be written in RecipeXML:

```
<!- Sample DTD for RecipeXML ->;
<!ELEMENT Recipe (Name, Description?,
   Ingredients?, Instructions?)>;
```

```
<!ELEMENT Name (#PCDATA)>;
<!ELEMENT Description (#PCDATA)>;
<!ELEMENT Ingredients (Ingredient*)>;
<!ELEMENT Ingredient (Quantity, Item)>;
<!ELEMENT Quantity (#PCDATA)>;
<!ATTLIST Quantity unit CDATA #REQUIRED>;
<!ELEMENT Item (#PCDATA)>;
<!ATTLIST Optional item CDATA "0"
   vegetarian CDATA "yes">;
<!ELEMENT Instructions (Step)+>;
<!ELEMENT Step (#PCDATA)>;
```

We can work out the validity rules from this DTD document and produce a more legible description:

- A recipe consists of a name (compulsory), description (optional), ingredients (optional) and instructions (optional).
- The name and description can contain alphanumeric characters (PCDATA means Parsed Character Data).
- The ingredients are a list of ingredient elements.
- An ingredient consists of an item and the quantity.
- The quantity is an alphanumeric value and the tag has an attribute, the unit, which describes the unit of measurement being used.
- A recipe item consists of the name (an alphanumeric value) and can have two attributes: optional (whether the ingredient is compulsory or not) and vegetarian (whether the ingredient is suitable for vegetarians).
- The instructions for preparation are a list of steps.
- A step consists of an alphanumeric text describing the step.

We will now look at DTD syntax in order to define XML dialects.

3.1.1. Syntactic conventions of DTD

As we have seen, the syntax of DTD is not very clear at first sight; however, it is not too complicated. The first step towards understanding it is to know the definitions and uses of the symbols used, which can be seen in Table

Symbol	Description
()	Parentheses are used to group subtags <!ELEMENT Ingredient (Quantity,Item)>
,	Exact order of the elements (Name, Description?, Ingredients?, Instructions?)
|	Just one of the elements indicated (Boil | Fry)
	If we do not indicate anything, the elements only appear once (Quantity, Item)
+	Once or more Step+
?	Optional element Instructions?
*	Zero times or more Ingredient*
#PCDATA	Parsed Character Data <!ELEMENT Item (#PCDATA)>

Syntactic elements of DTD

3.1.2. ELEMENT element

The DTD elements called ELEMENT define a tag in our XML dialect. For example:

```
<!ELEMENT Recipe (Name, Description?, Ingredients?, Instructions?)>
```

defines the Recipe tag, specifying that it contains the subtags: Name, Description, Ingredients and Instructions, and that the last three are optional (indicated by the ? symbol).

The definition of ELEMENT is as follows:

```
<!ELEMENT name category>
<!ELEMENT name (content)>
```

Empty elements

Empty elements are declared with the EMPTY category.

```
<!ELEMENT name EMPTY>
```

In XML, this name element would be used like this:

```
<name />
```

Character-only elements

Elements that only contain alphanumeric data are declared using #PCDATA inside parentheses.

```
<!ELEMENT name (#PCDATA)>
```

Elements with any content

The elements we declare using ANY as an indicator of content can contain any combination of parseable data:

```
<!ELEMENT name ANY>
```

Elements with subelements (sequences)

Elements with one or more child elements are defined with the name of the child elements inside parentheses:

```
<!ELEMENT name (child1)>
<!ELEMENT name (child1, child2, ........)>
```

For example:

```
<!ELEMENT car (make, number plate, colour)>
```

Children declared in a sequence of elements separated by commas must appear in the same order in the document. Child elements must also be declared in the DTD document. These child elements can also have children themselves.

The full declaration of car is therefore:

```
<!ELEMENT car (make, number plate, colour)>
<!ELEMENT make (#PCDATA)>
<!ELEMENT number plate (#PCDATA))>
<!ELEMENT colour (#PCDATA)>
```

Cardinality of element occurrences

The following declaration indicates that the child element can only occur inside the parent element:

```
<!ELEMENT name (child)>
```

If we want the child element to appear more than once and at least once:

```
<!ELEMENT name (child+)>
```

If we want it to appear any number of times (including the possibility that it does not appear at all):

```
<!ELEMENT name (child*)>
```

If we only want to allow it to appear once but do not want this to be compulsory:

```
<!ELEMENT name (child?)>
```

Mixed content elements.

We can declare elements containing other child elements and/or alphanumeric data.

```
<!ELEMENT name (#PCDATA child child2)*>
```

3.1.3. ATTLIST element

As we have seen, elements can have attributes in XML. Obviously, DTD has a mechanism for indicating which attributes an ELEMENT can have, the type, whether they are compulsory, etc. We use the ATTLIST element for this, whose syntax is:

```
<!ATTLIST element attribute type-attribute value-default&gt;
```

An example of its use would be:

```
<!ATTLIST payment method CDATA "cash on delivery" >
```

And its use in XML:

```
<payment method="cash on delivery"  />
```

The attribute type must be one taken from the following list:

Value	Description
CDATA	The value is character data
(V1/V2/...)	The value must be one from an enumerated list
ID	The value is a unique ID
IDREF	The value is the ID of another element
IDREFS	The value is a list of other IDs
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is a predefined XML value

The default value can be one of the following:

Value	Description
value	The default value of the attribute
#REQUIRED	The value of the attribute must appear in the element
#IMPLIED	The attribute does not need to be included value
#FIXED value	The value of the attribute is fixed

Default value

This example:

```
<!ELEMENT payment EMPTY><!ATTLIST payment method CDATA "cash on delivery" >
```

the following XML is considered valid:

```
<payment />
```

In this case, since we have not specified a value for method, it will contain the default value, cash on delivery.

Syntax of #IMPLIED

This example:

```
<!ELEMENT payment EMPTY>
<!ATTLIST payment method CDATA #IMPLIED >
```

will validate the following XML correctly:

```
<payment method="card" />
<payment  />
```

Thus, we use #IMPLIED when we do not want to force the user to use an attribute but where we cannot enter default values.

Syntax of #REQUIRED

This example:

```
<!ELEMENT payment EMPTY>
<!ATTLIST payment method CDATA #REQUIRED >
```

will validate the following XML correctly:

```
<payment method="card" />
```

but it will not validate:

```
<payment />
```

We use #REQUIRED when we cannot supply a default value but we want the attribute to appear and have a value assigned to it.

3.1.4. Linking documents to a DTD

There are two ways to link an XML document to a DTD: either include the DTD in the XML document or use an external reference to the DTD.

The first option is the easiest but has the most disadvantages because it increases the size of the XML documents and complicates their maintenance, since a change in the DTD will require revising every document that included it.

The format of an XML document in which it is included would look like this:

```
<?xml version="1.0"?>
<!DOCTYPE Recipe [
   <!ELEMENT Recipe (Name, Description?,
      Ingredients?, Instructions?)>
   <!ELEMENT Name (#PCDATA)>
   <!ELEMENT Description (#PCDATA)>
   <!ELEMENT Ingredients (Ingredient)*>
   <!ELEMENT Ingredient (Quantity, Item)>
   <!ELEMENT Quantity (#PCDATA)>
   <!ATTLIST Quantity unit CDATA #REQUIRED>
   <!ELEMENT Item (#PCDATA)>
   <!ATTLIST Optional item CDATA "0"
      vegetarian CDATA "yes">
   <!ELEMENT Instructions (Step)+>
```

```
   <!ELEMENT Step (#PCDATA)>
]>
<Recipe>
   <Name>Spanish omelette</NAME>
   <Description>
   The traditional, typical Spanish omelette,
   just like our mothers make it.
   </Description>
   <Ingredients>
      <Ingredient>
         <Quantity unit="piece">3</Quantity>
         <Item>Potatoes</Item>
      </Ingredient>
      <Ingredient>
         <Quantity unit="piece">2</Quantity>
         <Item>Eggs</Item>
      </Ingredient>
      <Ingredient>
         <Quantity unit="litre">0.1</Quantity>
         <Item>Oil</Item>
      </Ingredient>
   </Ingredients>
   <Instructions>
      <Step> Peel and slice the potatoes </Step>
      <Step> Add some oil to a frying pan </Step>
      <!-- ... ... -->
   </Instructions>
</Recipe>
```

We can reference an external DTD to the XML document using two possible types of reference: public or private.

This is an example of a private reference:

```
<?xml version="1.0"?>
<!DOCTYPE Recipe SYSTEM "Recipe.dtd">
<Recipe>
...
```

While the following example uses a public external reference:

```
<?xml version="1.0"?>
<!DOCTYPE Recipe
   PUBLIC "-//W3C//DTD XHTML 1.0 STRICT/EN
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<Recipe>
```

```
...
```

3.2. XML Schema

XML Schema was launched in 2001 as part of W3C's efforts to redress the obvious failures of DTD:

- We do not have significant control over what is considered valid.
- We have no control over data types (integers etc).
- It is not defined as XML.
- The syntax can sometimes be complicated.

XML Schema has certain features making it far more powerful than DTD:

- It is defined in XML, which means that it is possible to validate XML Schema documents too.
- It enables control over data types (integers etc).
- It allows us to define new data types.
- It allows us to describe the content of documents.
- It is easy to validate correct data.
- It is easy to define data patterns (formats).

Despite its many advantages, DTD is still the most common mechanism for defining the structure of XML documents.

Extensibility

XML Schema is extensible because it allows definitions from a schema can be reused in another schema. It can also be used to define data types from the data types in the standard and other schemas, and allows a single document to use several schemas.

We will introduce XML Schema (also known as XSD, XML Schema Definition) by comparing it to the now-familiar DTD. We will do this using an XML document, our RecipeXML.

```
<?xml version="1.0"?>
   <Recipe>
      <Name>Spanish omelette</Name>
      <Description>
      The traditional, typical Spanish omelette,
      just like our mothers make it.
      </Description>
      <Ingredients>
         <Ingredient>
            <Quantity unit="piece">3</Quantity>
            <Item>Potatoes</Item>
```

```
         </Ingredient>
         <Ingredient>
            <Quantity unit="piece">2</Quantity>
            <item>Eggs</Item>
         </Ingredient>
         <Ingredient>
            <Quantity unit="litre">0.1</Quantity>
            <Item>Oil</Item>
         </Ingredient>
      </Ingredients>
      <Instructions>
         <Step> Peel and slice the potatoes </Step>
         <Step> Add some oil to a frying pan </Step>
         <!-- And continue in this way... -->
      </Instructions>
   </Recipe>
```

This time, instead of showing the DTD associated with it as we saw in the previous section, we will define an XSD for the document in RecipeXML.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">
   <xs:element name="Quantity">
      <xs:complexType>
         <xs:simpleContent>
            <xs:extension base="xs:decimal">
               <xs:attribute name="unit"
                  type="xs:string" use="required"/>
               </xs:extension>
            </xs:simpleContent>
         </xs:complexType>
      </xs:element>
      <xs:element name="Description" type="xs:string"/>
      <xs:element name="Ingredient">
         <xs:complexType>
            <xs:sequence>
               <xs:element ref="Quantity"/>
               <xs:element ref="Item"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="Ingredients">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="Ingredient"
               maxOccurs="unbounded"/>
```

```
         </xs:sequence>
      </xs:complexType>
      </xs:element>
      <xs:element name="Instructions">
         <xs:complexType>
            <xs:sequence>
               <xs:element ref="Step"
                  maxOccurs="unbounded"
                  minOccurs="l"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="Item" type="xs:string"  />
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="Step" type="xs:string"/>
      <xs:element name="Recipe">
         <xs:complexType>
            <xs:sequence>
               <xs:element ref="Name"/>
               <xs:element ref="Description"/>
               <xs:element ref="Ingredients"/>
               <xs:element ref="Instructions"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
</xs:schema>
```

3.2.1. The <schema> element

The <schema> element is the root element of all XSDs:

```
<?xml version="1.0"?>
<xs:schema>
... ...
</xs:schema>
```

This element can have some attributes and generally looks similar to this:

```
<?xml version="1.0"?>
   <xs:schema
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.company.com"
      xmlns="http://www.company.com"
         elementFormDefault="qualified">
      ... ...
   </xs:schema>
```

The following fragment:

`xmlns:xs="http://www.w3.org/2001/XMLSchema"`

indicates that the elements and data types used in our `schema`, `element` etc. come from the namespace defined in http://www.w3.org/-2001/XMLSchema. It also assigns a prefix to this namespace, in this case, `xs`.

This fragment `targetNamespace="http://www.company.com"` indicates that the elements defined by this schema (Ingredient, etc.) come from the "http://www.company.com" namespace.

This one: `xmlns="http://www.company.com"`, indicates that the default namespace is http://www.company.com, which means that we can assign tags without a prefix to this namespace.

This one: `elementFormDefault="qualified"` indicates that all of the elements used in the XML document declared in this schema must be qualified in the namespace.

3.2.2. Simple elements

A simple element is an XML element that can only contain text. It cannot contain other elements or attributes. Nonetheless, the text it contains can be of any type included in the XSD definition (Boolean, integers, string, etc.) or it can be a type defined by us. We can also add restrictions to limit the content or require the data it contains to follow a given pattern.

The syntax for defining a simple element is:

```
<xs:element name="name" type="type"/>
```

where *name* is the name of the element and *type* is the data type of the element. Some examples of declarations might be:

```
<xs:element name="Item" type="xs:string"  />
<xs:element name="Age" type="xs:integer"/>
<xs:element name="Date" type="xs:date"/>
```

The following are XML elements that meet the above restrictions:

```
<Item>Potatoes</Item>
<Age>34</Age>
<Date>1714-09-11</Date>
```

XSD offers the following data types:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Fixed and default values

Simple elements can have a default or fixed value. A default value is automatically assigned to the element when we do not specify a value. This example assigns Oil as the default value.

```
<xs:element name="Item" type="xs:string"
 default="Oil"/>
```

An element with a fixed value always has the same value and we cannot assign another one to it.

```
<xs:element name="Item" type="xs:string"
 fixed="Oils"/>
```

3.2.3. Attributes

Simple elements cannot have attributes. If an element has attributes it is considered a complex type. The attribute, however, is always declared as a simple type. We have the same basic data types for attributes as we do for elements.

The syntax for defining an attribute is

```
<xs:attribute name="name" type="type"/>
```

where *name* is the name of the attribute and *type* is the data type of the attribute. One XML element with RecipeXML attributes is:

```
<Quantity unit="piece">3</Quantity>
```

and the corresponding XSD for this definition is:

```
<xs:attribute name="unit" type="xs:string"
 use="required"/>
```

Fixed and default values

The declaration of attributes with fixed and default values uses the same schema as elements:

```
<xs:attribute name="unit" type="xs:string"
 default="grams"/>
<xs:attribute name="unit" type="xs:string"
 fixed="grams"/>
```

Optional and required attributes

Attributes are optional by default. Nonetheless, we can openly specify that an attribute is optional:

```
<xs:attribute name="unit" type="xs:string"
 use="optional"/>
```

To specify that it is required:

```
<xs:attribute name="unit" type="xs:string"
 use="required"/>
```

3.2.4. Content restrictions

With XSD, we can extend the content restrictions of XSD data types (integers, etc.) with restrictions designed by ourselves (facets). For example, in the following code, we specify an element, age, which must have a whole value between 1 and 120.

```
<xs:element name="age">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="120"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>
```

Restrictions on a set of values

With XSD, we can limit the content of an XML element so that it can only contain a value from a set of acceptable elements. To do this, we use the enumeration restriction). For example, we can define an element called wine by specifying the possible values:

```
<xs:element name="wine">
<xs:simpleType>
   <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="White"/>
      <xs:enumeration value="Rosé"/>
      <xs:enumeration value="Red"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>
```

The wine element is a simple data type with restrictions. Its acceptable values are: *Red*, *Rosé* and *White*. We could also have defined it like so:

```
<xs:element name="wine" type="wineType"/>
<xs:simpleType name="wineType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="White"/>
      <xs:enumeration value="Rosé"/>
      <xs:enumeration value="Red"/>
   </xs:restriction>
</xs:simpleType>
```

In this case, we could also use the `wineType` data type for other elements because it is not part of the definition of `wine`.

Restrictions on a series of values

We can use patterns to define elements containing a specific series of numbers or letters. For example:

```
<xs:element name="letter">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

The **letter** element is a simple type with a restriction whose only acceptable value is **ONE** of the lowercase letters.

For example, the following element:

```
<xs:element name="productcode">

<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
```

```
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

defines a **productcode** formed by precisely five digits from 0 to 9.

The following code defines a **letters** type:

```
<xs:element name="letters">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="([a-z])*"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>
```

that can take the value of any lowercase letter that appears zero or more times, i.e. it can have a null value. However, in:

```
<xs:element name="letters">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="([a-z][A-Z])+"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

, the element can contain lowercase letters but it must contain at least one letter.

Patterns allow the same type of definitions as restrictions on a set of elements. For example:

```
<xs:element name="gender">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="male female"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

defines a gender element whose value can be: *male* or *female*. We can also define more complex types using patterns such as:

```
<xs:element name="password">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[a-zA-ZO-9]{8}"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

which defines a **password** element made up of eight characters, either letters or numbers.

Restrictions on whitespace characters

XSD has a restriction for specifying how to handle whitespaces. This is called the whiteSpace restriction. For example:

```
<xs:element name="address">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

allows us to define an element called **address** where we tell the XML processor that we do not want it to remove any whitespaces. However, the following definition:

```
<xs:element name="address">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:whiteSpace value="replace"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

tells the XML processor that we want it to replace *whitespace* characters (tabs, line breaks, etc.) with spaces. The *collapse* option will also replace *whitespace* characters with spaces and reduce consecutive multiple spaces and characters at the start or end of a line to a single space.

Length restrictions

To restrict the length of a value in an element, the following constraints are available: *length*, *maxLength* and *minLength*. We will define an element called **password**:

```
<xs:element name="password">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:length value="8"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

to have a compulsory length of 8. We can also set it to have a variable length of 5 to 8:

```
<xs:element name="password">

<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
   <xs:maxLength value="8"/>
   </xs:restriction>
</xs:simpleType>

</xs:element>
```

The following table summarises the restrictions that can be applied to data types:

Restriction	Description
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of permitted decimal digits. This must be greater than or equal to zero.
length	Specifies the exact required size. It must be greater than or equal to zero.
maxExclusive	Specifies the upper limit for numerical values (the value must be less than this number).

Restriction	Description
maxInclusive	Specifies the upper limit for numerical values (the value must be less than or equal to this number).
maxLength	Specifies the maximum permitted size. This must be greater than or equal to zero.
minExclusive	Specifies the lower limit for numerical values (the value must be greater than this number).
minInclusive	Specifies the lower limit for numerical values (the value must be greater than or equal to this number).
minLength	Specifies the minimum required size. This must be greater than or equal to zero.
pattern	Specifies the pattern defining the exact string of permitted characters.
totalDigits	Specifies the exact number of permitted digits. It must be greater than zero.
whiteSpace	Specifies how whitespace characters (spaces, tabs, etc.) should be dealt with.

3.2.5. Complex elements in XSD

A complex element is an XML element that contains other elements and/or attributes. We can divide complex elements into four main classes:

- Empty elements
- Elements that only contain other elements
- Elements that contain text only
- Elements that contain other elements and text

All complex elements can also contain attributes.

The following are complex elements of each type:

An **empty** product element:

```
<product id="1345"/>
```

A **student** element that contains other elements:

```
<student>
<name>John</name>
<surname>Smith</surname>
</student>
```

An **accommodation** element that contains only text:

```
<accommodation type="hotel">
South Coast Inn
```

```
</accommodation>
```

An **expedition** element containing both text and elements:

```
<expedition destination="Fitz Roy">
We arrive at Chaltén in Patagonia on
<date>22.08.2003</date> ....
</expedition>
```

Defining complex elements

Let's look at the **student** element that contains other elements:

```
<student>
<name>John</name>
<surname>Smith</surname>
</student>
```

We can define this XML element in different ways:

1)We can openly declare the **student** element:

```
<xs:element name="student">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="surname" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Thus, only the **student** element may use the complex type defined. Note that the elements contained in **student** (**name** and **surname**) are contained in a `sequence` command, which means that they must appear in the same order in the element.

2)The **student** element may have a type attribute referring to the complex type to be used:

```
<xs:element name="student" type="personinfo"/>

   <xs:complexType name="personinfo">
      <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
   </xs:sequence>
```

```
</xs:complexType>
```

With this technique, multiple elements can refer to the same complex type.
Thus:

```
<xs:element name="student" type="personinfo"/>
<xs:element name="teacher" type="personinfo"/>
<xs:element name="staff" type="personinfo"/>

<xs:complexType name="personinfo">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

We can also use a complex type as a base for building more elaborate complex types:

```
<xs:element name="student" type="personinfo"/>

<xs:complexType name="person">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="personinfo">
   <xs:complexContent>
      <xs:extension base="person">
         <xs:sequence>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Empty elements

Take the following XML element:

```
<product id="1345"/>
```

It is an XML element that contains neither text nor XML elements. To define it, we need to define a type that only allows elements in its content but we must not declare any elements. So:

```
<xs:element name="product">
   <xs:complexType>
      <xs:complexContent>
         <xs:restriction base="xs:integer">
            <xs:attribute name="id" type="xs:positiveInteger"/>
         </xs:restriction>
      </xs:complexContent>
   </xs:complexType>
</xs:element>
```

Here, we define a complex type that only contains elements (with `complexContent`). This directive indicates that we want to derive the content from a complex type but do not enter content. We also have a restriction that adds a whole attribute. We can define the declaration more compactly:

```
<xs:element name="product">
   <xs:complexType>
      <xs:attribute name="id" type="xs:positiveInteger"/>
   </xs:complexType>
</xs:element>
```

or define it as a name type to use in the definition of other elements:

```
<xs:element name="product" type="producttype"/>
<xs:complexType name="producttype">
   <xs:attribute name="id" type="xs:positiveInteger"/>
</xs:complexType>
```

Defining complex types that contain elements only

Study the following element, which only contains other elements:

```
<student>
<name>John</name>
<surname>Smith</surname>
</student>
```

We can define this element in XSD as follows:

```
<xs:element name="student">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
```

```
         <xs:element name="surname" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

As before, we can define it as a name type for use in multiple elements:

```
<xs:element name="student" tipo="person">
   <xs:complexType name="person">
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="surname" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
```

Note the use of `sequence`, indicating that the elements (**name** and **surname**) must appear inside the element in this order.

Elements containing text only

To define elements that only contain text, we can define an extension or restriction inside a `simpleContent` element like so:

```
<xs:element name="element">
   <xs:complexType>
      <xs:simpleContent>
         <xs:extension base="type">
            ....
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>
```

or, as a restriction:

```
<xs:element name="element">
   <xs:complexType>
      <xs:simpleContent>
         <xs:restriction base="type">
            ....
         </xs:restriction>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>
```

Look at the following example of an element containing text only:

```
<accommodation type="hotel"> South Coast Inn</accommodation>
```

One possible definition would be:

```
<xs:element name="accommodation" type="acctype"/>

<xs:complexType name="acctype">
   <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="type" type="xs:string" />
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>
```

Defining types containing text and elements

Study the following element, which only contains both text and elements:

```
<expedition>
We arrive at Chaltén in Patagonia on
<date>22.08.2003</date> ....
</expedition>
```

The **date** element is a child element of **expedition**. One possible definition might be:

```
<xs:element name="expedition">
   <xs:complexType mixed="true">
      <xs:sequence>
         <xs:element name="date" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

To include text as well as elements in the content of **expedition** we need to change the `mixed attribute` to **true**. We can obviously also define the type with a name and use it to define other elements or types.

3.2.6. Indicators for complex types

XSD indicators are used to control how we use elements in complex types. There are seven indicators:

- Order indicators:
  - All
  - Choice

  - Sequence

- Occurrence indicators:
  - maxOccurs
  - minOccurs

- Group indicators:
  - Group
  - attributeGroup

Order indicators

Order indicators are used to define how elements occur.

All indicator

The `all indicator` specifies that each child element must appear only once and in any order

```
<xs:complexType name="person">
   <xs:all>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
   </xs:all>
</xs:complexType>
```

If we use this indicator, we can use the `minOccurs` set at 0 or 1, and `maxOccurs` set only at 1 (we will describe these two indicators later on).

The Choice indicator

This indicator specifies that only one of the children may appear.

```
<xs:complexType name="person">
   <xs:choice>
      <xs:element name="Identdocumen" type="xs:string"/>
      <xs:element name="passport" type="xs:string"/>
   </xs:choice>
</xs:complexType>
```

The Sequence indicator

This indicator specifies that the children must appear in a specific order:

```
<xs:complexType name="person">
   <xs:sequence>
```

```
      <xs:element name="name" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

Occurrence indicators

Occurrence indicators are used to define how often an element can occur.

For all order and group indicators (**any**, **all**, **choice**, **sequence**, **group**) the default value of `maxOccurs` and `minOccurs` is 1.

maxOccurs indicator

The maxOccurs indicator specifies the maximum number of times that an element can occur.

For an element to appear an unlimited number of times, we need to assign `maxOccurs="unbounded"`.

minOccurs indicator

This indicator defines the minimum number of times that an element must appear.

```
<xs:element name="student">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="name" type="xs:string"/>
         <xs:element name="subject" type="xs:string" maxOccurs="10" minOccurs="0"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

This example shows how to use `minOccurs` and `maxOccurs` to limit the appearance of the **subject** between 0 and 10 times.

Group indicators

Group indicators are used to define groups of related elements.

Element groups

Element groups are defined as follows:

```
<xs:group name="groupname">
   ...
```

```
</xs:group>
```

We must use an order indicator in our definition. For example:

```
<xs:group name="personGroup">
   <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
      <xs:element name="surname2" type="xs:string"/>
      <xs:element name="DOB" type="xs:date"/>
   </xs:sequence>
</xs:group>
```

We can now use this group to define a type, element, etc. For example:

```
<xs:complexType name="studentInfo">  <xs:sequence> <xs:group ref="personGroup"/>  <xs:element name="departmen
```

Attribute groups

Attribute groups behave in a similar way to element groups and are defined with `attributeGroup` like this:

```
<xs:attributeGroup name="name">
   ...
</xs:attributeGroup>
```

For example:

```
<xs:attributeGroup name="personAttGroup">
   <xs:attribute name="name" type="xs:string"/>
   <xs:attribute name="surname" type="xs:string"/>
   <xs:attribute name="surname2" type="xs:string"/>
   <xs:attribute name="DOB" type="xs:date"/>
</xs:attributeGroup>
```

It can then be used like this:

```
<xs:element name="student">
   <xs:complexType>
      <xs:attributeGroup ref="personAttGroup"/>
   </xs:complexType>
</xs:element>
```

4. Transformations: XSLT

XSL (eXtensible Stylesheet Languageis an XML language for expressing style sheets (how a specific XML language should be represented). It has three main components: XSLT (XSL Transformations), XPath and XSL-FO (XSL-Formatting Objects).

In contrast to HTML, where the meaning of each tag is clearly defined (paragraph break, line break, header, bold) and assigning styles (fonts, sizes, colours, etc.) to these tags is a simple task, tags are not defined in XML. Instead, the user defines them. In XML, the `table` tag can represent either an HTML table or the measurements for a wooden table, so browsers do not know how to represent the tags. As a result, the presentation language of style sheets must describe how to display an XML document more clearly.

The XML style language, XSL, has three main components:

- XSLT, a document transformation language.
- XPath, a language for referencing parts of XML documents.
- XSL-FO, an XML document formatting language.

With these three components, XSL can:

- Transform XML into XML, such as XHTML or WML.
- Filter and/or sort XML data.
- Define parts of an XML document.
- Format an XML document based on the values of stored data.
- Extract XML data to XSL-FO to generate files like PDFs.

XSL is a standard W3C language that was standardised in two stages. The first, in November 1999, included XSLT and XPath, while the second was completed in October 2001 and included XSL-FO.

XSLT is the part of the XML standard used to transform XML documents into other XML documents (for example, XHTML, WML etc).

Normally, XSLT does this by transforming each XML element into another XML element. XSLT can also add other XML elements on output and it can remove elements. It can resort and reposition elements and check and make decisions on which elements to display.

During transformation, XSLT uses XPath to specify or reference parts of the document that follow one or more defined patterns. When it comes across a matching pattern, XSLT transforms the matching part of the original document into the result document. The non-matching parts are not transformed and remain unchanged in the result document.

4.1. Simple transformation

Like virtually all W3C recommendations, XSLT is essentially an XML language and must start with a root element. This root element is of the

`xsl:stylesheet` or `xsl:transform` type (the two are equivalent). The correct way to use them is:

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

This declaration identifies the namespace recommended by the W3C. If we use this namespace, we must also include the `version` attribute with the value **1.0**.

Incorrect declaration

In draft versions of the standard, the correct declaration of a style sheet was:

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

This declaration is now obsolete but if you use the IE-5 browser, then you must use it.

The example XML file that we will be transforming is the following one:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<student report="Linus Torvalds">
   <subject id="1">
      <name>
      Basic Programming
      </name>
      <grade>
      Good
      </grade>
   </subject>
   <subject id="2">
      <name>
      Operating systems
      </name>
```

```
      <grade>
      Excellent
      </grade>
   </subject>
   <subject>
      ...
```

This XML document is an academic report on a university student. It is a very basic document but perfectly valid for our needs.

The XSL document for converting this XML document to another XHTML document is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">

   <html>
   <body>
      <h2>Academic Report</h2>
      <table border="1">
      <tr bgcolor="#9acd32">
         <th align="left">Subject</th>
         <th align="left">Grade</th>
      </tr>
      <xsl:for-each select="report/subject">
      <tr>
         <td><xsl:value-of select="name"/x</td>
         <td><xsl:value-of select="grade"/></td>
      </tr>
      </xsl:for-each>
      </table>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

If we name the XSL document `report.xsl` we can link it to our XML by adding a reference to the style sheet at the start of the XML, as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="report.xsl"?>
<student report="Linus Torvalds">
   <subject id="1">
```

```
      <name>
      Basic Programming
      </name>
      <grade>
      Good
      </grade>
   </subject>
   ...
```

If you use a browser with XSL support (Mozilla 1.2 or higher), when opening the XML document, the browser will use the XSL document to transform it into XHTML.

4.2. The `xsl:template element`

An XSL style sheet consists of a series of transformationtemplates. Each `xsl:template element` contains the XSL transformations that must be applied if the template specified in the element matches that found in the XML document.

To specify the XML element to which we must apply the template, we use the `match` attribute (we can also apply the template to the entire XML document by specifying `match="/"`). The values that we can give the `match` attribute are specified by the XPath standard.

For example, the following XSL transformation returns a specific XHTML code when processing the document with the student report.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <body>
      <h2>Academic Report</h2>
      <table border="1">
      <tr bgcolor="#9acd32">
         <th align="left">Subject</th>
         <th align="left">Grade</th>
      </tr>
      </table>
   </body>
   </html>
</xsl:template>
```

```
</xsl:stylesheet>
```

As we will see if we test this XSL document, it only produces a page header. If we analyse the XSL document, we see that it has a template that is applied when it matches the root element of the document (`match="/"`) and prints the contents of this tag in the result.

4.3. The `value-of` element

The `value-of` element is used to select and add the value of the selected XML element to the output stream.

For example, if we add the following code to our previous example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
   <html>
   <body>
    <h2>Academic Report</h2>
       <table border="1">
       <tr bgcolor="#9acd32">
          <th align="left">Subject</th>
          <th align="left">Grade</th>
       </tr>
       <tr>
          <td><xsl:value-of
             select="report/subject/name"/></td>
          <td><xsl:value-of
             select="report/subject/grade"/></td>
       </tr>
       </table>
   </body>
   </html>
</xsl:template>
</xsl:stylesheet>
```

we see that the first grade of the report appears in the result. This is because the `value-of` tags select the value of the first element that matches the specified pattern.

4.4. The `xsl:for-each` element

The `xsl:for-each` element in XSL can be used to select each of the elements of the XML document belonging to a given set.

In the above example, where only the first grade of the report appeared, we can add an `xsl:for-each` that will run through the entire report as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <body>
      <h2>Academic Report</h2>
      <table border="1">
      <tr bgcolor="#9acd32">
         <th align="left">Subject</th>
         <th align="left">Grade</th>
      </tr>
         <xsl:for-each select="report/subject">
      <tr>
         <td><xsl:value-of select="name"/></td>
         <td><xsl:value-of select="grade"/></td>
      </tr>
      </xsl:for-each>
   </table>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

This will give us a list of all grades for all subjects.

4.5. Sorting information: `xsl:sort`

To sort our output, we simply need to add an `xsl:sort` element to the `xsl:for-each` element in our XSL file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <body>
      <h2>Academic Report</h2>
      <table border="1">
      <tr bgcolor="#9acd32">
         <th align="left">Subject</th>
```

```
         <th align="left">Grade</th>
      </tr>
         <xsl:for-each select="report/subject">
         <xsl:sort select="name"/>
      <tr>
         <td><xsl:value-of select="name"/x/td>
         <td><xsl:value-of select="grade"/></td>
      </tr>
      </xsl:for-each>
      </table>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

The `select` attribute is used to indicate the element that we will use to base sorting on; in this case, the name of the subject.

4.6. Conditions in XSL

There are two XSL elements for implementing conditions in our transformations. These are `xsl:if` and `xsl:choose`.

4.6.1. The xsl:if element

The `xsl:if` element allows us to use a template only if the specified condition is met (it is true).

An example of `xsl:if` format is:

```
<xsl:if test="grade &lt; 5">
   .....it will only appear with grades of less than 5.....
</xsl:if>
```

We can modify the above code to show only grades above 5, for example.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <body>
      <h2>Academic Report</h2>
      <table border="1">
      <tr bgcolor="#9acd32">
```

```
         <th align="left">Subject</th>
         <th align="left">Grade</th>
      </tr>
      <xsl:for-each select="report/subject">
      <xsl:if test="grade &gt; 5">
      <tr>
         <td><xsl:value-of select="name"/></td>
         <td><xsl:value-of select="grade"/></td>
      </tr>
      </xsl:if>
      </xsl:for-each>
      </table>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

4.6.2. The `xsl:choose element`

The `xsl:choose element` (together with `xsl:when` and `xsl:otherwise`) is used to express multiple conditional tests. In other words, using a multiple condition (with multiple possible values), we can obtain diverse results.

An example of `dexsl:choose` format is:

```
<xsl:choose>
   <xsl:when test="grade  &lt; 5">
      ... código (fail) ....
   </xsl:when>
   <xsl:when test="grade &lt; 9">
      ... code (normal) ....
   </xsl:when>
   <xsl:otherwise>
      ... code (excellent) ....
   </xsl:otherwise>
</xsl:choose>
```

We can modify the above example so that grades under 5 appear in red:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <body>
```

```
      <h2>Academic Report</h2>
      <table border="1">
      <tr bgcolor="#9acd32">
         <th align="left">Subject</th>
         <th align="left">Grade</th>
      </tr>
         <xsl:for-each select="report/subject">
            <tr>
            <td><xsl:value-of select="name"/></td>
            <td>
            <xsl:choose>
         <xsl:when test="grade &lt; 5">
            <font color="#FF0000">
               <xsl:value-of select="grade"/>
            </font>
         </xsl:when>
         <xsl:otherwise>
            <xsl:value-of select="grade"/>
         </xsl:otherwise>
            </xsl:choose>
            </td>
            </tr>
            </xsl:if>
            </xsl:for-each>
      </table>
   </body>
   </html>
</xsl:template>

</xsl:stylesheet>
```

4.7. The `xsl:apply-templates` element

`apply-templates` applies a template to the current element or child elements of the current element. The `select` attribute is used to process only the child elements that we specify and the order in which they are processed.

For example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
   <html>
   <body>
      <h2>Academic Report</h2>
```

```
      <xsl:apply-templates />
   </body>
   </html>
</xsl:template>

<xsl:template match="report">
   <xsl:apply-templates select="subject"/>
</xsl:template>

<xsl:template match="subject">
<p>
   <xsl:apply-templates select="name"/>
   <xsl:apply-templates select="grade"/>
</p>
</xsl:template>

<xsl:template match="name">
Name: <xsl:value-of select="."/></span>
<br />
</xsl:template>

<xsl:template match="grade">
Grade: <xsl:value-of select="."/></span>
<br />
</xsl:template>

</xsl:stylesheet>
```

As we can see, this organisation is much more modular and allows for better maintenance and revision of the style sheet.

4.8. Introduction to XPath

XPath is a W3C recommendation that defines a set of rules for referencing parts of an XML document. Based on W3C's definition, we can define XPath as:

- XPath is a definition of syntax for referencing parts of an XML document.
- XPath uses "paths" to define XML elements.
- XPath defines a library of functions.
- XPath is a basic element of XSL.
- XPath is not defined in XML.
- XPath is a W3C recommendation.

XPath uses expressions similar to the "paths" of files in operating systems (e.g. `/home/user/file.txt`).

Let's take the following XML file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<student report="Linus Torvalds">
   <subject id="1">
      <name>
      Basic Programming
      </name>
      <grade>
      Good
      </grade>
   </subject>
   <subject id="2">
      <name>
      Operating systems
      </name>
      <grade>
      Excellent
      </grade>
   </subject>
   <subject>
      ...
```

Supplementary content

Like file systems, if an element begins with / this indicates an absolute path to an element.

Supplementary content

If an element begins with // all elements matching the criterion will be selected, regardless of their location in the XML tree.

The following XPath expression selects the root **report** element:

```
/report
```

This expression selects all **subject** elements of the **report element**:

```
/report/subject
```

This one selects all grade elements of all **subject** elements of the **report element**:

```
/report/subject/grade
```

4.8.1. Selecting unknown elements

As with all file systems, we can use special characters (*****) to indicate unknown elements.

The following expression selects all child elements of all **subject** elements of the **report element**:

```
/report/subject/*
```

This expression selects all **name** elements descended from the **report element** regardless of the parent element:

```
/report/*/name
```

This expression selects all elements in the document:

```
//*
```

4.8.2. Selecting branches of the tree

We can specify which parts of the node tree we wish to select using square brackets ([ ]) in XPath expressions.

For example, we can select the first **subject** element of the **report element**

```
/report/subject[1]
```

This expression selects the last child **subject** element of the **report element**:

```
/report/subject[last()]
```

This expression selects all child **subject** elements of the **report element** containing a **grade element**:

```
/report/subject[grade]
```

This one also forces the **grade element** to have a specific value:

```
/report/subject[grade>5]
```

This expression selects the names of subjects with a **grade element** of a specific value:

```
/report/subject[grade>5]/name
```

4.8.3. Selecting multiple branches

We can use the | operator in XPath expressions to select multiple paths.

For example, the following expression selects all **grade** and **name** elements of the **subject** element of the **report element**:

```
/report/subject/name/report/subject/grade
```

This expression selects all **name** and **grade** elements in the document:

```
//name//note
```

4.8.4. Selecting attributes

In XPath, attributes are specified with the prefix @.

This XPath expression selects all attributes called **id**:

```
//@id
```

The following expression selects all **subject** elements with an **id** attribute of a specific value:

```
//subject[@id=1]
```

The following expression selects all **subject** elements with any attribute:

```
//subject[@*]
```

4.8.5. Functions library

XPath has a functions library that can be used in XPath predicates to fine-tune our selection. Included in this library is `last()` the function we saw earlier.

Some of the most important functions include:

Name	Syntax	Description
count()	n=count(nodes)	Returns the number of nodes in the set provided
id()	nodes=id(value)	Selects nodes for their unique ID
last()	n=last()	Returns the position of the last node in the list of nodes to be processed.
name()	cad=name()	Returns the name of the node
sum()	n=sum(nodes)	Returns the value of the sum of the specified set of nodes

There are also several functions for handling strings, numbers, etc.

5. Practical: creating an XML document with its corresponding XML Schema and transformations with XSLT

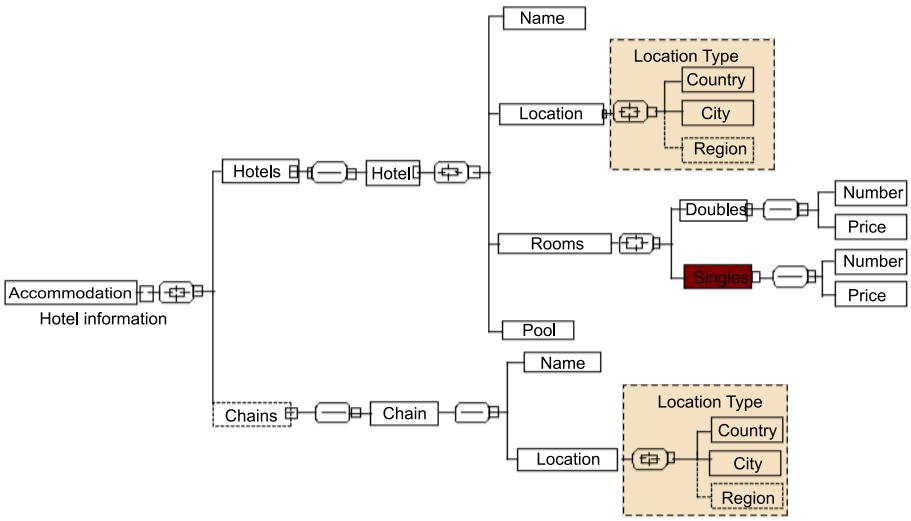
We are going to create an XML document to store hotel information. To do so, we must first design the XML Schema file we are going to use. This will allow us to validate our design as we add the different parts to it.

The XML Schema produced by our design will be:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:element name="Accommodation">
      <xs:annotation>
         <xs:documentation>Hotel information</xs:documentation>
      </xs:annotation>
      <xs:complexType>
         <xs:all>
            <xs:element name="Hotels">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="Hotel" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:all>
                              <xs:element name="Name" type="xs:string"/>
                              <xs:element name="Location"
                                 type="LocationType"/>
                              <xs:element name="Rooms">
                                 <xs:complexType>
                                    <xs:all>
                                       <xs:element name="Doubles">
                                          <xs:complexType>
                                             <xs:sequence>
                                                <xs:element
                                                   name="Number" type="xs:int"/>
                                                <xs:element
                                                   name="Price" type="xs:float"/>
                                             </xs:sequence>
                                          </xs:complexType>
                                       </xs:element>
                                       <xs:element name="Singles">
                                          <xs:complexType>
```

```
                                        <xs:sequence>
                                           <xs:element
                                              name="Number" type="xs:int"/>
                                           <xs:element
                                              name="Price" type="xs:float"/>
                                        </xs:sequence>
                                     </xs:complexType>
                                  </xs:element>
                               </xs:all>
                            </xs:complexType>
                         </xs:element>
                         <xs:element name="Pool" type="xs:boolean"/>
                         <xs:element name="Category" type="xs:int"/>
                      </xs:all>
                   </xs:complexType>
                </xs:element>
             </xs:sequence>
          </xs:complexType>
       </xs:element>
       <xs:element name="Chains" minOccurs="0">
          <xs:complexType>
             <xs:sequence>
                <xs:element name="Chain" maxOccurs="unbounded">
                   <xs:complexType>
                      <xs:sequence>
                         <xs:element name="Name"/>
                         <xs:element name="Location"
                            type="LocationType"/>
                      </xs:sequence>
                   </xs:complexType>
                </xs:element>
             </xs:sequence>
          </xs:complexType>
       </xs:element>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:complexType name="LocationType">
    <xs:all>
       <xs:element name="Country"/>
       <xs:element name="City"/>
       <xs:element name="Region" minOccurs="0"/>
    </xs:all>
</xs:complexType>
</xs:schema>
```

The design we have chosen for our XML is:



We will now create a document based on the schema we have designed. With the use of test data, this document will be:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="Hotels.xslt"?>
<Accommodation
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="hotels.xsd">
   <Hotels>
      <Hotel>
         <Name>Hotel Port Aventura</Name>
         <Location>
            <Country>Spain</Country>
            <City>Salou</City>
            <Region>Costa Dorada</Region>
         </Location>
         <Rooms>
            <Doubles>
               <Number>50</Number>
               <Price>133</Price>
            </Doubles>
            <Singles>
               <Number>10</Number>
               <Price>61</Price>
            </Singles>
         </Rooms>
         <Pool>false</Pool>
         <Category>3</Category>
      </Hotel>
      <Hotel>
         <Name>Hotel Arts</Name>
```

```
            <Location>
               <Country>Spain</Country>
               <City>Barcelona</City>
            </Location>
            <Rooms>
               <Doubles>
                  <Number>250</Number>
                  <Price>750</Price>
               </Doubles>
               <Singles>
                  <Number>50</Number>
                  <Price>310</Price>
               </Singles>
            </Rooms>
            <Pool>true</Pool>
            <Category>5</Category>
         </Hotel>
         <Hotel>
            <Name>Parador Seu d'Urgell</Name>
            <Location>
               <Country>Spain</Country>
               <City>Seu d'Urgell</City>
               <region>Pyrenees</Region>
            </Location>
            <Rooms>
               <Doubles>
                  <Number>40</Number>
                  <Price>91.5</Price>
               </Doubles>
               <Singles>
                  <Number>2</Number>
                  <Price>44.4</Price>
               </Singles>
            </Rooms>
            <Pool>true</Pool>
            <Category>4</Category>
         </Hotel>
      </Hotels>
      <Chains>
         <Chain>
            <Name>HUSA</Name>
            <Location>
               <Country>Spain</Country>
               <City>Barcelona</City>
            </Location>
         </Chain>
         <Chain>
```

```
         <Name>NH Hoteles</Name>
         <Location>
            <Country>Spain</Country>
            <City>Pamplona</City>
         </Location>
      </Chain>
      <Chain>
         <Name>Paradores de Turismo</Name>
         <Location>
            <Country>Spain</Country>
            <City>Madrid</City>
         </Location>
      </Chain>
   </Chains>
</Accommodation>
```

We will now use the following XSLT document:

```
<?xml version="1.0" encoding="UTF-8"?>
   <xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" version="1.0"
      encoding="UTF-8" indent="yes"/>

   <xsl:template match="Location">
      <xsl:value-of select="City"  />,
      <xsl:value-of select="Country" />
      <xsl:if test="Region ">
         <i>(<xsl:value-of select="Region"  />)</i>
      </xsl:if>
   </xsl:template>

   <xsl:template match="Hotels">
      <hl>List of hotels</hl>
      <xsl:for-each select=".">
         <xsl:apply-templates select="Hotel" />
      </xsl:for-each>
   </xsl:template>

   <xsl:template match="Pool">
      <xsl:choose>
         <xsl:when test="node() = 'true'">
            <b>Yes</b>
         </xsl:when>
         <xsl:otherwise>
            <b>No</b>
         </xsl:otherwise>
```

```
      </xsl:choose>
   </xsl:template>

   <xsl:template match="Hotel">
      <h2>Hotel</h2>
      Name: <xsl:value-of select="Name" /> (Stars:
         <xsl:value-of select="Category"  />)<br />
      Location: <xsl:apply-templates select="Location" /xbr />
      Pool: <xsl:apply-templates select="Pool"  /><br />
      <br />
      <h3>Rooms</h3>
      <table>
         <tbody>
            <tr>
               <th>Type</th>
               <th>Number</th>
               <th>Price</th>
            </tr>
            <tr>
               <td>Singles</td>
               <td><xsl:value-of
                  select="Rooms/Singles/Number" /></td>
               <td><xsl:value-of select="Rooms/Singles/Price" /></td>
            </tr>
            <tr>
               <td>Doubles</td>
               <td><xsl:value-of select="Rooms/Doubles/Number" /></td>
               <td><xsl:value-of select="Rooms/Doubles/Price" /></td>
            </tr>
         </tbody>
      </table>
   </xsl:template>

   <xsl:template match="Chain">
      <h2>Chain</h2>
      Name: <xsl:value-of select="Name" /> <br />
      Location:  <xsl:apply-templates select="Location" /><br />
   </xsl:template>

   <xsl:template match="Chains">
      <hl>List of hotel chains</hl>
      <xsl:for-each select=".">
         <xsl:apply-templates select="Chain" />
      </xsl:for-each>
   </xsl:template>

   <xsl:template match="/">
```

```
      <html>
         <head>
         <title>Hotel information</title>
         </head>
         <body>
            <xsl:apply-templates />
         </body>
      </html>
   </xsl:template>
</xsl:stylesheet>
```

To obtain a list of the hotels in HTML:

```
<?xml version="1.0" encoding="UTF-8"?>
   <html>
      <head>
         <title>Hotel information</title>
      </head>
      <body>
         <h1>List of hotels</h1>
         <h2>Hotel</h2>
         Name: Hotel Port Aventura (Stars: 3)<br/>
         Location: Salou, Spain<i> (Costa Dorada)</i><br/>
         Pool:  <b>No</b><br/><br/><h3>Rooms</h3>
         <table>
            <tbody>
               <tr>
                  <th>Type</th>
                  <th>Number</th>
                  <th>Price</th>
               </tr>
               <tr>
                  <td>Singles</td>
                  <td>10</td>
                  <td>61</td>
               </tr>
               <tr>
                  <td>Doubles</td>
                  <td>50</td>
                  <td>133</td>
               </tr>
            </tbody>
         </table>
         <h2>Hotel</h2>
         Name: Hotel Arts (Stars: 5)<br/>
         Location: Barcelona, Spain<br/>
         Pool:  <b>Yes</b><br/><br/>
```

```
            <h3>Rooms</h3>
            <table>
               <tbody>
                  <tr>
                     <th>Type</th>
                     <th>Number</th>
                     <th>Price</th>
                  </tr>
                  <tr>
                     <td>Singles</td>
                     <td>50</td>
                     <td>310</td>
                  </tr>
                  <tr>
                     <td>Doubles</td>
                     <td>250</td>
                     <td>750</td>
                  </tr>
               </tbody>
            </table>
            <h2>Hotel</h2>
            Name: Parador Seu d'Urgell (Stars: 4)<br/>
            Location: Seu d'Urgell, Spain<i> (Pyrenees)</i><br/>
            Pool:  <b>Yes</b><br/><br/>
            <h3>Rooms</h3>
            <table>
               <tbody>
                  <tr>
                     <th>Type</th>
                     <th>Number</th>
                     <th>Price</th>
                  </tr>
                  <tr>
                     <td>Singles</td>
                     <td>2</td>
                     <td>44.4</td>
                  </tr>
                  <tr>
                     <td>Doubles</td>
                     <td>40</td>
                     <td>91.5</td>
                  </tr>
               </tbody>
            </table>
            <h1>List of hotel chains</h1>
            <h2>Chain</h2>
            Name: HUSA<br/>
```

```
            Location: Barcelona, Spain<br/>
            <h2>Chain</h2> Name: NH Hoteles<br/>
            Location: Pamplona, Spain<br/>
            <h2>Chain</h2> Name: Paradores de Turismo<br/>
            Location: Madrid, Spain<br/>
         </body>
      </html>
```

Bibliography

Rusty Harold, Elliotte; Means, W. Scott (2002). *XML in a Nutshell, 2nd Edition*. O'Reilly.

Tidwell, Doug (2001). *XSLT*. O'Reilly.

van der Vlist, Eric (2001). *XML Schema*. O'Reilly.

Ray, Eric T. (2001). *Learning XML*. O'Reilly.

Dynamic content

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Carles Mateu

PID_00148398

Universitat Oberta de Catalunya
www.uoc.edu

Index

1. CGI

One of the first mechanisms for generating dynamic content for the web was the API called CGI (the acronym of *Common Gateway Interface*). This very simple mechanism allows a web server to run a program written in any programming language (whether in response to a HTML form, from a link, etc.), that can pass certain parameters to it (either from the user, via forms, or through server configuration parameters, execution environment, etc.) and lastly, it makes it possible to send the result of the execution of this program to the user as a web page or any other type of content (graphic, etc).

With this simple mechanism, web pages that had static and unmovable content,—until the appearance of CGIs,are generated dynamically in response to specific requests. This opens up a whole new world for web application programmers. We will now look at the API of CGIs, often relegated to a secondary role because of the many problems it suffers, the main one being performance issues.

1.1. Introduction to CGIs

Unlike with servlets, etc., there are no restrictions on the programming language we can use to write a CGI. We can use those *scripts* written in the *shell* language of the operating system, programs written in the assembler and the broad range of programming languages currently available: C, C++, Perl, Python, etc. Until now, the most popular language for writing CGIs has been Perl, as it offers utilities to the programmer that greatly simplify the task of writing CGI programs.

1.2. Communicating with CGIs

The first thing to bear in mind when writing programs like CGIs is the mechanism of communication provided by the web server. We have two options for sending data to a CGI (the data generally come from a user, usually via a form):

- GET method. The GET method passes all information (except for files) to the CGI in the address line of the HTTP request.
- POST method. The POST method passes all information to the CGI in the standard entrance, including files.

Once it receives a request that it needs to direct to a CGI file, the server executes this program, the CGI, and sends it the information through environment variables (or through the standard entrance, if applicable). Some of the environment variables defined by the CGI standard are:

`SERVER_NAME` Name of the server.

`SERVER_PROTOCOL` Protocol used by the request.

`REQUEST_METHOD` Method used for invocation (GET or POST).

`PATH_INFO` Information on the path specified in the request.

`PATH_TRANSLATED` Physical path to the location of the CGI on the server.

`SCRIPT_NAME` Name of the CGI.

`REMOTE_ADDR` IP address of the computer making the request.

`REMOTE_HOST` Name of the computer making the request.

`REMOTE_USER` User making the request.

`AUTH_TYPE` Type of authentication.

`CONTENT_TYPE` MIME type of the request content, particularly useful in POST requests.

`CONTENT_LENGTH` Size of the content, particularly useful in POST requests.

Most web servers also provide the `QUERY_STRING`, which contains the data of the request if it is GET type or if data has been added to the URL. Some web servers add extra data to the environment. Most of these additional variables begin with `HTTP_` to avoid conflicts with later versions of the standard.

> For example, the Roxen web server adds a variable called `QUERY_` which is a parameter for each parameter of a form.

1.3. CGI response

CGIs respond to requests by constructing part of the HTTP response that will be received by the clients themselves. Firstly, they must indicate the MIME type of the served content. They can then add extra fields (those specified in the HTTP standard). The content must appear after a blank line of separation.

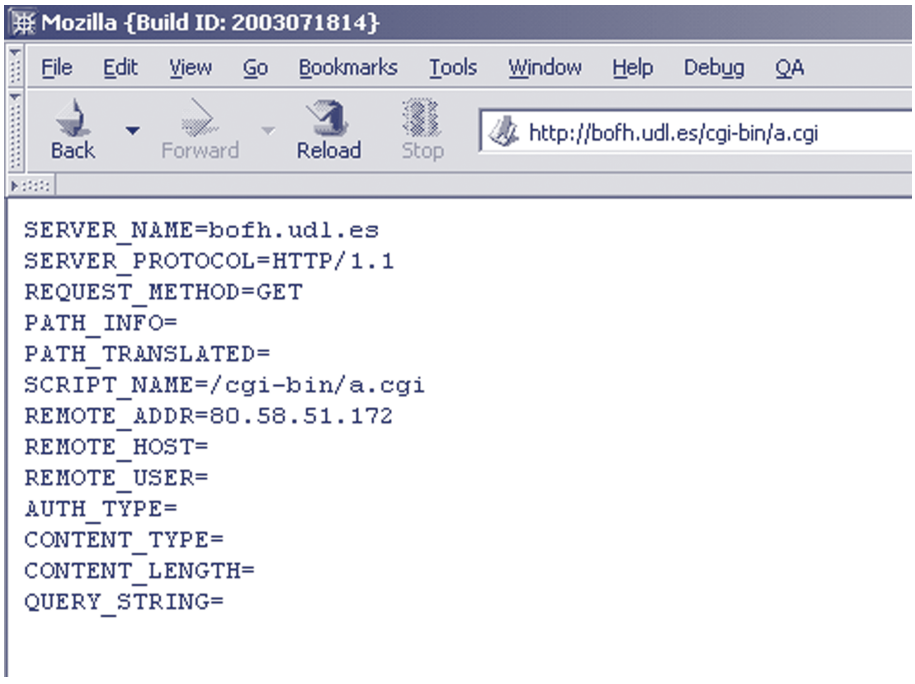
The simplest possible CGI, written here in *shell script* and enumerating the environment variables commented above, is:

```
#!/bin/sh

echo Content-type: text/plain
echo
echo
echo SERVER_NAME=$SERVER_NAME
echo SERVER_PROTOCOL=$SERVER_PROTOCOL
echo REQUEST_METHOD=$REQUEST_METHOD
echo PATH_INFO=$PATH_INFO
echo PATH_TRANSLATED=$PATH_TRANSLATED
echo SCRIPT_NAME=$SCRIPT_NAME
echo REMOTE_ADDR=$REMOTE_ADDR
echo REMOTE_HOST=$REMOTE_HOST
echo REMOTE_USER=$REMOTE_USER
echo AUTH_TYPE=$AUTH_TYPE
echo CONTENT_TYPE=$CONTENT_TYPE
echo CONTENT_LENGTH=$CONTENT_LENGTH
echo QUERY_STRING=$QUERY_STRING
```

As we can see in this example (the *shell script* syntax used is very simple), to list the environment variables received, we send the type of content, followed by a compulsory blank line and all of the environment variables mentioned.

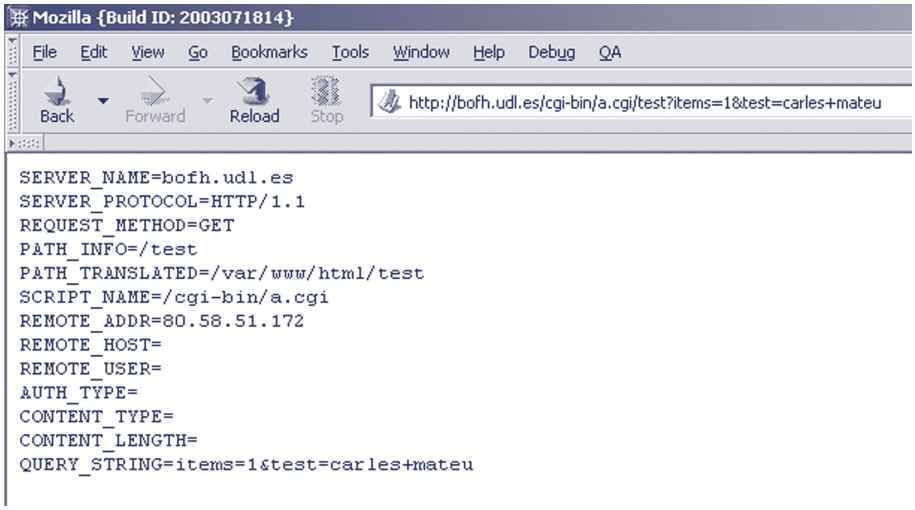
If we execute this server with no additional parameters, we end up with:

Figure 17.



As we can see, if we simply call the CGI without parameters and it does not activate a form, it will have few variables with values. However, if we call the CGI by passing parameters and an extra PATH (note the directory after the name of the CGI), the result is as follows:

Figure 18.



1.3.1. Decoding of the QUERY_STRING

As we have seen in the examples above, the parameters sent to our CGI used a specific and very special coding. One of the disadvantages of using CGI compared to more modern alternatives like servlets is that we need to decode and analyse this string manually. Fortunately, there are libraries for almost every programming language that make this task easier.

The coding rules are as follows:

- We separate the list of parameters from the rest of the URL address with the character?

- We separate parameters (which are always in name, value pairs) using the character &. In some cases, the character ; is accepted as a substitute for separation.

- The names of parameters are separated from the values with the character =.

- The special characters are replaced according to the following table:
  - The ' ' character (blank space) is changed to +.

  - Non-alphanumeric characters and special characters, like those used for coding (+, etc.), are represented as %HH, where HH represents the hex value of the ASCII code of the character.

– Line breaks are represented as `%0D %0`A.

1.4. Redirections

We can redirect the client to a different page from a CGI program. To do so, we must not return the standard HTML code preceded by `Content-type`. Instead, we must return a status code field followed by the location of the new page, as in the example:

```
#include <stdio.h>

int main()
 {
   printf("Status: 302\r\n");
   printf("Location: new.html\r\n");
   exit(1);
 }
```

2. PHP

PHP (a recursive acronym of *hypertext preprocessor*), is a simple language with an easy syntax similar to that of languages like Perl, C y C++. It is fast, interpreted, object-oriented and cross-platform, and there are many libraries available for it. PHP is an ideal language for learning to develop web applications and for developing complex web applications. On top of this, PHP has the advantage that the PHP interpreter, the range of modules and the number of libraries developed for PHP are free software, so the PHP programmer has recourse to an astonishing arsenal of free software tools with which to develop applications.

PHP is usually used with Perl, Apache, MySQL or PostgreSQL in Linux systems, forming an economical (all of the components are free software), powerful and versatile combination. The expansion of this combination has been so great that it has even been christened with the name LAMP (formed by the initials of each of the products).

Apache, like other web servers, including Roxen, can incorporate PHP as a module of the server itself. This means that applications written in PHP are much faster than normal CGI applications.

2.1. The workings of PHP

If we request a PHP page from our server, the latter sends the page to the PHP interpreter that executes it (in fact, it is simply a program) and returns the result (generally HTML) to the web server, which, in turn, sends it to the client.

Let's suppose we have a PHP page with the following content:

```
<?php echo "<h1>Hello world!</h1>";?>
```

If we have this code in a file with the extension `.php`, the server will send the page to the PHP interpreter, which will then execute the page and obtain the following result:

```
<h1>Hello world!</h1>
```

The server will send it to the client browser that requested the page and the message will appear on the latter's screen. We will see how PHP allows HTML and PHP to be combined on the same page, which means that working with

the latter is considerably easier. However, this can also be a hidden danger because it complicates matters if web designers and programmers are working together on the site.

In systems where PHP is installed, we have a global PHP configuration file called `php.ini`, which can help us to configure certain global settings. It is a good idea to check this file because, although the default values are usually correct, we may wish to make certain changes.

2.2. PHP syntax

To introduce the syntax of the language, we will analyse a basic PHP program:

```
<?php
  $MYVAR = "1234";
  $myvar = "4321";
  echo $MYVAR. "<br>\n";
  echo $myvar."<br>\n";
?>
```

If we run this program (display it in a browser), the result will be as follows:

```
1234<br>
4321<br>
```

The first point we need to make is that PHP code blocks are delimited in HTML by `<?php` and `?>`. We can therefore write an HTML page including several PHP instruction blocks:

```
<HTML>
 <HEAD>
   <TITLE>Document title</TITLE>
 </HEAD>
<BODY>
 <h1>Header H1</h1>
 <?php echo "Hello" ?>
 <h1>Second header H1</h1>
 <?php
 $MYVAR = 1234;
 $myvar = 4321;
 echo $MYVAR. "<br>";
 echo $myvar."<br>";
 // This program displays some numbers on the screen
 ?>
 </BODY>
</HTML>
```

The second point worth mentioning is that the names of variables are distinguishable because they always begin with $ and, as in C/C++, they are *case-sensitive*, meaning that we differentiate between capital and lower-case letters. Note also that to concatenate text (variables and "<br>"), we use the full stop character "." and that all statements end in ";".

You should also be aware that although variables are numerical, they can be concatenated with a text ("<br>"). In this case, the interpreter converts the numerical value of the variable into text to perform concatenation.

You will see that there is a comment inside the code. This comment will not affect the program in any way nor will it be sent to the client browser (in fact, the client browser never receives PHP code). There are two options for inserting comments in our code:

```
// Single-line comment
/*  This comment takes up several lines.
    So we use this other marker
    to indicate the start and end of the comment */
```

2.3. Variables

In PHP, we do not need to declare *a priori* the variable or type of variable that we are going to use. PHP will declare the variable and assign the correct type of data to it when we use it for the first time:

```
<?php $string = "Hello World";
 $number = 100;
 $decimal = 8.5;
?>
```

As we can see, the three variables were defined when they were assigned a value and we did not need to define types.

In PHP, variables can basically have two scopes: global, where they can be accessed from the entire code, and local, where they are only accessible from the function in which we create them. To assign a global scope to a variable, simply declare it (in this case, you must make a variable declaration) and use the reserved word `global` in the declaration:

```
<?php
 global $test;
?>
```

The scope of variables that we do not qualify as global but which are defined outside a function, will be global.

We simply need to define a variable within a function. In this case, the scope will be restricted to the function where we declare it.

```
<?php
 global $variable; // Global variable
 $a=1; // Implicit global variable
 Add function()
 {
   $b=1; // b is a local variable
   $res=$a+$b; // res is a local variable
 }
?>
```

We can see that both `a` and `variable` are `global variables` while `b` and `res` are local variables.

In PHP, we also have *arrays*. These are variables that can contain lists of elements, which we access through an index.

```
<?php
 $seas = array(); //with array() we declare an array
 $seas[0]= "Mediterranean";
 $seas[1]= "Aral";
 $seas[2]= "Dead";
?>
```

As you can see, we have declared the `seas` variable with a call to `array()`. This tells PHP that the variable is an elements array.

To access the individual elements of the array, we will need to use the name of the vector and indicate the position of the element we wish to access in square brackets. In PHP, array numbering starts at 0.

As well as arrays with numerical indices, PHP also supports arrays with text string indices:

```
<?php
 $mountains=array(); //with array() we declare an array
 $mountains["Everest"]= "Himalaya";
 $mountains["Fitz Roy"] = "Andes";
 $mountains["Montblanc"] = "Alps";

 echo $mountains["Everest"]; // Will print Himalaya
?>
```

2.4. Operators

Operators are symbols that are used to perform both mathematical operations and comparisons or logical operations.

The most common ones in PHP are:

- Mathematical operators:
  a) + Adds several numbers: `5 + 4 = 9`.

  b) - Subtracts several numbers: `5 - 4 = 1`.

  c) * Performs a multiplication: `3 * 3 = 9`.

  d) / Performs a division: `10/2 = 5`.

  e) % Returns the remainder of a division: `10 % 3 = 1`.

  f) ++ Increments by 1: `$v++ (Increments $v by 1)`.

  g) -- Decrements by 1: `$v-- (Decrements $v by 1)`.

- Comparison operators:
  a) == Evaluates as true if the condition for equality is met:
  `2 == 2 (True).`

  b) != Evaluates as true if the condition for equality is not met:
  `2 != 2 (False).`

  c) < Evaluates as true if a number is less than another
  `2 < 5 (True).`

  d) > Evaluates as true if a number is greater than another
  `6 > 4 (True).`

  e) <=Evaluates as true if a number is less than or equal to another
  `2 <= 5 (True).`

  f) >=Evaluates as true if a number is greater than or equal to another
  `6 >= 4 (True).`

- Logical operators:
  a) && Evaluates as true if the two operators are true.

  b) || Evaluates as true if one of the operators is true.

c) `And` Evaluates as true if the operators are true.

d) `Or` Evaluates as true if one of the operators is true.

e) `Xor` Evaluates as true if one operator or another is true.

f) `!` Reverses the true value of the operator.

This example indicates the most common mathematical operators:

```
<?php
 $a = 5;
 $b = 10;
 $c = ($a + $b); //$c is equal to 15
 $d = ($b - $a); //$d is equal to 5
 $e = ($a * $b); //$e is equal to 50
 $f = ($b / $a); //$f is equal to 2
 $g = ($b % $a); //$g is equal to 0
?>
```

2.5. Control structures

PHP control structures allow us to control the flow of operation of our program, ensuring that portions of code are executed at all times in line with certain conditions.

2.5.1. Conditionals

Conditionals are structures that allow us to perform certain operations only if a given condition is met. They are usually called forks because they allow us to divide the execution flow of the program according to the true value of a statement or condition.

In PHP, we have two main conditionals, the `if/else` conditional and the `switch`.

The `if` conditional is used to choose between code blocks, according to whether or not a condition is met.

```
<?php
 $a = 0;
 $b = 1;
 if($a == $b)
 {
     echo "It turns out that 0 is equal to 1";
 }
 else
```

```
 {
     echo "Everything is as it was. 0 is not equal to 1";
 }
?>
```

If we follow the execution flow of this program, we see that two variables are created initially, `a` and `b`, to which we assign two different numerical values. We then come to the conditional statement `if`. This confirms the truth or compliance of a specified condition. In this case, we have an `==` equality operator returning that the comparison is false; hence, the `if` statement does not execute the first code block, the one it would have executed had the condition been met. Instead, it executes the second, the one preceded by `else`.

We can therefore define the structure of `if/else` as:

```
if(condition)
 {
     code executed if the condition is true
 }
else
 {
     code executed if the condition is false
 }
```

We can check more than one condition by chaining several `if/else`:

```
if(condition1)
 if(condition2)
    {
      code executed if condition2 is true
      and condition1 is true
    }
 else
    {
      code executed if condition2 is false
      and condition1 is true
    }
else
  {
    code executed if condition1 is false
  }
```

An advanced case of `if/else` chaining is that corresponding to events where we need to execute a different code depending on the value of a variable. Although we can carry out `if/else` chaining by checking the value of this

variable, if we need to check a number of values, the code can be rather cumbersome. PHP therefore offers a more ideal conditional construction called `switch`.

```
<?php
 $a=1;
 switch($a)
 {
    case 1:
    case 2: echo "A is 1 or 2"; break;
    case 3: echo "A is 3"; break;
    case 4: echo "A is 4"; break;
    case 5: echo "A is 5"; break;
    case 6: echo "A is 6"; break;
    default: echo "A is another value";
 }
```

Executing `switch` is a rather complex task.. I fact, it is very similar to C. The `switch` statement is executed line by line. Initially, no codes or any lines are executed. When it comes to a `case` with a value that matches the value of the `switch` variable, PHP begins to execute the statements. This continues until the end of `switch` or until it comes to a `break`. So, in our example, if the variable has a value or 1 or a value of 2, the same code block is executed.

There is also a special value, `default,` which always matches the value of the variable.

2.5.2. Loops

Loops are another important control structure. These are used to execute a code block repeatedly in accordance with a condition.

PHP has three main loops: `for`, `while` and `foreach`.

The `while loop`

The `while loop` is the simplest of the three but, even so, it is probably the most common one. The loop is executed while the condition we have passed to it is true:

```
<?php
 $a = 1;
 while($a < 4)
 {
   echo "a=$a<br>";
   $a++;
 }
```

```
?>
```

In this case, the loop will be executed four times. Each time it is executed, we will increment the value of `a` and print a message. Each time the code is executed, the `while loop` checks the condition and, if met, executes the code again. The fourth time that it is executed, since `a` will have the value of four, the condition specified will not be met and the loop will no longer be executed.

The `for loop`

For the above type of loop, where the condition for continuing is that a variable increases or decreases with each iteration, we can use a more appropriate type of loop: `for`.

Using `for` the above code would end up as follows:

```
<?php
 for($a=1;$a < 4; $a++)
 {
   echo "a=$a<br>";
 }
?>
```

As we can see, in the case of the `for` loop, in the same statement we declare the variable over which we will be iterating, the condition for ending and condition for incrementing or continuing.

`foreach`

When we want our loop to run through the elements of an array, we can use a statement to simplify this: `foreach`.

```
<?php

$a = array (1, 2, 3, 17);

foreach ($a as $v
{
 print "Value: $v.\n";
}
?>
```

As we can see, in its simplest form, `foreach` assigns a variable `v` to each of the values of an array a, one by one.

2.6. Functions

Another key point about PHP are functions. In PHP, functions may or may not receive parameters and can always return a value. Functions are used to give greater modularity to the code, thus avoiding code repetition, allowing us to re-use code in other projects, etc.

One function schema is as follows:

```
<?php
 examp function ($arg_1, $arg_2, ..., $arg_n)
 {
 // Code of the function
 return $return;
 }
?>
```

We can call the functions from the main code or from other functions:

```
<?php
      sum function ($a1, $a2)
      {
         $return=$a1+$a2;
         return $return;
      }

      summation function ($b1, $b2, $b3)
      {
         for($i=$b1;$i<$b2;$i++)
         {
           $res=sum($res,$b3);
         }
         return $res;
      }
      echo summation(1,3,2);
?>
```

The result of executing this program will be to print the number six.

In PHP, functions usually receive parameters by value, i.e. the variable passed as the parameter in the code called is not modified if the parameter of the function is modified. Nonetheless, we can pass parameters by reference (in a similar way to pointers in other programming languages):

```
<?php
 modifi function ($&a1, $a2)
 {
```

```
    $a1=0;
    $a2=0;
 }
 $b1=1;
 $b2=1;
 modifi($b1,$b2);
 echo $b1." ".$b2;
?>
```

In this case, the result of the program will be:

```
1 0
```

2.7. Using PHP for web applications

To use PHP as a web application development language, the first thing we need to do is know how PHP will interact with our web user. We can divide this interaction in two parts, displaying information to the user and collecting information from the latter.

2.7.1. Displaying information

PHP can display information to users in two ways: it can write current HTML pages, inserting only the PHP code we require in the middle of the HTML code. For example:

```
<HTML>
 <HEAD>
    <TITLE>Document title</TITLE>
 </HEAD>
<BODY>
 <h1>Header H1</h1>
 <?php $a=1; ?>
 <h1>Second header H1</h1>
 <?php $b=1; ?>
 </BODY>
</HTML>
```

Alternatively, we can use PHP to generate dynamic content. To do this, we need to use the PHP data output instructions, the most important being `echo`.

```
<HTML>
 <HEAD>
    <TITLE>Document title</TITLE>
 </HEAD>
<BODY>
 <h1>Header H1</h1>
```

```
 <?php echo "Content of<B>page</B>"; ?>
 <h1>Second header H1</h1>
 </BODY>
</HTML>
```

2.7.2. Collecting user information

To collect user information, we can use HTML forms, using our PHP programs as the `ACTION` of these forms. Because PHP was designed to create web applications, access to the values entered by users in the form fields is extremely easy in PHP as it defines an array called `REQUEST` accessible with the name of the field as the index and which contains the value inside the latter when the PHP program is executed.

If we have this form:

```
<HTML>
   <HEAD>
     <TITLE>Document title</TITLE>
   </HEAD>
<BODY>
   <FORM ACTION="program.php" METHOD=GET>
   Type in the name: <INPUT TYPE=TEXT NAME="name">
   <INPUT TYPE=submit>
   </FORM>
   </BODY>
</HTML>
```

And we define the following PHP program as `program.php` in order to respond to the form:

```
<HTML>
 <HEAD>
    <TITLE>Document title</TITLE>
 </HEAD>
 <BODY>
 <?php
 echo "Hello".$REQUEST["name"];
 ?>
 </BODY>
</HTML>
```

This program will pick up the name entered by the user and display it to us on the screen.

2.8. String functions

PHP has a very interesting series of functions for working with text strings. Some of the most important of these are:

`strlen` Returns the length of a string.

`explode` Divides a string with a separating character and returns an array with each of the parts of the string.

`implode` Does the opposite to `explode` by joining several strings of an array with a joining character.

`strcmp` Compares two strings at binary level.

`strtolower` Converts a string to lower-case.

`strtoupper` Converts a string to upper-case.

`chop` Deletes the last character of a string, useful for deleting line breaks or trailing white spaces.

`strpos` Searches inside a string for another specified string and returns its position.

`str_replace` Replaces an appearance of a substring inside a string with another substring.

The following example shows how some of these functions work:

```
<?php
 $string1 = "hello";
 $string2 = "pear,apple,strawberry";

 $length = str_len($string1); //length=4

 $parts = explode(",",$string2);
 //generates the array $parts with $parts[0]="pear",
 //$parts[1]="apple"; and $parts[2]="strawberry";

 $chop = chop($string); // chop deletes the "a"

 $string3 = str_replace(",",";",$otherstring);
 //$string3 contains: pear-apple-strawberry
 //We change the , for -
?>
```

2.9. File access

PHP offers a wide range of methods for accessing files. Here, we will look at the most practical and straightforward of these, ideal if the files we are accessing are small.

The code we will use is:

```
<?php
 $file = file("input.txt");
 $linnum = count($file);

 for($i=0; $i < $linnum; $i++)
 {
     echo $file[$i];
 }
?>
```

In this example, we read a file called `input.txt` and display it as output. The first step is to declare the `file` variable, which will generate an array in which PHP will place all of the lines in the file. For this we will use the library called `file`. The next step involves finding out how many elements are in `file`. To do this, we will use the `count` function, which returns the size of an array - in this case, the array we generated on reading the file. Lastly, we can write a loop that will run the array, processing each line of the file.

PHP offers many more file-processing functions. For example, we have the `fopen` function, which allows us to open files or resources without fully reading them in memory. It can open files as follows:

```
 <?php
 $resource = fopen ("input.txt", "r");
 $resource = fopen ("output.gif", "wb");
 $resource = fopen ("http://www.uoc.edu/", "r");
 $resource = fopen ("ftp://user:password@uoc.edu/output.txt", "w");
?>
```

Here we can see how to open a file for reading (`''r''`), writing in binary (`''wb''`), a web page to read it as though it were a file and a file via FTP to write it, respectively.

2.10. Database access

PHP offers methods for accessing a large number of database systems (mySQL, PostgreSQL, Oracle, ODBC, etc). This feature is essential in the development of complex web applications.

2.10.1. Access to mySQL from PHP

mySQL is one of the most popular database systems for the development of light web applications because of its high performance when working with simple databases. Many query web applications, etc. are developed with the PHP-mySQL tandem. Hence, PHP's mySQL access API is highly developed.

We will now look at an example of access to the database from PHP to show how easily we can use databases in our web applications:

```
<?php

 $connection=mysql_connect($server,$user,$password);
 if(!$connection).
 {
   exit();
 }
 if(!(mysql_select_db($database,$connection)))
 {
   exit();
 }
 $query=mysql_query(
   "selectname,telephonefromcontactsorderbyname",
   $connection);
 while($row = mysql_fetch_array($query))
 {
   $name = $row["name"];
   $telephone = $row["telephone"];

   echo "$name: $telephone\n<br>";
 }
 mysql_free_result($query);
 mysql_close($connection);
?>
```

We can see that the first step in accessing the database is to open a connection with it. For this we will need the address of the computer containing the database, the user with whom we will connect and the word to access the database. Once connected to the mySQL server, we will need to select one of the multiple databases that the server we want to work with can have. Following this connection sequence, we will have the mySQL connection data in the connection variable. We must pass this variable to all PHP functions that access the database. This means that we can have a number of connections to different databases open at the same time and work simultaneously with them.

The next step will be to execute a database query statement in our database language, SQL in this case. For this we will use the PHP function called `mysql_query`, which will return the result of the query, which we will then save in the `query` variable. This specific `query` enumerates the content of a table in the database called `agenda`, which contains two columns called `name` and `telephone`.

We can then execute a loop that will run through all of the records to return our `query` to the database, accessing them one by one in order to display the results.

After completing access to the database, we need to free up the memory and resources used in the query. To do this, we will use the `mysql_free_result` function and then close the connection to mySQL.

2.10.2. Access to PostgreSQL from PHP

We can access databases on PostgreSQL servers in much the same way as we access mySQL. As in the previous section, we will use a code to enumerate the content of our table called `agenda`.

```
<?php
 //connecting to the database
 //$connection = pg_connect("dbname=".$database);

 // connecting to the server database port "5432"
 //$connection = pg_connect(
 // "host=$port server=5432 dbname=$database");
 // connecting to the server database port "5432"
 //with user and password
 $connection = pg_connect("host=$port server=5432 ".
   "dbname=$userdatabase=$userpassword=$password")
   or die "Does not connect";

 $result = pg_query($connection,
   "select name, telephone from contacts order by name");

 while($row = pg_fetch_array($result))
 {
   $name = $row["name"];
   $telephone = $row["telephone"];
   echo "$name: $telephone\n<br>";
 }
 pg_close($dbconn);
?>
```

As we can see if we compare this example with the mySQL one above, the two codes are very similar. Yet, despite the similarities, PostgreSQL's API for PHP has some differences with regards mySQL. On the one hand, it offers greater flexibility for connections and, on the other, it provides support for working with large objects, etc., demonstrating that PostgreSQL is more powerful than mySQL. A controversial point of PostgreSQL's API is that it totally isolates us from the PostgreSQL system of transactions, which is perfectly acceptable in most situations but we may sometimes want greater control over these.

2.11. More information

One of the strong points and, indeed, keys to the success of PHP as a web application programming language are the many libraries, modules, etc., that have been developed for it. PHP is offering an increasing number of APIs, functions, modules, classes (remember that PHP is gradually becoming an object-oriented programming language), allowing us to operate with the increasing complexity of web applications. This diversity of support includes:

- Session control.
- User identity control.
- HTML templates.
- Shopping carts.
- Dynamic HTML creation.
- Dynamic creation of images.
- Handling of *cookies*.
- File transfer.
- Handling of XML, XSLT, etc.
- Multiple communication protocols: HTTP, FTP, etc.
- Creation of PDF files.
- Access to LDAP directories.
- Interfaces with a wide range of databases: Oracle, Sybase, etc.
- Regular expressions.
- Network equipment handling: SNMP.
- Web services: XMLRPC, SOAP.
- Handling of Flash content.

Besides its usefulness as a web application programming language, PHP is being increasingly employed as a general purpose programming language, including in its arsenal of functions interfaces with the libraries of more common graphic interfaces (Win32 or GTK, among others), direct access to operating system functions, etc.

Hence, if you want to use PHP to develop a project, we strongly recommend you to visit the project's website (http://www.php.net), since you may well find a number of tools to make the job a whole lot easier. We also have PEAR, a PHP repository that will provide us with most of the tools we could need.

3. Java servlets and JSP

3.1. Introduction to Java servlets

Java servlets are the Java technology proposal for the development of web applications. A servlet is a program that runs on a web server and builds a web page that is returned to the user. This page is built dynamically and may contain information from databases, be a response to data entered by the user, etc.

Java servlets offer a series of advantages over CGIs, the traditional method of web application development. These are more portable, more powerful, much more efficient, more user-friendly, more scalable, etc.

3.1.1. Efficiency

With the traditional CGI model, each request that reaches the server triggers the execution of a new process. If the lifetime of the CGI (the time it takes to be executed) is short, the instantiation time (the time taken to launch a process) can exceed that of execution. With the servlets model, the Virtual Java Machine, the environment from which they are run, starts up when the server starts and remains in operation throughout execution of the same. To deal with each request, instead of launching a new process, a *thread*, a light-weight Java process, is started which is much faster (it is actually instantaneous). Moreover, if we have *x* simultaneous requests from a CGI, we will have `x` simultaneous processes in memory, thus consuming *x* times the space of a CGI (which, if interpreted, is usually the case, consumes *x* times the interpreter). With servlets, there is a certain number of *threads*, but there is only one copy of the Virtual Machine and its classes.

The servlets standard offers additional alternatives to CGIs for optimisation: `caches` of previous calculations, `pools` of database connections, etc.

3.1.2. Ease of use

The servlets standard provides a wonderful web application development infrastructure, with methods for the automatic analysis and decoding of HTML form data, access to HTTP request headers, handling of `cookies`, monitoring, control and management of sessions, among many other features.

3.1.3. Power

Java servlets can be used for many things that are difficult or impossible to do with traditional CGIs. Servlets can share data with each other, which means that they can share data, database connections etc. They can also maintain information request after request, facilitating tasks such as the monitoring of user sessions, etc.

3.1.4. Portability

Servlets are written in Java and use a well documented, standard API. As a result, servlets can be run on all platforms with Java servlet support without the need for recompilation, modification etc., regardless of the platform (Apache, iPlanet, IIS, etc.) and operating system, architecture *hardware*, etc.

3.2. Introduction to Java Server Pages or JSP

Java Server Pages (JSP) are a technology that allows us to mix static HTML with HTML generated dynamically using Java code embedded on pages. When we program web applications with CGIs, the bulk of the page generated by the CGIs is static and does not vary from execution to execution. The variable part of the page is truly dynamic and very small. Both CGIs and servlets require us to generate the page fully from our program code, which makes it more difficult for maintenance, graphic design, code comprehension, etc. With JSP, however, we can easily create pages.

Example

The parts of the page that do not vary from execution to execution are headers, menus, decorations, etc.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
    <HEAD>
      <TITLE>Store. Welcome.</TITLE>
    </HEAD>
 <BODY>
    <H1>Welcome to our store</H1>
    <SMALL>Welcome,
    < % out.println(Tools.readNameOfCookie(request)); %>
    </SMALL>
 </BODY>
</HTML>
```

As this example shows, a JSP page is nothing more than a HTML page where the special tags < % and %> allow us to include Java code.

This gives us a series of obvious advantages: firstly, we have practically the same advantages as we do when using Java servlets; in fact, JSP servers "translate" these to servlets before executing them. Secondly, JSPs offer

considerable simplicity and ease of development. It is much easier to write the example page than to write a servlet or CGI that prints each of the lines in the above page.

However, this simplicity is also one of the disadvantages of JSP. With complex applications containing numerous calculations, database accesses, etc., JSP syntax embedded inside HTML becomes tedious. Thus, JSPs and servlets do not usually compete, but rather they complement one another since the standards include capabilities for communication between them.

3.3. The servlets/JSP server

To use both servlets and JSP on our web server, we generally need to complement it with a servlets/JSP server (usually called a servlets container). There are many free software and proprietary containers. Sun, the inventors of Java, keep an updated list of servlet containers at:

`http://java.sun.com/products/servlet/industry.html`

- Apache Tomcat. Tomcat is the official implementation of reference for servlet and JSP specifications after versions 2.2 and 1.1, respectively. Tomcat is a very robust, highly efficient product and one of the most powerful servlet containers available. Its only weakness is that it is complicated to configure because there are many options to choose from. For more details, visit the official Tomcat website: `http://jakarta.apache.org/`.

- JavaServer Web Development Kit (JSWDK). JSWDK was the official reference implementation for specifications Servlet 2.1 and JSP 1.0. It was used as a small server to test servlets and JSP pages in development. However, it has now been abandoned in favour of Tomcat. Its website is: `http://java.sun.com/products/servlet/download.html`.

- Enhydra. Enhydra is an applications server whose many functionalities include a very powerful servlet/JSP container. Enhydra (`http://www.enhydra.org`) is a very powerful tool for developing web services and applications, including tools for the control of databases, templates, etc.

- Jetty. This is a very lightweight web server/servlet container written entirely in Java that supports the Servlet 2.3 and JSP 1.2 specifications. It is the ideal server for development because it is small and takes up little memory. Its web page is: `http://jetty.mortbay.org/jetty/index.html`.

3.4. A simple servlet

The following example shows the basic structure of a simple servlet that handles HTTP GET requests (servlets can also handle POST requests).

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

public class BasicServlet extends HttpServlet
{
 public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // We can use request to access the data of the
    // HTTP request.
    // We can use response to modify the HTTP response
    // that the servlet will generate.

    PrintWriter out = response.getWriter();
    // We can use out to return data to the user
    out.println("¡Hello!\n");
 }
}
```

To write a servlet, we must write a Java class that extends (by inheritance) the `HttpServlet` class (or the most generic `servlet class`) and overwrites the `service` method or one of the more specific request methods (`doGet`, `doPost` etc).

Service methods (`service`, `doPost`, `doGet`, etc.) have two arguments: a `HttpServletRequest` and a `HttpServletResponse`.

The `HttpServletRequest` gives us the methods for reading incoming information such as the data from a HTML form (FORM), HTTP request headers or the *cookies* of the request, etc. In contrast, `HttpServletResponse` has methods for specifying the HTTP response codes (200, 404, etc.), response headers (`Content-Type`, `Set-Cookie` etc). Most importantly, they allow us to obtain a `PrintWriter` (a Java class representing an output "file") used to generate the output data that will be returned to the client. For simple servlets, the bulk of the code is used to work with this `PrintWriter` in `println` statements that generate the desired page.

3.5. Compiling and executing servlets

The servlet compilation process is very similar regardless of the web server or servlet container used. If using Sun's Java development programming, the official JDK, we need to make sure that our `CLASSPATH`, the list of libraries and directories where the classes we use in our programs are searched, contains the Java servlets API libraries. The name of this library varies from version to version of the Java API but it is usually: `servlet-version.jar`. Once the servlets library is in our `CLASSPATH`, the servlet compilation process is as follows:

```
javac BasicServlet.java
```

We must locate the resulting `class` file in the directory that our servlet container requires to execute the servlet. To then test it, we need to direct the browser to the URL of our servlet, formed, on the one hand, by the directory where our servlet container displays the servlets (for example, `/servlets`) and, on the other, by the name of the servlet.

For example, in JWS, Sun's test server, `servlets` are located in a servlets subdirectory of the JWS installation directory and the URL is formed thus:

`http://server/servlet/BasicServlet`

In Tomcat, servlets are located in a directory indicating the web application under development, in the `WEB-INF` subdirectory, inside the subdirectory `classes`. Then, if the web application were called `test` for example, the resulting URL would be:

`http://server/test/servlets/BasicServlet`

3.6. Generating content from servlets

As we have seen, the API gives us a class called `PrintWriter` to which we can send all of our results. However, this is not enough to make our servlet return HTML to the client.

The first step for building a servlet that returns HTML to the client is to tell the servlet container that the return of our servlet is HTML. Remember that HTTP includes the transfer of multiple data types by sending the MIME type marker tag: `Content-Type`. To do this, we have a method for indicating the type returned, `setContentType`. So, before any interaction with the response, we need to mark the content type.

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

```
public class Helloweb extends HttpServlet
{
 public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException
 {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println(
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n" +
      "<HTML>\n" +
      "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
      "<BODY>\n" +
      "<H1>Hello web</H1>\n" +
      "</BODY></HTML>");
 }
}
```

As we can see, generating the result in HTML is a very tedious task, especially if we consider that part of this HTML does not change from servlet to servlet or execution to execution. The solution to this type of problem is to use JSP instead of servlets. However, if you really must use servlets, there are a number of time-saving tricks. The main solution is to declare methods that really return these common HTML parts: the `DOCTYPE` line, t e header and even a common header and footer for the company's whole website.

To do this, we need to build a class containing a series of utilities that we can use in our web application project.

```
public class Utilities
{
 public static final String DOCTYPE =
   "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD"+
   " HTML 4.0 Transitional//EN\">";

 public static String titleHeader(Title string) {
   return(DOCTYPE + "\n" +
     "<HTML>\n" +
     "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
 }
 // Here, we will add some utilities
}
```

3.7. Handling form data

Obtaining data sent by a user from a form is one of the most complex and monotonous tasks of CGI programming. Since we have two methods for passing values, `GET` and `POST`, which behave differently, we need to develop two methods to read these values. We must also analyse, *parse* and decode the strings containing coded values and variables.

One of the advantages of using servlets is that the servlets API solves all of these problems. This task is automatic and the values are made available to the servlet through the `getParameter` method of the class called `HttpServletRequest`. This parameter passing system is independent of the method used by the form to pass parameters to the servlet (GET or POST). There are also other methods to help us collect the parameters sent by the form. Firstly, we have a version of `getParameter` called `getParameters` that we need to use if the parameter we are looking for can have more than one value. We also have `getParameterNames`, which returns the name of the parameters passed.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class BasicServlet extends HttpServlet
{
 public void doGet(HttpServletRequest request,
                 HttpServletResponse response)
      throws ServletException, IOException
 {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String tit= "Reading 2 Parameters";
    out.println(Utilities.titleHeader(tit) +
       "<BODY>\n" +
       "<H1 ALIGN=CENTER>" + tit + "</H1>\n" +
       "<UL>\n" +
       " <LI>param1: "
       + request.getParameter("param1") + "\n" +
       " <LI>param2: "
       + request.getParameter("param2") + "\n" +
       "</UL>\n" +
       "</BODY></HTML>");
 }

 public void doPost( HttpServletRequest request,
                     HttpServletResponse response)
       throws ServletException, IOException
```

```
 {
   doGet(request, response);
 }
}
```

This example of a servlet reads two parameters called `param1`, `param2` and displays their values in a HTML list. We can see how `getParameter` is used and how, by making `doPost` call `doGet`, the application is made to respond to the two methods. If required, we have methods for reading the standard input, as in CGI programming.

We will now look at a more complex example to illustrate the full potential of the servlets API. This example receives data from a form, searches for the names of the parameters and prints them, indicating those with the value of zero and those with multiple values.

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

public class Parameters extends HttpServlet
{
 public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
       throws ServletException, IOException
 {
       response.setContentType("text/html");

       PrintWriter out = response.getWriter();
       String tit= "Reading Parameters";

       out.println(Utilities.titleHeader(tit) +
              "<body BGCOLOR=\"#FDF5E6\">\n" +
              "<H1 ALIGN=CENTER>" + tit + "</H1>\n" +
              "<TABLE BORDER=1 ALIGN=CENTER>\n" +
              "<TR BGCOLOR=\"#FFAD00\">\n" +
              "<TH>Parameter Name<TH>Parameter(s) Value");

       // Reading the names of the parameters
       Enumeration params = request.getParameterNames();

       // Going through the names array
       while(params.hasMoreElements())
       {
         // Reading the name
         String param = (String)params.nextElement();
```

```
        // Printing the name
        out.println("<TR><TD>" + paramName + "\n<TD>");

        // Reading the values array of the parameter
        String[] values = request.getParameterValues(param);

        if (values.length == 1)
        {
          // Only one empty value
          String value = values[0];

          // Empty value.
          if (value.length() == 0)
        out.print("Empty");
     else
     out.print(value);
    }
    else
    {
      // Multiple values
       out.println("<UL>");
      for(int i=0; i<values.length; i++)
  {
      out.println("<LI>" + values[i]);
  }
      out.println("</UL>");
  }
  }
      out.println("</TABLE>\n</BODY>\n</HTML>");
 }

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

We first look for the names of all parameters using the method `getParameterNames`. This returns an `enumeration`. We then use the standard method to run `enumeration` (using `hasMoreElements` to determine when to stop and `nextElement` to obtain each input). Since `nextElement` returns an `object` object, we convert the result to `String` and we use them with `getParameterValues` to obtain a `String`. If this array only has one entry and contains only one empty `String`, the parameter has

no values and the servlet will generate an "empty" entry in italics. If the array contains more than one entry, the parameter has multiple values, which are displayed in an unsorted list. Otherwise, the only value is displayed.

This is an HTML form that will be used to test the servlet, as it sends a group of parameters to it. Since the form contains a PASSWORD type field, we will use the POST method to send the values.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML> <HEAD>
 <TITLE>Form with POST</TITLE>
</head>

<BODY BGCOLOR="#FDF5E6">
 <H1 ALIGN="CENTER">Form with POST</H1>

 <FORM ACTION="/examples/servlets/Parameters" METHOD="POST">
    Code: <INPUT TYPE="TEXT" NAME="code"><BR>
    Quantity: <INPUT TYPE="TEXT" NAME="quantity"><BR>
    Price: <INPUT TYPE="TEXT" NAME="price" VALUE="\$"><BR>
    <HR>
    Name:
    <INPUT TYPE="TEXT" NAME="Name"><BR>
    Surname:
    <INPUT TYPE="TEXT" NAME="Surname"><br>

    Address:
    <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><br>

    Credit card:<BR>
       <INPUT TYPE="RADIO" NAME="cred"
             VALUE="Visa">Visa<BR>
       <INPUT TYPE="RADIO" NAME="cred"
             VALUE="MasterCard">MasterCard<BR>
       <INPUT TYPE="RADIO" NAME="cred"
             VALUE="Amex">American Express<BR>
       <INPUT TYPE="RADIO" NAME="cred"
             VALUE="Maestro">Maestro<BR>
Card number:
<INPUT TYPE="PASSWORD" NAME="cardno"><br>

 Re-enter card number:
 <INPUT TYPE="PASSWORD" NAME="cardno"><BR><BR>
 <CENTER>
         <INPUT TYPE="SUBMIT" VALUE="Place order">
 </CENTER>
</FORM>
```

```
</BODY>
</HTML>
```

3.8. The HTTP request: HttpRequest

When an HTTP client (the browser) sends a request, it can send a specific number of optional headers, except for Content-Length, which is required in POST requests. These headers provide additional information to the web server, which can use them to adapt its response to suit the browser request.

Some of the most common and useful headers are:

- Accept. The MIME types preferred by the browser.
- Accept-Charset. The character set accepted by the browser.
- Accept-Encoding. The types of data encoding accepted by the browser. For example, it can indicate that the browser accepts compressed pages, etc.
- Accept-Language. The language preferred by the browser.
- Authorization. Authorisation information, usually in response to a server request.
- Cookie. XML `cookies` stored in the browser that correspond to the server.
- Host. Server and port of the original request.
- If-Modified-Since. Send only if it has been modified since the specified date.
- Referrer. The URL of the page containing the link followed by the user to obtain the current page.
- User-Agent. Type and brand of browser, useful for adapting the response to specific browsers.

To read the headers, we simply need to call the method `getHeader` of `HttpServletRequest`. This will return a `String`, if the indicated header was sent in the request, and `null` if it was not.

Some header fields are used so often that they have their own methods. The `getCookies` method is used to access *cookies* sent with the HTTP request, analysing and storing them in a `cookie`. The `getAuthType` and `getRemoteUser` methods allow access to each of the components of the

`Authorization` field in the header. The `getDateHeader` and `getIntHeader` methods read the specific header and convert it to the values `Date` and `int`, respectively.

Instead of searching for a specific header, we can use `getHeaderNames` to obtain an `enumeration` of all the header names of a specific request. If this is the case, we can run through this list of headers, etc.

Lastly, as well as accessing the header fields of the request, we can obtain information about the request itself. The `getMethod` returns the method used for the request (usually `GET` or `POST`, but HTTP has other, less common methods, such as `HEAD`, `PUT` and `DELETE`). The `getRequestURI` method returns the `URI` (the part of the `URL` that appears after the name of the server and port but before the form data). The `getRequestProtocol` method returns the protocol used, generally `"HTTP/1.0"` or `"HTTP/1.1"`.

3.9. Additional request information

Besides the headers of the HTTP request, we can obtain a series of values that will provide us with further information about the request. Some of these values are available for CGI programming as environment variables. They are all available as `HttpRequest`.

getAuthType (). If an `Authorization` header s supplied, this is the specified schema (`basic` or `digest`). CGI variable: `AUTH_TYPE`.

getContentLength (). Only for POST requests, the number of bytes sent.

getContentType (). The MIME type of the attached data, if specified. CGI variable: `CONTENT_TYPE`.

getPathInfo (). Information on the *path* attached to the URL. CGI variable: `PATH_INFO`.

getQueryString (). For GET requests; these are the data sent as a single string with encoded values. They are not generally used in servlets, since direct access to the decoded parameters is available. CGI variable: `QUERY_STRING`.

getRemoteAddr (). The IP address of the client. CGI variable: `REMOTE_ ADDR`.

getRemoteUser (). If an Authorization header is supplied, the user part. CGI variable: `REMOTE_USER`.

getMethod (). The request type is normally GET or POST, but it can also be HEAD, PUT, DELETE, OPTIONS or TRACE. CGI variable: `REQUEST_METHOD`.

3.10. Status and response codes

When a browser's web request is processed, the response usually contains a numerical code that tells the browser whether the request has been fulfilled and, where applicable, the reasons why this is not the case. It also includes some headers to give the browser further information about the response. Servlets can be used to indicate the HTTP return code and the value of some of these headers. This means that we can redirect the user to another page, indicate the type of response content, request a password from the user, etc.

3.10.1. Status codes

To return a specific status code, our servlets can use the `setStatus`. which tells the web server and servlet container the status that they should return to the client. In the `HttpServletResponse` class, the servlets API provides a table of constants to facilitate the use of response codes. These constants have names that are easy to remember and use.

For example, the constant for code 404 (qualified in standard HTTP as *not found*), is SC_NOT_FOUND.

If the code we return is not the default one (200, SC OK), we will need to call `setStatus` before using `PrintWriter` to return the client content. We can also use `setStatus` to return error codes for two more specialised methods: `sendError` to return errors (code 404), which allows us to add a HTML message to the numerical code, and `sendRedirect` (code 302), which is used to specify the address to which the client is redirected.

3.10.2. Return headers

Besides including a numerical code when responding to the http request, the server can add a series of values in response headers. These headers tell the browser about the expiry of the information sent (`Expires`), that it must refresh the information after a specific time (`Refresh`), etc. We can modify the value of these headers or add new ones from our servlets. To do so, we can use the `setHeader` method of the class called `HttpServletResponse` class, hich allows us to assign random values to the headers we return to the client. As with return codes, we must select the headers before sending a value to the client. There are two auxiliary methods for `setHeader` for times when we want to send headers containing dates or integers. These methods, `setDateHeader` and `setIntHeader`, do not rule out the need for converting dates and integers to `String`, the parameter accepted by `setHeader`.

There are also specialised methods for some of the more common headers:

`SetContentType`. Provides a value for the `Content-Type` header and must be used in most servlets.

`SetContentLength`. Allows us to assign a value to the `Content-Length`.

`AddCookie`. Assigns a *cookie* to the response.

`SendRedirect`. As well as assigning status code 302, as we saw, it assigns the address to which the user is redirected in the header `Location`.

3.11. Session monitoring

HTTP is a stateless protocol, which means that each request is totally independent of the previous one. This means that we cannot link two consecutive requests, which is disastrous if we want to use the web for something more than simply viewing documents. If we are developing an e-commerce application such as an on-line store, we need control over the products that our client has selected to ensure that we have the correct shopping list when the client reaches the order page. How can we obtain the list of objects selected for purchase when this screen is reached?

There are three possible solutions to this problem:

1) Use *cookies*. *Cookies* are small pieces of information sent by the server to the browser, which the latter resends every time it accesses the website. Despite excellent support from *cookies*, using this technique to monitor a session is still an arduous task:
- Control the *cookie* containing the session identifier.
- Control expiry of the latter.
- Associate the contents of the *cookie* with information from a session.

2) Rewrite the URL. We can use the URL to add further information to identify the session. This solution has the advantage that it works with browsers that have no *cookies* support or where it is disabled. However, it is still a tedious method:
- We need to ensure that all URLs reaching the user have the right session information.

- It causes problems for users trying to add addresses to their *bookmarks*, because these contain expired session information.

3) Hidden fields in forms. We can use the `HIDDEN` fields of HTML forms to spread information in our interest. Clearly, this suffers from the same problems as the above solutions.

Fortunately, the servlets API has a solution to this problem. Servlets have a high-level API, `HttpSession`, for session management, which is carried out using *cookies* and URL rewriting. This API isolates the author from the servlets of the details of session management.

3.11.1. Obtaining the session associated with the request

To obtain the session associated with the HTTP request in course, we can use a `getSession` method of the class called `HttpServletRequest`. If a session exists, this method will return a `HttpSession`. If it does not exist, it will return `null`. We can call `getSession` using an additional parameter that will create the session automatically if it does not exist.

```
HttpSession session = request.getSession(true);
```

3.11.2. Accessing the associated information

The `HttpSession` objects representing the information associated with a session allow us to store a series of named values inside. To read these values, we can use `getAttribute`, and to modify them, we have `setAttribute`.

Note

In versions prior to 2.2 of the servlets API, the functions for accessing information were: `getValue` and `setValue`.

One schema for accessing this session data might be:

```
HttpSession session = request.getSession(true);

 Languagestring=(String)session.getAttribute("language");
 if (language == null)
 {
   language=new String("Spanish");
   response.setAttribute("language",language);
 }
 // we can now display the data in the language
 // preferred by the user
```

There are methods for accessing the list of attributes saved in the session, such as `getAttributeNames`, which returns an `enumeration`, similar to the `getHeaders` and `getParameterNames` methods of `HttpServletRequest`.

Note

In versions prior to 2.2 of the servlets API, the list of value names function was `getValueNames`.

There are also some useful functions for accessing session information:

getId returns a unique identifier generated for each session.

isNew returns *true* if the client has never seen the session because it has just been created.

getCreationTime returns the time in milliseconds since 1970, the year in which the session was created.

getLastAccessedTime returns the time in milliseconds since 1970, the year in which the session was sent to the client for the last time.

3.12. Java Server Pages: JSP

Java Server Pages, or JSP, are a HTML extension developed by Sun used to embed Java instructions (*scriptlets*) in the HTML code. This simplifies matters when it comes to designing dynamic websites. We can use any of the many HTML editors to create our web or we can leave this to the designers, focusing instead on the development of the Java code that will generate the dynamic parts of the page so that we can subsequently embed this code in the page.

An example of a basic JSP page that will introduce us to some of the main concepts of the standard is as follows:

```
<HTML>
 <BODY>
 <H1>Welcome. Date: < %= date %> </h1>
 <B>
 < % if(name==null)
      out.println("New user");
   else
      out.println("Welcome back");
 %>
 </b>
 </BODY>
</HTML>
```

JSP pages normally have the extension `.jsp` and are located in the same directory as HTML files. As we can see, a `.jsp` page is simply a HTML page in which we embed pieces of Java code, delimited by `< %` and `%>`. Constructions delimited by `< %` and `%>` can be of three types:

- *Script* elements allowing us to enter a code that will form part of servlet resulting from translation of the page.
- Directives, used to tell the servlet container how we want the servlet to be generated.
- Actions allow us to specify components that should be used.

When the server/servlet container processes a JSP page, it converts this into a servlet in which all of the HTML that we have entered in the JSP page is printed on output and subsequently used for compiling this servlet and passing the

request to it. This conversion/compilation step is generally only carried out the first time we access the page or if the JSP file has been modified since the last time it was compiled.

3.12.1. *Script elements*

Script elements allow us to insert Java code inside a servlet produced by the compilation of our JSP page. There are three options when it comes to inserting code:

- Expressions of the type `< %= expression %>` which are evaluated and inserted in the output.
- *Scriptlets* of the type `< % code %>` that are inserted within the servlet's Service method.
- Declarations of the type `< %! code %>` that are inserted in the body of the servlet class, outside any existing method.

Expressions

JSP expressions are used to insert a Java value directly in the output. Their syntax is:

```
< %= expression %>
```

The expression is evaluated and produces a result that is converted into a string, which is inserted in the resulting page. The evaluation is carried out in execution time, when the page is requested. Hence, expressions can access HTTP request data. For example,

```
< %= request.getRemoteUser() > logged on on
< %= new java.util.Date() >
```

This code will display the remote user (if authenticated) and the date on which the page was requested.

We can see in our example that we are using a variable, `request`, which represents the HTTP request. This predefined variable belongs to a series of predefined variables that we can use:

- `request`: the `HttpServletRequest`
- `response`: the `HttpServletResponse`
- `session`: the `HttpSession` associated with `request` (if it exists)

- `out`: the `PrintWriter` used to send the output to the client

There is an alternative syntax for entering expressions. This syntax was introduced to make JSP compatible with XML editors, *parsers*, etc. It is based on the concept of *tagActions*. The syntax for an expression is:

```
<jsp:expression> expression</jsp:expression>
```

Scriptlets

XML *scriptlets* are used to insert random Java code in the servlet that will result from compilation of the JSP page. A *scriptlet* looks like this:

```
< % code %>
```

In a *scriptlet* we can access the same predefined variables as in an expression. For example:

```
< %
 String user = request.getRemoteUser();
 out.println("User: " + user);
%>
```

XML *scriptlets* are inserted in the resulting servlet as they are written, while the HTML code entered is converted into `println`. This means that we can create constructions such as:

```
<% if (obtainTemperature() < 20) { %>
       <B>Wrap up! It's cold! </B>
<% } else { %>
       <B>Have a nice day!</B>
< % } %>
```

In this example, we see that the Java code blocks can affect and include the HTML defined on the JSP pages. Once the page has been compiled and the servlet generated, the above code will look something like this:

```
if (obtainTemperature() < 20) {
    out.println("<B>Wrap up! It's cold! </B>");
} else {
    out.println("<B>Have a nice day!</B>");
}
```

The XML equivalent for *scriptlets* is:

```
<jsp:scriptlet> code </jsp:scriptlet>
```

JSP declarations

Declarations are used to define methods or fields that are subsequently inserted in the servlet outside the `service`. They look similar to this:

```
< %! code %>
```

Declarations do not generate output. As a result, they are usually used to define global variables, etc. For example, the following code adds a counter to our page:

```
< %! private int visits = 1; %>
Visits to the page while server is running:
< %= visits++ %>
```

This counter is restored to one each time the servlet container is restarted or each time we modify the servlet or JSP file (which requires the server to reload it). The equivalent to declarations for XML is:

```
<jsp:declaration> code </jsp:declaration>
```

3.12.2. JSP directives

Directives affect the general structure of the servlet class. They look like this:

```
< %@ attribute directive1="value1"
                 attribute2="value2"
                 ...
                 %>
```

There are three main directives:

`page` allowing us to modify compilation of the JSP page to the servlet.

Example

For example, we can import classes, modify the servlet class, etc.

`include` which allows us to insert another file in the resulting servlet (this is inserted when translating JSP to servlet).

`taglib` which is used to indicate which tag libraries we wish to use. JSP allows us to define our own tag libraries.

The `page directive`

We can define the following attributes using the page directive, which will modify translation of JSP to servlet:

- `import="package.class"` or `import="package.class1, ... ,package.classN". Import` allows us to specify the packets and classes

that need to be imported by Java to compile the resulting servlet. This attribute can appear several times in each JSP. For example:

```
< %@ page import="java.util.*" %>
< %@ page import="edu.uoc.campus.*" %>
```

- `contentType="MIME-Type"` or `contentType="MIME-Type; charset=Character-Set"` This directive is used to specify the resulting MIME type of the page. The default value is `text/html`. For example:

```
< %@ page contentType="text/plain" %>
```

This is equivalent to using the *scriptlet*:

```
< % response.setContentType("text/plain"); %>
```

- `isThreadSafe="true|false"`. A `true` value (the default value) indicates that the resulting servlet will be a normal servlet, in which multiple requests can be processed simultaneously, assuming that the instance variables shared between *threads* will be synchronised by the author. A `false` value indicates that the servlet must implement a `SingleThreadModel`.

- `session="true|false"`. A `true` value (the default value) indicates that there must be a predefined `session` variable (of the type `Httpsession`) with the session or, if there is no session, one must be created. A `false` value indicates that sessions will not be used and attempts to access them will result in errors when it comes to translating to servlet.

- `extends="package.class"`. This indicates that the servlet generated must extend a different superclass. It must be used with extreme caution, since the servlet container we use may require the use of a specific superclass.

- `errorPage="URL"`. Specifies which JSP will be processed if an exception is launched (an object of the type `Throwable`) and it is not captured on the current page.

- `isErrorPage="true|false"`. Indicates whether the current page is an error processing page.

The equivalent XML syntax is:

```
<jsp:directive.Directive attribute=value />
```

For example, the following two lines are equivalents:

```
< %@ page import="java.util.*" %>
<jsp:directive.page import="java.util.*" />
```

The `include` directive

The `include` directive is used to include files in the JSP page when translated to servlet. The syntax is as follows:

```
< %@ include file="file to be included" %>
```

The file to be included can be relative to the position of the JSP on the server, for example, `examples/example1.jsp` or absolute, for example, `/general/header.jsp`. and it can contain any JSP construction: html, scriptlets, directives, actions, etc.

The `include` directive can save us a lot of work because it allows us to write elements like the menus of our website on a single page, which means that we only need to include them in each JSP we use.

```
1.
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
 <HTML>
   <HEAD>
     <TITLE>Website</TITLE>
     <META NAME="author" CONTENT="carlesm@asic.udl.es">
     <META NAME="keywords" CONTENT="JSP, Servlets">
     <meta NAME="description" CONTENT="One page">
     <LINK REL=STYLESHEET HREF="style.css" TYPE="text/css">
   </HEAD>
   <body>

2.
<HR>
 <CENTER><small>&#169; Web developer, 2003. All
rights reserved</SMALL></center>

</BODY> </html>

3.
< %@ include file="/header.html" %>
 <!-- JSP page -->
   .
   .
< %@ include file="/footer.html" %>
```

In this example, we have three files: `header.html, footer.html` and a JSP page of the website, respectively. As we can see, having a fragment of the page content in separate files considerably simplifies the writing and maintenance of JSP pages.

One thing to bear in mind is that it is included when the JSP page is translated to servlet. If we change anything in the files included, we will need to force re-translation of the entire site. Although this may seem a problem, it is greatly compensated by the benefits gained by the efficiency of only having to include the files once.

If we want them to be included in each request, we have an alternative in the XML version of the directive:

```
<jsp:include file="/header.html">
 <!-- JSP page -->
   .
   .
<jsp:include file="/header.html">
```

In this case, the inclusion is made when the page is served. However, we cannot include any JSPs in the file we are going to include; it can only be in HTML.

3.12.3. Predefined variables

In JSPs, we have a group of defined variables to make code development easier.

- `request`. The `HttpServletRequest` object associated with the request. This allows access to the request parameters (through `getParameter`), the type of request and the HTTP headers (*cookies*, `referrer` etc).

- `response`. This is the `HttpServletResponse` object associated with the servlet response. Since the *stream* output object (the `out` variable defined later) has a *buffer*, we can select the status codes and response headers.

- `out`. This is the `PrintWriter` object used to send the output to the client.

- `session`. This is the `HttpSession` object associated with the request. Sessions are created automatically by default. This variable exists even if there is no reference session. The only exception is if we use the `session` attribute of the `page directive`.

- `application`. This is the `ServletContext` object obtained through `getServletConfig().getContext()`.

- `config`. The `ServletConfig` object for this page.

3.12.4. Actions

JSP actions use constructions with a valid XML syntax to control the behaviour of the servlet container. These actions are used to insert files dynamically, use JavaBeans components, resend another page to the user, etc.

`jsp:include`

This action is used to insert files into the page being generated. The syntax is:

```
<jsp:include page="relative URL" flush="true" />
```

Unlike the `include` directi e, which inserts the file when the JSP page is being converted to servlet, this action inserts the file when the page is requested. On the one hand, this results is less efficiency and means that the included page cannot contain JSP code. On the other hand, however, it increases flexibility because we can change the inserted files without having to recompile the pages.

Here is an example of a page that inserts a news file into a website. Each time we want to change the news, we simply need to change the file included. This is a job that can be left with the copywriters without the need to recompile the JSP files.

```
< %@ include file="/header.html" %>
Latest news:
<jsp:include page="news/news.html" />
< %@ include file="/footer.html" %>
```

`jsp:useBean`

This action can be used to load a JavaBean on the JSP page so that we can use it. It is a very useful capability because it allows us to make use of the reusability of Java classes. The simplest way of specifying the use of a bean is:

```
<jsp:useBean id="name" class="package.class" />
```

The meaning of this code is: it instances an object of the `class` specified by class and assigns it to the variable called `id`. We can also add a `scope` attribute, indicating that the `Bean` must be associated to more than one page.

Note

Remember JavaBeans. A property **X** of the type **Y** of a `Bean` means: a `getX ()` method that returns an object of the type **Y** and a `setX (Y)`.

Once we have the `Bean` instanced, we can access its properties. It is possible to do this from a *scriptlet* or with one of the following two actions: `jsp:setProperty` and `jsp:getProperty`.

We will describe these actions, `jsp:setProperty` and `jsp:getProperty`, in detail later. For now, you simply need to be aware that they have an attribute, `param` to specify which property we want.

Here is a small example of how Beans are used on JSP pages:

```
< %@ include file="/header.html" %>
Latest news:

 <jsp:useBean id="mess" class="MessageBean" />
 <jsp:setProperty name="mess"
                     property="text"
                     value="Hello" />
      <H1>Message: <I>
      <jsp:getProperty name="mess" property="text" />
      </I></h1>

< %@ include file="/footer.html" %>
```

Note

Location of beans.

To ensure correct loading of beans, we need to ensure that the servlet container will find them. To do this, check the documentation to find out where they should be placed.

The Bean code used for the example is as follows:

```
public class MessageBean
{
 private String text = "No text";
 public String getText()
 {
      return(text);
 }
 public void setText(String text)
 {
      this.text = text;
 }
}
```

The `jsp:useBean` action includes other facilities for working with Beans. If we wish to execute a specific code when the Bean is instanced (in other words, when it is first loaded), we can use the following construction:

```
<jsp:useBean ...>
 code
</jsp:useBean>
```

Note that `Bean` can be shared among different pages. However, not all uses of `jsp:useBean` result in the instantiation of a new object. For example:

```
<jsp:useBean id="mess" class="MessageBean" >
 <jsp:setProperty name="mess"
```

```
                    property="text"
                    value="Hello" />
</jsp:useBean>
```

Besides those mentioned, `useBean` has some other, less used attributes. These are now listed:

- `id`. Gives a name to the variable to which we will assign the bean. It will instance a new object if we cannot find one with the same `id` and `scope`. In this case, the existing one will be used.

- `class`. Designates the full name of the Bean package.

- `scope`. Indicates the context in which the Bean will be available. There are four possible scopes:
  - `page directive`: indicates that the Bean will only be available for the current page. This means that it will be stored in the `PageContext` of the current page.

  - `request`: the bean will only be available for the current client request, stored in `ServletRequest`.

  - `session`: tells us that the object is available for all pages for the lifetime of the current `HttpSession` .

  - `application`: indicates that it is available for all pages that share the same `ServletContext`.

  The importance of scope lies in the fact that a `jsp:useBean` entry will only result in a new object if no previous objects exist with the same `id` and `scope`.
- `escribir`. Specifies the type of variable to which the object will refer. This must match the name of the class or superclass or an interface that implements the class. Remember that the name of the variable is designated with the attribute `id`.

- `beanName`. Gives the name of the Bean as we would supply it in the bn *instantiate* method. We can supply a type and BeanName, and ignore the attribute `class`.

`jsp:getProperty`

This action records the value of a ban property, converts it into a string and inserts this value in the output. It has two required attributes, which are:

- `name`: the name of a Bean loaded previous with `jsp:useBean`.

- `property`: the property of the Bean whose value we wish to obtain.

The following code reveals the operation of `jsp:getProperty`.

```
<jsp:useBean id="bean" ... />
<UL>
 <LI>Quantity:
    <jsp:getProperty name="bean" property="quantity" />
 <LI>Price:
    <jsp:getProperty name="bean" property="price" />
</UL>
```

jsp:setProperty

The `jsp:setProperty` action is used to assign values to Bean properties that have already been loaded. There are two options for assigning these values:

- At the time of instantiation. We can use `jsp:setProperty` when we are instantiating a bean. As a result, the values will only be assigned once during the lifetime of the Bean:

```
<jsp:useBean id="mens" class="MessageBean" >
<jsp:setProperty name="mess"
                    property="text"
                    value="Hello" />
</jsp:useBean>
```

- At some point in our code, if we use `jsp:setProperty` outside a `jsp:useBean` context, the value will be assigned to the property, regardless of whether it is the first instantiation or if the Bean had previously been instantiated.

```
<jsp:useBean id="mens" class="MessageBean" />
.....
<jsp:setProperty name="mess"
                    property="text"
                    value="Hello" />
....
<jsp:setProperty name="mess"
                    property="text"
                    value="Bye" />
```

The `jsp:setProperty` action has four possible attributes:

- `name`. This attribute designates the Bean whose property is to be modified.

- `property`. This attribute indicates the property we wish to operate on. There is a special case: a value of `"*"` means that all of the parameters of the HTTP request whose names match the names of the Bean's properties will be passed to the appropriate selection methods.

- `value`. This optional attribute specifies the value for the property. Values are automatically converted with the standard method `valueOf` in the source or enclosing class. We cannot use `value` and `param` together, but we can ignore both.

- `param`. This optional parameter indicates that a parameter of the HTTP request will be used to give a value to the property. If the HTTP request does not have this parameter, the system does not call the `setX` bean property method.

Where it exists, the following code passes the value of the `numItems` parameter to the Bean for the latter to assign it to its property, `numberItems`.

```
<jsp:setProperty name="order"
                  property="numberItems"
                  param="numItems" />
```

If in the `jsp:setProperty` action we ignore `value` and `param`, the container will assign the value of the HTTP request parameter with an identical name to the specified property. Using the capability of not specifying the property being assigned (by using `"*"`), we can easily assign all properties corresponding to HTTP request parameters to a Bean.

```
<jsp:setProperty name="order"
                  property="*" />
```

`jsp:forward`

This action is used to resend the request made to another page. It has only one parameter, `page`, which will contain the target URL. We can use static values or use a value generated dynamically.

```
<jsp:forward page="/underconstruction.jsp" />
<jsp:forward page="< %= urlDestination %>" />
```

`jsp:plugin`

This action is used to insert a specific OBJECT or EMBED element of the browser to specify that the browser must execute an applet using the Java plug-in.

4. Other dynamic content options

In addition to the technologies we have seen so far, there are other systems, technologies and languages designed for the development of dynamic web content.

One of the preferred systems, as an alternative to the ones we have seen, is `mod_perl`, an Apache server module that can be used to write web pages in Perl programming language in a similar way to how PHP is used. This module offers a series of obvious advantages over writing CGIs in Perl:

- Enhanced memory use. It behaves in a similar way to PHP, since the Perl module is only launched once when the web server is booted and remains in the memory from this point on. This avoids the problem of having to start Perl for each CGI.

- Faster response. When a module is preloaded, the response is more agile, which is the case of precompiled Perl programs (Perl precompiles the code to an intermediate code which then interprets it).

- It gives programs more direct access to server information. The module provides a richer and more efficient gateway than that facilitated by the CGI environment variables.

- It allows server extensions to be written entirely in Perl.

Two of the main advantages of `mod_perl` are the substantial increase in program performance and speed on the one hand, and the fact that a CGI program written in Perl needs only minimal attention to convert it into one in `mod_perl`. These two advantages make it a very valid option in situations where we already have several CGI programs written in Perl. One of the disadvantages of `mod_perl` is that it is only available for Apache servers, so it will not be a valid alternative if we cannot use Apache for our work.

Many web servers, including some of the ones we have seen, like Roxen, offer programming mechanisms with a similar philosophy to JSP. Roxen in particular, perhaps one of those with the greatest wealth of options for application development, offers us the possibility of extending our HTML pages with:

- RXML code, a Roxen HTML extension that incorporates all the elements of a programming language: conditionals, loops etc., and a rich

function library that includes elements like access to databases, LDAP, communications, graphics, string handling, etc.

- Code written in Pike, the object-oriented language in which Roxen was developed.

- PHP code, equalling the features of Apache in this case.

- Perl code, which does not offer the same features as `mod_perl`, but does have a wide range of options.

Like Roxen, both AOLServer and Apache can be used to develop server extension modules that would allow the handling of new HTML tags, new requests or communication protocols etc. Some of these systems can be used to develop programming extensions, such as template languages, with a similar philosophy to JSP. This is the case of Mason, DTL, etc.

Other options include the use of a "complex" server incorporating extension mechanisms (with an own language and another for general use), mechanisms for developing dynamic content and dynamic pages, all in a single product. One of the best known of these is Zope, based on Python programming language, which is a free software applications server for building portals, web applications, content managers, etc. It offers programmers a wide range of development features with a rich and powerful API for processing HTTP requests, database access, etc.

Lastly, there are some top-level options, many based on one of the above products and designed for the development of complex web applications. Some of these, like Enhydra (http://www.enhydra.org), are based on JSP/Servlets (Enhydra is also an excellent servlet container). Others, like OpenACS, are applications very much oriented to a specific type of website.

Example

OpenACS, for example, was designed to develop websites for communities, portals, etc.

One of the weaknesses of OpenACS is that it depends on AOLServer and TCL. This is a common feature of very high level packages, which are usually very closely linked to a specific web server. On the other hand, OpenACS offers a wide range of modules and features for creating websites.

It also has a wide variety of CMS (*Content Management Systems*) products for most free software web servers and servlet containers, with sufficient features for modifying, adapting and programming some of the more complex projects. In this cases, the complexity linked to the full development of a web application is not needed.

5. Practical: creation of a simple application with the techniques described

We are going to create a form that collects greetings and displays them on the screen. To do this, we will use two of the techniques described: CGI and servlets.

5.1. CGI

We have chosen Perl programming language to write our CGI program. The program code is:

```
#!/usr/bin/perl

print "Content-type: text/html\n\n\n";
print "<html>\n";
print "<body>\n";

$QS=$ENV{"QUERY_STRING"};
if ($QS ne "")
{
 @params=split /&/,$QS;

 foreach $param (@params)
 {
     ($nom,$val)=split /=/,$param;
     $val=&#732;s/\+/ /g;
     $val =&#732; s/ %([0-9a-fA-F]{2})/chr(hex($1))/ge;
     if($nam eq "greeting")
     {
           open FIT,">>list";
           print FIT "$val\n";
           close FIT;
     }
 }
}
 open FIT,"<list";
 while(<FIT>)
 {
     print "$_\n<HR>\n";
 }
 close FIT;

print << "EOF";
```

```
<FORM METHOD=GET ACTION="visit.cgi">
Greeting: <INPUT TYPE=Text NAME="greeting" SIZE=40>
<BR>
<INPUT TYPE="submit" NAME="SEND" VALUE="SEND">
</FORM>
</BODY>
</HTML>
EOF
```

As we can see, in this case, we need to manually process the environment variable containing the parameters.

5.2. Java Servlet

We will now look at an equivalent implementation using Java Servlets:

```
import java.io.*; import java.text.*; import java.util.*; import
javax.servlet.*; import javax.servlet.http.*;
/**
 * Form processing
 *
 * @author Carles Mateu
 */

 public class Form extends HttpServlet {
     Greetings vector;
     public void init(ServletConfig sc)
             throws ServletException
     {
          greetings=new Vector();
     }

     void listGreetings(PrintWriter out)
     {
     for (Enumeration e = greetings.elements() ; e.hasMoreElements() ;)
    {
          out.println(e.nextElement()+"<HR>");
          )
     }

     public void doGet(HttpServletRequest request,
                 HttpServletResponse response)
     throws IOException, ServletException
 {
     response.setContentType("text/html");
     PrintWriter out = response.getWriter();
 String greeting=request.getParameter("greeting");
```

```
  if(greeting!=null)
  {
       greetings.add(greeting);
  }
       out.println("<html>");
       out.println("<body bgcolor=\"white\">");
  listGreetings(out);
  out.println("<FORM METHOD=GET ACTION=\"Form\">\n"+
       "Greeting: <INPUT TYPE=Text NAME=\"greeting\" SIZE=40>\n"+
       "<BR> <INPUT TYPE=\"submit\" "+









       "NAME=\"SEND\" VALUE=\"SEND\">\n"+
       "</FORM> </BODY> </HTML>\n");
  out.println("</body>");
  out.println("</html>");
  }
 }
```

In this case, the persistence of the data is only for the servlet, since the data are saved to the memory.

Database access: JDBC

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Carles Mateu

PID_00148405

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Index

Introduction

One of the most basic needs we face when developing applications for the Internet with Java, and for many other applications, is to have a powerful, robust, fast and easily accessible data store. There are several free software database management systems (DBMS). We will look at some of these in the following sections before introducing a technology from Java for DBMS access, particularly relational databases, called JDBC (sometimes incorrectly expanded as *Java database connectivity*).

1. Introduction to databases

One of the keys to the development of web applications is the choice of DBMS to use. There are currently several free software DBMS available, many of a similar quality to the more well-known commercial DBMS.

Most free software DBMS come from two main sources: either from projects that started out as free software projects (research projects, etc.) or DBMS that belonged to proprietary software companies whose main business was not DBMS. These companies subsequently opted to release the product under a free software licence, thus opening up its development to the community. We will now look at some of the most emblematic free software DBMS.

1.1. PostgreSQL

PostgreSQL (or Postgres) is one of the most well-known and long-standing DBMS in the free software community. It was launched in the mid-1980s at the University of Berkeley with the name Postgres following research on the group of Berkeley databases (particularly by Michael Stonebraker). Postgres continued to evolve until Postgres 4.2 in 1994. Postgres did not use SQL as a query language; it had its own language called Postquel. In 1995, Andrew Yu and Jolly Chen added an SQL interpreter to Postgres 4.2, giving rise to the birth of Postgres95, a product released under a free software licence that left the confines of Berkeley to become an Internet development. In 1996, a new name was chosen that would withstand the passing of time and reflect the project's relationship with the original Postgres (still available) and new differences (basically the use of SQL). Hence, PostgreSQL was born.

PostgreSQL has since become one of the databases of choice in numerous projects, offering users features of the level of commercial database management systems such as Informix and Oracle.

The most significant features of PostgreSQL are:

- Transaction support.
- Subqueries.
- Viewing support.
- Referential integrity.
- Table inheritance.
- User-defined types.
- Columns as arrays that can store more than one value.
- Addition of fields to tables at runtime.
- User-definable aggregation functions (like sum() and count()).
- *Triggers*, SQL commands that need to be executed when acting on a table.

- System tables that can be queried to obtain information on tables, the database and the database engine.
- Support for large binary objects (over 64 KB).

1.2. MySQL

MySQL and PostgreSQL are currently fighting it out for the title of most well-known and widely-used free software DBMS. MySQL is a DBMS developed by MySQL AB, a Swedish company that develops it under a free software licence (specifically GPL), although it can also be purchased with a commercial licence if required, to be included in projects that are not free software.

MySQL is an extremely fast database management system. While lacking the capabilities and features of many other databases, it offsets this lack of features with an excellent performance that makes it the database of choice when we only need basic capabilities.

The most significant functionalities of MySQL are:

- Transaction support (new in MySQL 4.0 if InnoDB is used as the storage engine).
- Replication support (with a *master* updating multiple *slaves*).
- Library for embedded use.
- Text search.
- *Search* cache (for enhanced performance).

1.3. SAP DB

SAP DB is nothing more than the DBMS of the business management software giant SAP (authors of the famous SAP/R3). For a long time, this company's product *portfolio* included a relational DBMS called SAP DB. In April 2001, the company decided to make it available to the world under a new licence, GPL. From this point on, SAP DB has been developed under a free software licence.

SAP DB is a very powerful database which originated from a very specialised environment, SAP applications, and has thus not spread very well among the free software community. Nonetheless, SAP DB has some very powerful features which, combined with the prestige of the company that created it, make it a serious candidate to become the database of choice for some of our free software projects.

- Support for *outer joins*.
- Support for user roles.
- Updatable views.
- Implicit transactions and locks.
- Scrollable *cursors*.

- Stored procedures.

1.4. FirebirdSQL

FirebirdSQL is a free software database originating from the free software version of Interbase that Borland/Inprise released in the summer of 2000. Since the licence used to release this version and the way that Borland planned to work were not very clear, a group of developers started their own version of Interbase, which they called FirebirdSQL.

The first aim of the developers of FirebirdSQL was to stabilise the code and eliminate the many *bugs* as well as to increase the number of platforms on which the database could work. Since then, both the features of the database and the number and quality of the functions it offers have been gradually developed. Some of its most important functionalities currently include:

- Version architecture to avoid readers/writers locks.
- Events alert to react to database changes.
- Very rich data types (BLOBS etc).
- Stored procedures and *triggers*.
- ANSI SQL-92 compatibility.
- Referential integrity.
- Transactions.
- Support for multiple interconnected databases.

2. Controllers and addresses

2.1. JDBC controllers

Despite the many similarities between the different DBMS, their languages, features, etc., the communication protocols that need to be used to access them vary substantially from one to the next. Hence, to communicate with the different DBMS from JDBC, we need to use a *driver* to isolate the specific features of the DBMS and its communication protocol.

There are several different types of JDBC drivers, which we can classify as follows:

- Type 1 drivers. *Bridging drivers*. These drivers translate JDBC calls to calls from another DBMS access language (such as ODBC). They are used when there is no more appropriate JDBC driver. It involves installing the driver on the client machine in order to translate JDBC calls. The most well-known is the JDBC-ODBC driver, which acts as a bridge between JDBC and ODBC.

- Type 2 drivers. *Native API Partly Java Drivers*. These drivers use the Java JNI (Java *native interface*) API to present a Java interface to a native binary DBMS driver. As with type 1 drivers, these require the native driver to be installed on the client machine. Their performance is usually superior to drivers written entirely in Java, although an operating error in the native part of the driver can cause problems in the Java Virtual Machine.

- Type 3 drivers. *Net-protocol All-Java Drivers*. Controllers written in Java defining a communication protocol that interacts with a *middleware* program that, in turn, interacts with a DBMS. The communication protocol with the *middleware* is a network protocol independent of the DBMS and the *middleware* program must be able to communicate the clients with the diverse databases. The disadvantage of this option is that we need another level of communication and another program (the *middleware*).

- Type 4 drivers. *Native-protocol All-Java Drivers*. These are the most widely-used drivers in intranet accesses (those generally used in web applications). They are written entirely in Java and translate JDBC calls to the DBMS's own communication protocol. They do not require further installation or extra programs.

Almost all modern DBMS already have a JDBC driver (especially type 4).

Note

For a list of existing drivers, visit http://java.sun.com/products/jdbc/jdbc.drivers.html

2.2. Loading the Java driver

To use a JDBC driver, we must first register it in the JDBC `DriverManager`. This is usually done by loading the driver class using the `forName` method of the class called `Class`. The usual construction is:

```
try
{
Class.forName("org.postgresql.Driver");
}
catch(ClassNotFoundException e)
{
....
}
```

2.3. Database addresses

To identify a given connection to a database, DBMS use a URL (*Universal Resource Locator*) address format. This address usually takes the form:

```
jdbc:driver:database
```

In actual fact, the format is very flexible as manufacturers have complete freedom to define it.

Some of the most common formats are:

Table 10. Formats

PostgreSQL	jdbc:postgresql://127.0.0.1:5432/database
Oracle	jdbc:oracle:oci8:@DBHOST
JDBC-ODBC	jdbc:odbc:dsn;optionsodbc
MySQL	jdbc:mysql://localhost/database?user=joseph&password=joe
SAP DB	jdbc:sapdb://localhost/database

We can see that PostgreSQL specifies the IP address of the server and the port (5432) and the name of the database. Oracle, on the other hand, specifies a subdriver (oci8) and a database name base in the line of those defined by Oracle TNS. The examples of connection addresses reveal that, while different, they all follow a very similar pattern (particularly PostgreSQL, MySQL and SAP DB).

2.4. Connecting to the database

The simple method of connecting to a database will provide us with a `Connection` type object that will encapsulate a simple connection. In each application, we can have as many connections as system resources will allow (especially those of the DBMS) and maintain connections to different DBMS.

To obtain a `Connection` we will use the `DriverManager.getConnection()`. Never instantiate a `Connection` type object directly.

```
Connection with =
   DriverManager.getConnection ("url","user","password");
```

We pass three parameters to `getConnection`: the address of the database in the format seen above, the user and the password. For databases that do not require a user and password, leave these blank. When we call this method JDBC will ask each registered driver if it supports the URL we have passed and, if so, it returns a `Connection`.

When a `Connection` is no longer going to be used, we must close it explicitly with `close()` so as not to use up resources. It is particularly important to free database connections as they are a very costly resource.

Version 2.0 onwards of JDBC also has a mechanism for *pooling* connections, allowing us to use a block of preset connections that are used again and again.

3. Basic database access

Once we have a `Connection` object, we can start to use it to execute SQL commands in the database. There are three basic types of SQL statement in JDBC:

`Statements`. This is a basic SQL statement, whether for querying (SELECT) or handling data (INSERT, UPDATE etc).

`PreparedStatement`. Represents a precompiled SQL statement with better features than basic statements.

`CallableStatement`. Represents a call to a stored SQL procedure.

3.1. Basic statements

To obtain a `Statement` object, we use the `createStatement` method of the `Connection`:

```
Statement sent=con.createStatement();
```

Once we have created the `Statement` object, we can use it to execute SQL commands in the database. SQL commands may or may not return results. If they return results in table form (for example, with a SELECT type SQL command) we use the `executeQuery` method of `Statement` to execute them:

```
ResultSet rs=sent.executeQuery("SELECT * FROM CLIENTS");
```

We will study `ResultSet` in detail later. In this code, we have used `executeQuery` to execute a data query. There is another method, `executeUpdate`, for executing statements that do not return results, such as UPDATE or DELETE. `executeUpdate` returns an integer that tells us the number of rows affected by the SQL command sent.

```
int columns=sent.executeUpdate("UPDATE CLIENTS SET BALANCE=0");
```

Where we do not know *a priori* if a statement will return a table of results (like `executeQuery`) or a number of rows affected (like `executeUpdate`), we have a more generic method, `execute`. `execute` returns `true` if there is a `ResultSet` associated with a statement and `false` if this statement returns an integer. In the first case, we can record the resulting `ResultSet` with `getResultSet`, while in the second, using `getUpdateCount` we can record the number of rows affected.

```
Statement sent=con.createStatement();
if(sent.execute(SQLstatement))
{
 ResultSet rs=sent.getResultSet();
 // show results
}
else
{
 int affected=sent.getUpdateCount();
}
```

Note that a `Statement` represents a single SQL statement, so if we make a call to `execute`, `executeQuery` or `executeUpdate` the `ResultSet` associated with this `Statement` will be closed and released. It is therefore very important to have finished processing the `ResultSet` before launching any other SQL command.

To close a `Statement`, we can use the `close`. Although when we close the `Connection` we also close the `Statement` associated with it, it is much better to close them explicitly in order to free up occupied resources first.

3.1.1. Multiple results

It is possible for a `Statement` to return more than one `ResultSet` or more than one number of affected rows. `Statement` supports multiple returns with the `getMoreResults`. This method returns `true` if there are more `ResultSet` waiting. The method returns `false` if the next return is a number of affected rows, even though there could be more `ResultSet` after the number. In this case, we will know whether we have processed all of the results if `getUpdateCount` returns -1.

Note

Whether or not an SQL statement can return more than one result or modified column count will depend on the DBMS, generally as a result of stored procedures.

Thus, we can modify the above code to support multiple results:

```
Statement sent=con.createStatement();
sent.execute(SQLstatement);
while(true)
{
 rs=sent.getResultSet();
 if(rs!=null)
 {
 // show results
 }
 else
 {
 // show number
 }
 // Next or last
```

```
 if((sent.getMoreResults()==false) &&
      (sent.getUpdateCount()==-1))
       break;
}
```

3.2. Results

The execution of any SQL query statement (SELECT) produces a table (a pseudo-table, in fact) containing the data that meet the established criteria. JDBC uses a `ResultSet` class to encapsulate these results and offer methods for accessing them.

We can imagine `ResultSet` as a series of data reaching us from the DBMS. We cannot go backwards in it, so we need to process them as we move through the series. In JDBC 2.0, *scrollable cursors* allow us to move freely through the results.

The following is an example of code that will allow us to process the results of a `ResultSet` is:

```
Statement sent=con.createStatement();
ResultSet rs=sent.executeQuery("SELECT * FROM CLIENTS");
while(rs.next())
{
  System.out.println("Name:"+rs.getString("NAME");
  System.out.println("City:"+rs.getString("CITY");
}
rs.close();
  sent.close();
```

This code runs through the series of `(ResultSet)` results, iterating through each row with the `next`. Initially, once `ResultSet` has bee obtained, JBDC positions us before the first element of the list. Hence, to access the first element, we need to call `next`. To read the second row, we need to call `next`. If there are no more rows to read, `next` returns `false`.

Once we are positioned in the row we wish to read, we can use the `getXXXX` methods to obtain the specific column we wish to display. There are several `getXXXX` methods one for each data type that we can read from the DBMS. The `getXXXX` methods take as their parameter a string that must be the name of the field or a numerical parameter that indicates the column by position, bearing in mind that numbering begins at 1, not 0, as is usually the case in `arrays`, etc.

Another useful method is `getObject`, which returns a Java object.

For example, if we use `getObject` with an entire column, it will return an `Integer` object ype, whereas if we use `getInt`, it will return an `int`.

The table below contains SQL data types together with the object types returned by JDBC and the specific method for the type. If the type returned by the method is different to the Java object, this type will be displayed in parentheses.

A useful option in a number of situations is `getString` which can be used for all types, since JDBC converts most SQL type characters to strings. As many web applications display data in a simple way, this is a very interesting option.

Table 11. SQL data types and JDBC returns

SQL type	Java type	getXXXX method
CHAR	String	getString()
VARCHAR	String	getString()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	Boolean (boolean)	getBoolean()
TINYINT	Integer (byte)	getByte()
SMALLINT	Integer (short)	getShort()
INTEGER	Integer (int)	getInt()
BIGINT	Long (long)	getLong()
REAL	Float (float)	getFloat()
FLOAT	Double (double)	getDouble()
DOUBLE	Double (double)	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBytes()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

3.2.1. Processing *null*

In some databases, certain columns can have *null*). The processing of these columns in JDBC is complicated because some drivers do not do it correctly. Some methods that return objects return `null` in this case, but those that are particularly vulnerable to error are those that do not return objects such as `getInt.getInt`.

For example, if -1 is returned when a null value is found in a column, we obviously cannot tell whether -1 was the value or whether the column was null.

There is a method for working out whether the last result obtained was null: `wasNull`.

```
int quantity=rs.getInt("QUANTITY");
if(rs.wasNull())
 System.out.println("The result was null");
else
 System.out.println("Balance: "+quantity);
```

Remember that `wasNull` refers to the last column read.

3.2.2. Large data types

We can obtain Java *streams* to access columns containing large data types, such as images, text, documents, etc. The JDBC methods for this are `getAsciiStream`, `getBinaryStream` and `getCharacterStream`. These methods return an object of the type `InputStream`. The following example reads an object of this type from the database and writes it in an `OutputStream` that could correspond to the output of a servlet for displaying an image.

```
ResultSet res=
 sent.executeQuery("SELECT PHOTO FROM PEOPLE"+
 " WHERE ID NO. ='"+id+"'");
if(res.next())
{
 BufferedInputStream image=
      new BufferedInputStream
      (res.getBinaryStream("PHOTO"));
 byte[] buf=new byte[4096]; // Buffer of 4 kbytes
 int lengt;
 while((lengt=image.read(buf,0,buf.length))!=-1)
 {
      outstream.write(buf,0,len);
 }
```

```
}
```

JDBC 2.0 also has two specific objects for processing large objects (BLOB, *Binary Large Objects* and CLOB, *Character Large Objects*), called `Blob` and `Clob`, respectively.

We can access these two objects as `streams` or directly, with methods such as `getBytes`. There are also methods for passing `Blobs` and `Clobs` to prepared statements.

3.3. Bug management

If a serious error is found during execution of a JDBC object preventing it from continuing, an exception is usually launched, specifically `SQLException`.

Example

These can be caused by wrong URLs, security problems, etc.

`SQLException` extends `Exception` and defines several additional methods. One of these is `getNextException`. This method strings several exceptions into one if JDBC finds more than one serious error.

`SQLException` also defines other methods for obtaining more information on the type and nature of the error: `getSQLState` and `getErrorCode`. `GetSQLState` returns a status/error code from the database in line with the table of codes defined by ANSI-92 SQL. `getErrorCode` returns an error code from the DBMS manufacturer.

An example of complete code of a `catch` managing all possible exceptions might be:

```
try
{
       // actions on the database
}
catch(SQLException e)
{
  while(e!=null)
  {
   System.out.println(
        "Exception: "+e.getMessage());
   System.out.println(
        "ANSI-92 SQL code: "+e.getSQLState());
   System.out.println(
        "Manufacturer's code:"+e.getErrorCode());
   e=e.getNextException();
  }
}
```

3.3.1. SQL *warnings*

Besides being able to launch exceptions in the event of errors, JDBC can generate a series of warnings about conditions that are wrong but not serious (allowing it to continue). The types of error generating a warning rather than an error are decided by the database manufacturer and vary. To access these warnings, we can use the `SQLWarning` object, used in a similar way to `SQLException` with the difference that we cannot use it in a `try-catch` block; nstead, we must interrogate JDBC if there are `SQLWarnings` after each operation.

During debugging or operation we can use the following trick to capture all `SQLWarnings`:

```
void impWarnings(SQLWarning w)
{
 while(w!=null)
 {
     System.out.println("\n SQLWarning: ");
     System.out.println(w.getMessage());
     System.out.println("ANSI-92 SQL State: "
          +w.getSQLState());
     System.out.println("Manufacturer Code: "
          +w.getErrorCode());
     w=w.getNextWarning();
 }
}
```

We can encompass all JDBC calls with our `impWarnings` method to capture all possible `SQLWarnings`. It can be used as follows:

```
ResultSet r=sent.executeQuery("SELECT * FROM SUPPLIERS");
impWarnings(sent.getWarnings());
impWarnings(r.getWarnings());
```

4. Prepared statements and stored procedures

The statements we have studied thus far can be used to execute all SQL orders in the database (insertion, query, elimination, etc.), albeit in a primitive way. To accelerate and enhance performance or to execute procedures that we may have in the database, we need to use other mechanisms.

4.1. Prepared statements

One of the most difficult steps in the execution of SQL statements is the compilation and planning of statement execution. In other words, decisions about which tables to consult first, how to access them, in which order, etc. A very widespread technique for getting round these difficulties is to use compiled or prepared statements. Prepared statements are SQL statements sent to the DBMS for the latter to prepare before executing them, which we will then use repeatedly, changing certain parameters while reusing the planning of an execution for the next one.

A `PreparedStatement` object is created, like `Statement`, from a database connection:

```
PreparedStatement sp=con.prepareStatement(
 "INSERT INTO CLIENTS (ID,NAME) VALUES (?,?)");
```

As the purpose of the `PreparedStatement` is to execute a statement repeatedly, we do not specify the values that need to be inserted. Instead, we add special markers that we will later replace with the values we actually want to insert:

```
sp.clearParameters();
sp.setString(1,"0298392");
sp.setString(2,"JIMINY CRICKET");
sp.executeUpdate();
```

Before assigning values to the markers, we need to clear the current assignation. To do so, we have a set of `setXXXX` calls, imilar to the `getXXXX` method of `ResultSet`, for assigning these values. The values are referenced positionally, beginning with 1. The `setObject` call is used to assign Java objects to markers, while JDBC converts the format. There are three options for calling `setObject`:

```
setObject(int index, Object ob, int SQLtype, int scale);
setObject(int index, Object ob, int SQLtype);
```

```
setObject(int index, Object ob);
```

In these calls, `SQLType` is a numerical reference to one of the SQL type constants defined in the class `java.sql.Types`.

To insert a null value in the database, we can use the `setNull` call or we can use `setObject` with a `null` paramet r, specifying the SQL type we want.

4.2. Stored procedures

Many modern databases come with their own programming language for developing procedures and functions executed by the DBMS itself. There are several advantages to this. Firstly, we have a code that is independent of the applications and can be used in many programs developed in numerous programming languages. Secondly, this code isolates design applications from the database, providing an interface that is independent of the form of the tables for certain operations, which means that we can modify these and only need to modify the stored procedures. Moreover, in some cases, this approach offers substantial performance improvements because not only do the data not have to travel through the network for processing, but the entire process is carried out locally in the DBMS and only the results have to travel.

This is an example of a stored procedure written in PL/PGSQL, one of the PostgreSQL programming languages; in other DBMS, the language and syntax would be totally different.

```
CREATE OR REPLACE FUNCTION proNewClient
 (VARCHAR) RETURNS INTEGER AS '
DECLARE
 name ALIAS FOR $1;
 iden integer;
BEGIN
 SELECT max(id) INTO iden
 FROM CLIENTS;
 iden:=iden+1;
INSERTINTOCLIENTS(ID,NAME)VALUES(iden,name);
 RETURN iden;
END
' LANGUAGE 'plpgsql';
```

This example receives a parameter, a string with the name of the client. It then inserts it and returns the identifier assigned to this client.

To call it, JDBC gives us the `CallableStatement`. Since each DBMS has its own syntax for calls to functions and stored procedures, JDBC provides a standard syntax for these.

If the stored procedure does not return values, the syntax of the call is:

```
{call procedurename [(?[,?...])]}
```

If the procedure returns values (as is the case of the function we have defined), the syntax will then be:

```
{? = call procedurename [(?[,?...])]}
```

Parameters are optional and represented by `?`, as in prepared statements. The JDBC driver will translate these calls to those corresponding to the DBMS. The Java code that would call the function we have defined here would be:

```
CallableStatement proc=
 con.prepareCall("{?=call proNewClient(?)}");
proc.registerOutParameter(1,Types.INTEGER);
proc.setInt(2,"Name");
proc.execute();
System.out.println("Result:"+proc.getInteger(1));
```

It is only necessary to use `CallableStatement` with stored procedures that return values. We can call the stored procedures that do not return values through the previous statement objects.

5. Transactions

One of the most important aspects and functionalities of modern DBMS is the possibility of carrying out transactions.

A transaction is a series of database operations carried out atomically, that is, as if they were an indivisible operation. This allows us to combine several SQL statements to perform operations to take us to a specific point.

For example, if inserting a student in an academic database requires inserting him or her in the students table, creating an academic record entry, creating an e-mail entry, etc., it is a good idea to ensure that these operations are executed as a single, atomic whole.

The steps for working with transactions are: begin the transaction, execute the operations and, lastly, if the transaction is correct, validate it and save the changes in the database; if it is not correct or a problem has occurred, we need to undo the changes. This ability to undo changes is key to the operation of transactions, meaning that if any of the operations in the transaction is incorrect, we can undo all changes and leave the database as if none of the operations had taken place. Thus, in our case, it will be impossible for us to end up in a situation where we have an entry in the student table but do not have the corresponding academic record or e-mail address entry.

Another feature of transactions is the possibility of choosing when the transaction data are visible to the rest of the applications. For example, we can choose to make the student data visible only when the transaction is completely finished, so the academic record cannot be read until the entire transaction has been performed.

In JDBC, transactions are managed by the `Connection object`. By default, JDBC operates in auto-transaction mode, in other words, each SQL operation is within a transaction that is immediately validated. To perform multiple operations in a transaction, we must first disable automatic validation. To do so, we can use `setAutoCommit`. Transactions can now be validated with `commit` and invalidated with `rollback`.

```
try
 {
 // Disable automatic validation
 con.setAutoCommit(false);

 sent.executeUpdate("INSERT
```

```
 ...
 sent.executeUpdate("INSERT ....");
 con.commit();
}
catch(SQLException e)
{
 // If an error occurs, invalidate
con.rollback();
}
```

JDBC also supports several transaction isolation modes, used to control how the database resolves conflicts between transactions. JDBC has five transaction resolution modes, which may not be supported by the DBMS. The default mode, which will be enabled if we do not specify another, will depend on the DBMS. As we increase the level of transaction isolation, performance drops. The five modes defined in `Connection` are:

- `TRANSACTION_NONE`. Transactions disabled or not supported.

- `TRANSACTION_READ_UNCOMMITTED`. Minimum transactions allowing *dirty reads*. The other transactions can see the results of the operations of transactions in progress. If these transactions subsequently invalidate operations, the other transactions would end up as invalid data.

- `TRANSACTION_READ_COMMITTED`. The other transactions cannot see the non-validated data (*dirty reads* not allowed).

- `TRANSACTION_REPEATABLE_READ`. Allows repeatable reads. If a transaction reads data that is subsequently altered by another (and the modification is validated), if the first transaction reads the data again, the result obtained will be no different to the first one. Only when the transaction is validated and we start it again will we obtain different data.

- `TRANSACTION_SERIALIZABLE`. Adds insertion protection to the behaviour of `TRANSACTION_REPEATABLE_READ`. If a transaction reads from a table and another transaction adds (and validates) data in the table, the first transaction will obtain the same data if they are re-read. This forces the DBMS to treat the transactions as if they were a series.

6. Metadata

Until now, we have accessed data inside the database. But we can also access another type of data, metadata, which are simply data on data. These metadata will provide us with information on the form of the data we are working with and information on the features of the database.

6.1. Database metadata

The metadata of the database will provide us with information on the basic features of the DBMS we are using, together with information on certain features of the JDBC driver we are using. To give us this access, Java offers a class (`DatabaseMetaData`) that will encompass all of this information.

We will obtain a `DatabaseMetaData` object type from a `Connection`. This means that, to obtain these metadata, we need to have established a connection to the database.

```
DatabaseMetaData dbmd= con.getMetaData();
```

Once we have obtained the DatabaseMetaData instance that will represent the metadata of the database, we can use the methods provided by the latter to access diverse types of information on the database, the DBMS and the JDBC driver. We can divide the information provided by JDBC into the following categories:

- Those that obtain information on the DBMS.
- Those that obtain information on the JDBC driver used.
- Those that obtain information on the operating limits of the DBMS.
- Those that obtain information on the database schema.

6.1.1. Information on the DBMS

There are methods that provide certain *informative* data on the data engine (versions, manufacturer, etc.), and certain information that can be useful for making our programs react better to possible changes in the DBMS.

These methods include:

Table 12. Methods

Name	Description
`getDatabaseProductName`	Name of the DBMS

Name	Description
`getDatabaseProductVersion`	Version of the DBMS
`supportsANSI92SQLEntryLevelSQL`	EL-ANSI-92 support
`supportsANSI92FullSQL`	ANSI-92 support
`supportsGroupBy`	GROUP BY support
`supportsMultipleResultSets`	Support for multiple results
`supportsStoredProcedures`	Support for stored procedures
`supportsTransactions`	Transaction support

For a full list, see the documentation on JDK (Java *Development Kit*).

6.1.2. Information on the JDBC driver used

Just as we can obtain information on the DBMS, we can also obtain certain information on the JDBC driver. The basic information we can obtain are name and version number.

Table 13.

Name	Description
`getDriverName`	Driver name
`getDriverVersion`	Driver version
`getDriverMajorVersion`	Major part of the driver version
`getDriverMinorVersion`	Minor part of the driver version

6.1.3. Information on the operating limits of the DBMS

These provide information on the limits of the specific DBMS, the maximum field measurement, etc. They are very useful for adapting the application to operate independently of the DBMS. Some of the most important of these are:

Note

Some DatabaseMetaData methods that accept strings as parameters accept special characters for queries, % to represent any character group and to represent a character.

Table 14.

Name	Description
`getMaxColumnsInSelect`	Maximum number of columns in a query
`getMaxRowSize`	Maximum measurement in a row permitted by the DBMS
`getMaxTablesInSelect`	Maximum number of tables in a query.

6.1.4. Information on the database schema

We can obtain information on the schema (tables, indexes, columns, etc.) from the database. This allows us to "explore" databases to reveal the database structure. Some of these methods include:

Table 15.

Name	Description
`getColumns`	Provides the columns of a table
`getPrimaryKeys`	Provides the keys of a table
`getTables`	Provides the tables of a database
`getSchemas`	Provides the schemas of the database

6.2. Results metadata

Not only can we obtain data on the DBMS (`DatabaseMetaData`), it is also possible to obtain data on the results of a query, which means that we can obtain information on the structure of a `ResultSet`. Thus, we can find out the number of columns, the type of column and its name, as well as additional information (whether nulls are supported etc). Some of the most relevant methods of `ResultSetMetaData` are:

Table 16. Methods `ResultSet MetaData`

Name	Description
`getColumnCount`	Number of columns of `ResultSet`
`getColumnLabel`	Name of a column
`getColumnTypeName`	Name of a column type
`isNullable`	Supports nulls
`isReadOnly`	Is modifiable

The following is an example of code displaying the form of a table:

```
ResultSetrs=sent.executeQuery("SELECT*FROM"+table=);
ResultSetMetaData mdr=rs.getMetaData();
int numcolumns=mdr=getColumnCount();
for(int col=1;col<numcolumns;col++)
{
 System.out.print(mdr.getColumnLabel(col)+":");
 System.out.println(mdr.getColumnTypeName(col));
}
```

7. Practical: database access

We will now create some simple programs allowing us to access databases from Java.

The first will perform the simplest function, a query:

```
// Simple example of JDBC.
// Selects the database
//

//
import java.sql.*;

public class SelectSimple {
 public static void main(java.lang.String[] args)
 {
    // Initial driver loading
    try
    {
       // Choose the appropriate driver for
       // your database
       Class.forName("org.postgresql.Driver");
    }
    catch (ClassNotFoundException e)
    {
      System.out.println("Driver not loaded");
      return;
 }
// All JDBC operations must process SQL
// exceptions.
try
  {
   // Connect
   Connection with = DriverManager.getConnection
     ("jdbc:postgresql:test", "user", "password");
   // Create and execute an SQL statement
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery
               ("SELECT * FROM students");

    // Display results
    while(rs.next()) {
          System.out.println(
```

```
            rs.getString("name") + " "+
            rs.getString("surname1"));
     }
     // Close database resources
     rs.close();
     stmt.close();
     con.close();
   }
   catch (SQLException se)
   {
      // Print errors
      System.out.println("SQL Exception: " + se.getMessage());
      se.printStackTrace(System.out);
   }
 }
}
```

The second program makes an insertion in the database:

```
//
// Example of updating
//
import java.sql.*;

public class UpdateSimple {
   public static void main(String args[])
   {
       Connection con=null;
          // driver loading.
          try
          {
               // Choose the appropriate driver
               Class.forName("org.postgresql.Driver");
          }
          catch (ClassNotFoundException e)
          {
               System.out.println("Error in driver");
               return;
          }
       try
       {
       // Connect to the database
       with = DriverManager.getConnection
         ("jdbc:postgresql:test", "user", "password");


         Statement s = con.createStatement();
```

```
         String ID no= new String("00000000");
         String name= new String("Carles");
         String sur1= new String("Mateu");
         int update_count = s.executeUpdate
              ("INSERT INTO students (ID no,name, sur1) " +
              "VALUES('" + ID no+ "','" + name+ "','" + sur1+ "')");
         System.out.println(update_count + " inserted columns.");
         s.close();
      }
      catch( Exception e )
      {
         e.printStackTrace();
      }
      finally
      {
         if( con != null )
          try { con.close(); }
              catch( SQLException e ) { e.printStackTrace(); }
      }
   }
}
```

And finally, a program that makes the query using prepared statements:

```
// Example of a prepared statement
//

//
import java.sql.*;
public class SelectPrepared {

      public static void main(java.lang.String[] args)
      {
         if(args.length!=1)
         {
           System.err.println("Argument: ID no of student");
           System.exit(1);
         }
         // Driver loading
         try
         {
           Class.forName("org.postgresql.Driver");
         }
         catch (ClassNotFoundException e)
         {
           System.out.println("Problems with driver");
           return;
```

```
          }
          try
          {
            // Connect to the database.
            Connection with = DriverManager.getConnection
            ("jdbc:postgresql:test", "user", "password");

            // Create statement
            PreparedStatement pstmt;
            String selec="SELECT * FROM students WHERE ID no=?";
            pstmt=con.prepareStatement(selec);

            // Assign arguments
            pstmt.setString(1, args[0]);
            ResultSet rs=pstmt.executeQuery();

            // Print results
            while(rs.next()) {
                System.out.println(
                  rs.getString("name") + " "+
                  rs.getString("sur1"));
           }
           // Close database resources
           rs.close();
           pstmt.close();
           con.close();
          }
            catch (SQLException se)
          {
            // Print errors
            System.out.println("SQL Exception: " + se.getMessage());
            se.printStackTrace(System.out);
        }
    }
}
```

Web services

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Carles Mateu

PID_00148399

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Introduction to web services

Web services are software components that have the following distinctive features for programmers:

- They are accessible via SOAP (*Simple Object Access Protocol*).
- Their interface is described with a WSDL (*Web Services Description Language*).

We will now look in detail at the meaning of these protocol names and formats. SOAP is a communication protocol using XML messages that is the basis for web services. SOAP allows applications to send XML messages to other applications. SOAP messages are unidirectional, although all applications can participate indiscriminately as senders or receivers. SOAP messages can serve for a number of purposes: request/response, asynchronous messaging, notification, etc.

SOAP is a high-level protocol that only defines the message structure and some basic processing rules for this. It is completely independent of the transport protocol. This means that SOAP messages can be exchanged through HTTP, SMTP, JMS, etc., although HTTP is the most common at present.

WSDL is a web services description standard based on an XML document. This document provides applications with all of the necessary information for accessing a web service. The document describes the purpose of the web service, its communication mechanisms, where it is located etc.

Another web services component is UDDI (*Universal Description, Discovery and Integration*). This is a web services registry service that stores the latter by their name, the URL of their WSDL, a text description of the service, etc. Interested applications can use SOAP to see which services are registered in UDDI, look up a service, etc.

2. XML-RPC

XML-RPC is a remote procedure call protocol that runs on the Internet. It is a much more straightforward protocol than SOAP and much easier to implement. XML-RPC works through the exchange of messages between the client of the service and the server, using HTTP to carry these messages. More specifically, XML-RPC uses HTTP POST requests to send a message in XML format, indicating:

- Procedure to be executed on the server
- Parameters

The server will return the result in XML format.

2.1. XML-RPC request format

We will now look at the format of an XML-RPC request. To do so, we will use an example request such as this:

```
POST /RPC2 HTTP/1.1
User-Agent: Frontier/5.1.2
Host: carlesm.com
Content-Type: text/xml
Content-Length: 186

<?xml version="1.0"?>
<methodCall>
 <methodName>example.Method</methodName>
 <params>
  <param>
   <value><i4>51</i4></value>
  </param>
 </params>
</methodCall>
```

We will now analyse this request line by line. First, the message header:

- The `URI` we first observe, `RPC2`, is not defined by the standard. This means that if the server responds to diverse types of request, we can use this `URI` to route them.

- The fields `User-Agent` and `host` are compulsory and the value must be valid.

- The `Content-Type` field will always be `text/xml`.
- The value of `Content-Length` must always be present and must be a correct value.

Then comes the body:

- The message contains a single element `<methodCall>` containing the sub-elements
- `<methodName>` which contains the string with the name of the method to invoke.
- If the method has parameters, it must have a `<params>`.
- With as many `<param>` as the method has parameters.
- Each of which has a `<value>`.

To specify the possible values of the parameters, we can use the following marks, which allow us to specify scalars:

Table 17. Marks

mark	type
`<i4> or <int>`	4-byte integer with sign
`<boolean>`	0 (false) or 1(true)
`<string>`	string `ASCII`
`<double>`	floating point with sign and double precision
`<dateTime.iso8601>`	date/time format `iso8601`
`<base64>`	binary encoded in `base-64`

If no type is specified, a `<string>` type will be assigned by default.

We can also use complex types. To do so, we can use a `<struct>`type, whose structure is as follows:

- It contains a series of `<member>`.
- Each of these has a `<name>` and `<value>` of one of the basic types.

For example, a `<struct>` type parameter would be:

```
<struct>
  <member>
```

```
      <name>name</name>
      <value><string>Juan Manuel</string></value>
    </member>
    <member>
      <name>Passport</name>
      <value><i4>67821456</i4></value>
    </member>
  </struct>
```

There is also a mechanism for passing list type values (*arrays*) to the methods called:

- It contains a single `<data>`.
- This can contain any number of `<value> subelements`.

For example:

```
<array>
  <data>
     <value><int>15</int></value>
     <value><string>Hello</string></value>
     <value><boolean>1</boolean></value>
     <value><int>56</int></value>
  </data>
<array>
```

2.2. XML-RPC response format

The XML-RPC response will be a 200 type HTTP response (OK) provided that there have been no low-level errors. XML-RPC errors are returned as correct HTTP messages and the errors are reported in the message contents.

The format of the response is as follows:

1) The `Content-Type` must be `text/xml`.

2) The `Content-Length` field is compulsory and must be correct.

3) The body of the response must contain a single `<methodResponse>` in the following format:
   a) If the process is correct:
      – It will contain a single `<params>`, which
      – will contain a single `<param>` field, which, in turn,

– will contain a single `<value> field`.

b) If an error occurs,

- it will contain a single `<fault>`, which

- will contain a single `<value>` which is a `<struct>` with the fields
  – `faultCode` which is `<int>` and

  – `faultString` which is `<string>`.

The following is an example of a correct response:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 172
Content-Type: text/xml
Date: Fri, 24 Jul 1998 17:26:42 GMT
Server: UserLand Frontier/5.1.2

<?xml version="1.0"?> <methodResponse>
   <params>
      <param>
         <value><string>Hello<string></value>
      </param>
   </params>
</methodResponse>
```

While an incorrect response would look something like this:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 444
Content-Type: text/xml
Date: Fri, 24 Jul 1998 17:26:42 GMT
Server: UserLand Frontier/5.1.2

<?xml version="1.0"?> <methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>FaultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>FaultString</name>
          <value><int>Too many parameters</int></value>
```

```
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

2.3. Development of applications with XML-RPC

In the *java* development kit (JDK) version 1.4.2, there is no support for the development of XML-RPC applications. We therefore need to use an additional class library. In this example, we will use the class library of the Apache project, available from: `http://ws.apache.org/xmlrpc/`.

This is our example server code:

```
import java.util.Hashtable;
import org.apache.xmlrpc.WebServer;

public class JavaServer {

   public Hashtable sumAndDifference(int x, int y) {
      Hashtable result = new Hashtable();
      result.put("sum", new Integer(x + y));
      result.put("difference", new Integer(x - y));
      return result;
   }
   public static void main(String[] args) {
      try {
      WebServer server =new WebServer(9090);
      server.addHandler("sample", new JavaServer());
      } catch (Exception exception) {
      System.err.println("JavaServer:" + exception.toString());
  }
 }
}
```

As we can see, this server provides us with a method called: `sample.sumAndDifference`.

The client code for the previous service would be:

```
XmlRpcClient server = new XmlRpcClient("192.168.100.1", 9090);

 Vector params = new Vector();
 params.addElement(new Integer(5));
 params.addElement(new Integer(3));
```

```
Hashtable result =
      (Hashtable) server.execute(
                                    "sample.sumAndDifference",
                                    params);
int sum = ((Integer) result.get("sum")).intValue();
int difference = ((Integer) result.get("difference")).intValue();

System.out.println(
                     "Sum: "
                     + Integer.toString(sum)
                     + ", Difference: "
                     + Integer.toString(difference));
```

3. SOAP

SOAP standardises the exchange of messages between applications. Hence, the basic function of SOAP is to define a standard message format (based on XML) that will encapsulate communication between applications.

3.1. SOAP messages

The general form of a SOAP message is:

```
<ENVELOPE atribs>
 <HEADER atribs>
     <directives />
 </HEADER>
 <BODY attribs>
     <body />
 </BODY>
 <FAULT attribs>
     <errors />
 </FAULT>
</ENVELOPE>
```

The meaning of each part is:

- `Envelope:` this is the root element of the SOAP format.
- `Header:` this is an optional element, used to extend the basic functionalities of SOAP (security etc).
- `Body:` this is the element containing the message data. It is compulsory.
- `Fault:` in the event of error, this section will contain information on the nature of the error.

There is an extended SOAP specification called SwA (SOAP *with Attachments*) that uses MIME encoding to transport binary information.

SOAP messages have the following form, as we can see in this message of a call to a weather information web service:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>.
    <ns1:getWeather
      xmlns:ns1="http://www.uoc.edu/soap-weather"
      SOAP-ENV:encodingStyle
      =" http://www.w3.org/2001/09/soap-encoding"
      <postcode xsi:type="xsd:string">
      25001
      </postcode>
    </ns1:getWeather>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this case, the message is a service request (the example uses a JavaBeans method called `getWeather`). In the message, we can distinguish a part called `ENVELOPE`, which contains a part called `Body`.

If we look closely at the message, we can see that we are defining a call to a method, called `getWeather`. By studying the format of the call:

```
<ns1:getWeather
  xmlns:ns1="http://www.uoc.edu/soap-weather"
  SOAP-ENV:encodingStyle=
  "http://www.w3.org/2001/09/soap-encoding"
  <postcode xsi:type="xsd:string">
  25001
  </postcode>
</ns1:getWeather>
```

We can see that the method receives a parameter called `postcode` which is a string type. SOAP allows us to define parameters for all of the types defined by XSchema and provides some of its own (such as `SOAP:Array`) as well as allowing us to define new types.

Similarly, the `getWeather` method will respond with a message, also encoded in SOAP. The response message will look something like this:

```
<?xml version='1.0' encoding='UTF-8'?>
 <SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://www.w3.org/2001/09/soap-envelope"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <SOAP-ENV:Body>
     <ns1:getWeatherResponse
       xmlns:ns1="http://www.uoc.edu/soap-weather"
       SOAP-ENV:encodingStyle=" http://www.w3.org/2001/09/soap-encoding
       <weatherResponse xsi:type="xsd:string">10</weatherResponse>
```

```
      </ns1:getWeatherResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

We can see in:

```
<ns1:getWeatherResponse
  xmlns:ns1="http://www.uoc.edu/soap-weather"
  SOAP-ENV:encodingStyle=" http://www.w3.org/2001/09/soap-encoding
  <weatherResponse xsi:type="xsd:string">10</weatherResponse>
</ns1:getWeatherResponse>
```

The service is responding to our weather request with the temperature value. It is common, in fact, it is a *de facto* standard that the structure containing the response is called the same as the structure containing the call, with the addition of the *response* In our case, `getWeather` the response comes in a `getWeatherResponse`. Besides the message in XML format, SOAP defines a HTTP extension consisting of a new header for the service request. This header will be optional from version 1.2 of SOAP.

3.2. Development of SOAP applications

SOAP does not define a standard library allowing us to program applications; in fact, it only defines the message format that the applications must use, together with certain directives for communication. To demonstrate the development of a small web service, we will use one of the available SOAP libraries, Apache AXIS.

The Apache AXIS library started life as IBM's Alphaworks product. Once the work and library list were completed, IBM donated it to Apache Foundation, which continued its development under the Apache licence. To use the Apache AXIS library, you will also need Tomcat (Apache's servlet container) and Xerces (Apache's XML analyser).

We will now develop a small web service using Apache AXIS. We will use names of methods and parameters that match those of a real method that `XMethods` (`http://www.xmethods.com`) hosts on its server and can be freely accessed.

We will first define a Java interface:

```
public interface IExchange
{
   float getRate( String country1, String country2 );
}
```

As we can see, the interface (`IExchange`) is used to calculate the exchange rate between the two currencies of two countries. It is not necessary to implement the interface with AXIS, but the use of interfaces may be necessary for other SOAP APIs.

We will now look at a "fictitious" implementation of this interface:

```
public class Exchange implements IExchange
 {
 publicfloatgetRate(Stringcountry1,String country2)
    {
       System.out.println( "getRate( "+ country1 +
        ", "+ country2 + ")" );
       return 0.8551F;
    }
}
```

We can now deploy the service on Apache AXIS, which we are running. The form of deployment will depend on the libraries we use, so we need to query these. Generally, no further steps are required as these libraries are usually charged with generating the WSDL file, directory publishing, etc.

We will now look at the code required to invoke this service:

```
import java.net.*;
import java.util.*;

// Classes related to the message
import org.apache.soap.*;
// Classes related to the calls
import org.apache.soap.rpc.*;

public class Client
  {
  public static void main( String[] args ) throws Exception
    {
  // Address of Apache SOAP service
  URL url = new URL(
  "http://myserver:8080/soap/servlet/rpcrouter" );

  // Service identifier. We have given it
  // on deploying this
  String urn = "urn:demo:change";

  // Prepare service invocation
  Call call = new Call();
  call.setTargetObjectURI( urn );
```

```
    call.setMethodName( "getRate" );
    call.setEncodingStyleURI( Constants.NS_URI_SOAP_ENC );

    // Parameters
    Vector params = new Vector();
    params.addElement(
     new Parameter( "country1",
     String.class, "USA", null ) );
    params.addElement(
     new Parameter( "country2",
     String.class, "EUR", null ) );
    call.setParams( params );
    try
      {
       System.out.println(
       "invoke service\n"
       + " URL= " + url
       + "\n URN ="
       + urn );

     // invoke
     Response response = call.invoke( url, "" );
     // Fault?
     if( !response.generatedFault() )
     {
       // NO FAULT
       Parameter result = response.getReturnValue();
       System.out.println( "Result= " + result.getValue() );
     }
    else
     {
     // FAULT
     Fault f = response.getFault();
     System.err.println( "Fault= "
       + f.getFaultCode() + ", "
       + f.getFaultString() );
     }
    }
  // The call has had problems
  catch( SOAPException e )
    {
     System.err.println(
       "SOAPException= " + e.getFaultCode()
       + ", " + e.getMessage() );
    }
   }
```

```
}
```

We will now change our program to connect to an existing web service. If we visit the Xmethods web, we will see a service called `Currency Exchange Rate`. We note down the connection details:

Figure 19.

XMETHODS

RPC Profile for Service "Currency Exchange Rate"

As a convenience for those who need to manually configure basic SOAP RPC calls, we provide this page which summarizes all the necessary parameters need to configure a SOAP RPC call.

This information is derived automatically from the service WSDL file. It is a subset of what can be found in the more comprehensive WSDL Analyzer available from the service detail page.

Method Name	getRate	
Endpoint URL	http://services.xmethods.net:80/soap	
SOAPAction		
Method Namespace URI	urn:xmethods-CurrencyExchange	
Input Parameters	**country1** **country2**	string string
Output Parameters	**Result**	float

We will now modify the URL and URN appropriately in our code. We will now be able to use the web service, which is being executed on the computers of `XMethods`. If we wish to carry out more tests on the Xmethods web, we can use a list of services made public by the Internet community.

4. WSDL and UDDI

WSDL is an acronym of `Web Services Description Language`, a language based on XML that lets us describe the web services we deploy. WSDL is also used to locate these web services on the Internet.

A WSDL document is actually an XML document that describes some features of a web service, its location and the methods and parameters it supports.

4.1. Structure of a WSDL document

A WSDL document defines a web service using the following XML elements:

Table 18. XML elements

The	Meaning
`<portType>`	Operations provided by the web service
`<message>`	Messages used by the web service
`<types>`	The data types used by the web service
`<binding>`	The communication protocols used by the web service

A WSDL document therefore has a similar structure to this:

```
<definitions>
 <types>
    data types...
 </types>
 <message>
    message definitions...
 </message>
 <portType>
    operation definitions...
 </portType>
 <binding>
    protocol definitions...
 </binding>
</definitions>
```

A WSDL document can also contain other elements, such as extensions, and a service element that makes it possible to group different definitions of various web services in a single WSDL document.

4.1.1. WSDL ports

The <portType> is the XML element of WSDL that defines a web service, the operations that can be performed through this service and the messages involved. The <portType> element is similar to a library function in classical programming (or to an object-oriented programming class).

4.1.2. WSDL messages

The message element defines the data that take part in each operation. Each message can consist of one or more parts and each part can be considered similar to the parameters of a method or function call in traditional programming languages or object-oriented programming language, respectively.

4.1.3. Data types in WSDL

WSDL uses the <types> element to define the data types we will use in the web service. WSDL uses XML Schema for these definitions.

4.1.4. Links in WSDL

The <binding> element defines the message format and protocol details for each port.

This is a schematic example of what a WSDL document looks like:

```
<message name="obtReqTerm">
 <part name="param" type="xs:string"/>
</message>
<message name="obtReqTerm">
 <part name="value" type="xs:string"/>
</message>
<portType name="dictionaryTerms">
 <operation name="obtTerm">
   <input message="obtReqTerm"/>
   <output message="obtReqTerm"/>
 </operation>
</portType>
```

In this example, the portType element defines dictionaryTerms as the name of a port and obtTerm as the name of an operation. This operation has an incoming message (parameter) called obtTermReq and an outgoing message (result) called obtTermResp. The two message elements define the data types associated with the messages.

`dictionaryTerms` is the equivalent in classical programming of a function library, `obtTerm` is the equivalent of a function and `obtTermReq` and `obtTermResp` are equivalent to the incoming and outgoing parameters, respectively.

4.2. Ports

The port defines the point of connection to a web service. It can be defined as a function library or a class in classical or object-oriented programming. Each operation defined for a port can be compared to a function in traditional programming language.

4.2.1. Operation types

There are several types of operation in WSDL. The most common is request-response, though we also have:

Table 19. Types of operation in WSDL

Type	Description
One-way	The operation receives messages but does not return responses
Request-response	The operation receives a request and returns a response
Request-response	The operation can send a request and will wait for the response
Notification	The operation can send a message but does not expect a response

4.2.2. One-way operations

Here is an example of a one-way operation:

```
<message name="newValue">
 <part name="term" type="xs:string"/>
 <part name="value" type="xs:string"/>
</message>
<portType name="dictionaryTerms">
 <operation name="newTerm">
    <input name="newTerm" message="newValue"/>
 </operation>
</portType >
```

As we can see, a one-way operation has been defined in this example called `newTerm`. This operation can be used to enter new terms in our dictionary. To do this, we use an incoming message called `newValue` with the parameters `term` and `value`. However, we have not defined an output for the operation.

4.2.3. Request-response operations

We will now look at an example of a request-response operation:

```
<message name="obtReqTerm">
 <part name="param" type="xs:string"/>
</message>
<message name="obtReqTerm">
 <part name="value" type="xs:string"/>
</message>
<portType name="dictionaryTerms">
 <operation name="obtTerm">
      <input message="obtReqTerm"/>
      <output message="obtReqTerm"/>
 </operation>
</portType>
```

In this example, as we saw earlier, we can see that two messages are defined, one incoming and one outgoing, for the `obtTerm`.

4.3. Links

WSDL links (*bindings*) allow us to define message and protocol formats for web services. An example of a link for a request-response operation for SOAP might be:

```
<message name="obtReqTerm">
  <part name="param" type="xs:string"/>
</message>

<message name="obtReqTerm">
  <part name="value" type="xs:string"/>
</message>

<portType name="dictionaryTerms">
  <operation name="obtTerm">
    <input message="obtReqTerm"/>
    <output message="obtReqTerm"/>
  </operation>
</portType>

<binding type="dictionaryTerms" name="tD">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
<operation>
    <soap:operation
    soapAction="http://uoc.edu/obtTerm"/>
```

```
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
    </operation>
  </binding>
```

The `<binding>` has two attributes: the attribute `name` and the attribute `type`. With the `name` attribute (we can use any name, it does not necessarily need to have anything to do with the name used in the definition of *port*), we define the name of the link and the attribute `type` indicates the port of the link; in this case, the port is `dictionaryTerms`.

The `soap:binding` element has two attributes `style` and `transport`.

The `style` attribute can be `rpc` or `document`. Our example used `document`. The `transport` attribute defines which protocol SOAP to use; in this case, HTTP.

The `operation` element defines each of the operations provided by the port. For each one, we need to specify the corresponding SOAP action. We must also specify how to encode the `input` and `output`). In our case, the encoding is `"literal"`.

4.4. UDDI

UDDI is the acronym of *Universal Description, Discovery and Integration*, a directory service where companies and users can publish and search for web services. UDDI is a standard and independent platform structure for describing these web services, searching services, etc.

UDDI is built on the Internet standards of the W3C and the IETF (Internet *Engineering Task Force*), like XML, HTTP, etc. To describe the interfaces to the web services, it uses the WSDL language described above and for its cross-platform programming needs, it uses SOAP, which allows for full interoperability.

UDDI is a major breakthrough for the development of the Internet as a platform for information technology business. Before its development, there was no standard that allowed the location or publicising of information processing services. Nor was there a method that could integrate the diverse information systems of companies.

Some of the benefits of using UDDI are:

- It allows us to discover the right business (or service) from the thousands currently registered on some servers via the Internet.
- It defines how to interact with the chosen service, once located.
- It allows us to reach new customers and facilitates and simplifies access to existing customers.
- It extends the potential market of users of our business methods and services.
- It automatically describes the services and components or business methods (or it is automatable) in a secure, open and simple environment.

4.4.1. Using UDDI

We will look at a possible example of how UDDI could be used to solve a hypothetical case that will clearly demonstrate the advantages of UDDI.

> If the hotel industry published an UDDI standard for room booking, the different hotel chains could register their services in an UDDI directory. Travel agencies could then search the UDDI directory to find the reservations interface of the hotel chain. Once it found the interface, the travel agency could then make the booking, since this interface would be described in a known standard.

4.4.2. Programming in UDDI

There are two APIs for developing applications with UDDI. Most existing implementations for developing in UDDI are commercial but there are two free software implementations: pUDDing (`http://www.opensorcerer.org/`) and jUDDI (`http://ws.apache.org/juddi/`). The two APIs mentioned are the query API (*inquiry*) and the publication API (*publish*). The *inquiry* API searches for information on the services offered by a company, the specification of the latter and information on what to do in the event of an error. All read accesses to UDDI records use API *inquiry*. This does not require authentication, so HTTP is used for access.

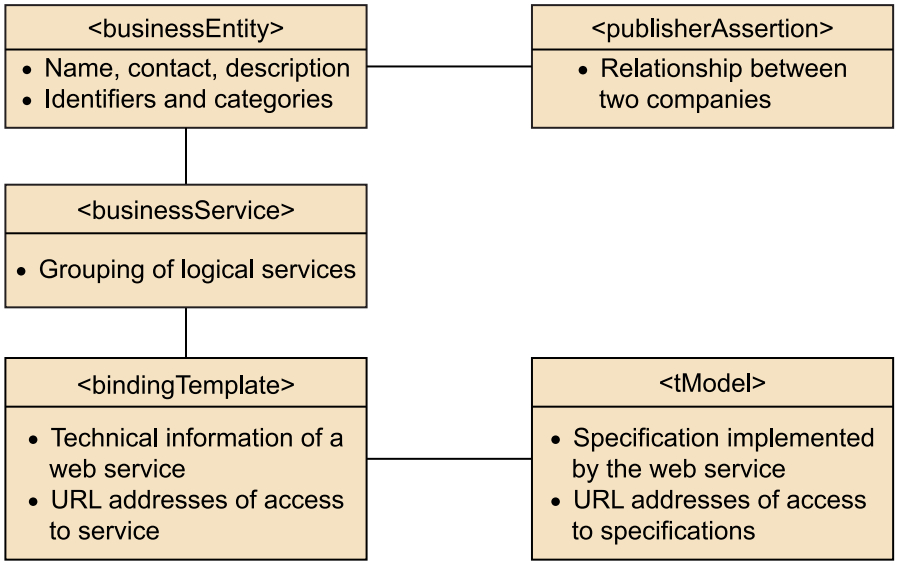
The other API, publication, is used for the creation, registration, updating, etc., of information located in an UDDI record. All of the functions in this API require authenticated access to an UDDI record, so HTTPS is used instead of HTTP.

Both APIs were designed to be straightforward. All operations are synchronous and stateless.

To aid understanding of the structure of APIs, the following schema demonstrates the relationship between the diverse UDDI data structures.

We can see a simple example of how a client might send an UDDI request to find a business.

Figure 20.



The request we would send to search for the business would be:

```
<uddi:find_business generic="2.0" maxRows="10">
 <uddi:name>
     Software company
 </uddi:name>
</uddi:find_business>
```

As we can see, the message we send is very simple and only indicates the name of the business. The response we would receive to such a request would be:

Note

UUID.

The instances of data structures are identified by a universally unique identifier known as UUID. UUIDs are assigned the first time that the structure is inserted in the record. They are hex strings structured according to the ISO/IEC 11578:1996 standard, meaning that their uniqueness is guaranteed.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
    <businessList xmlns="urn:uddi-org:api_v2"
      generic="2.0" operator="SYSTINET">
      <businessInfos>
       <businessInfo
       businessKey="132befd0-d09a-b788-ad82-987878dead98">
       <name xml:lang="en">
        Software company
       </name>
       <description xml:lang="en">
            A company that develops web services software
```

```
        </description>
        <serviceInfos>
         <serviceInfo
          serviceKey="23846ac0-dd99-22e3-80a9-801eef988989"
          businessKey="9a26b6e0-c15f-11d5-85a3-801eef208714">
                <name xml:lang="en">
                      It sends
                </name>
         </serviceInfo>
        </serviceInfos>
       </businessInfo>
      </businessInfos>
     </businessList>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5. Security

The emergence of web services has given rise to certain issues that had previously not been taken into consideration. The high degree of interoperability and ease of connection and data exchange have meant that web services have encouraged the appearance of new data security risks that had not previously existed:

- The clarity of XML encoding (allowing it to be easily understood) makes light work of *hacking*, etc.

- We now need a standard for interoperability, since the various participants in a transaction using web services may not all use the same software.

- The very nature of SOAP/WS, which facilitates the appearance of sporadic connections between machines, the appearance of intermediaries (*proxies*, etc.), makes it difficult to control who has access to the data sent.

- Applications are now available to everybody; with common protocols like http, we have universalised access to our applications. Even with standards such as WSDL and UDDI, there are application directories where anybody can discover the applications published, their methods, etc.

As a result, the definition of security mechanisms designed specifically for web services that go beyond SSL is becoming increasingly important in the definition of web services.

5.1. Incorporating security mechanisms in XML

Current standards, still in development at the time of writing, afford some basic security features:

- Digital signature, used to check the identity of the sender of a message and the integrity of the latter.
- Authentication, allowing us to check identities.
- No repudiation, used to prevent a sender from denying that it is the origin of the message.

For this, we will need to have an operative public key infrastructure (PKI).

5.1.1. Digital signature

A digital signature is the mathematical equivalent of a handwritten signature. It consists of a code added to a block of information that can be used to check the origin and integrity of the latter.

The XML digital signature specification defines an optional element that facilitates the inclusion of a digital signature in an XML document.

An example of a document signed with XML is:

```
<Grades xmlns="urn:uocedu">
   <student id="1293">
      <name>Joan Oro</name>
        <city>Lleida</city>
   </student>
   <grades>
      <grade id="MOLBIO">
        <subject>Molecular Biology</subject>
        <score>9.56</score>
        <comment>Excellent work</comment>
      </grade>
   </grades>
   <Signature Id="EnvelopedSig"
      xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo Id="EnvelopedSig.SigInfo">
      <CanonicalizationMethod
Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
        <SignatureMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <Reference Id="EnvelopedSig.Ref" URI="">
          <Transforms>
          <Transform
Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
           </Transforms>
           <DigestMethod
Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
           <DigestValue>
           yHIsORnxE3nAObbjMKVo1qEbToQ=
           </DigestValue>
        </Reference>
      </SignedInfo>
      <SignatureValue Id="EnvelopedSig.SigValue">
GqWAmNzBCXrogn0BlC2VJYA8CS7gu9xH/XVWFa08eY9HqVnr
fU6Eh5Ig6wlcvj4RrpxnNklBnOuvvJCKqllQy4e76Tduvq/N
8kVd0SkYf2QZAC+j1IqUPFQe8CNA0CfUrHZdiS4TDDVv4sf0
V1c6UBj7zT7leCQxAdgpOg/2Cxc=
```

```
      </SignatureValue>
      <KeyInfo Id="EnvelopedSig.KeyInfo">
       <KeyValue>
          <RSAKeyValue>
      <Modulus>
AIvPY8i2eRs9C5FRc61PAOtQ5fM+g3R1Yr6mJVd5zFrRRrJz B/
awFLXb73kSlWqHao+3nxuF38rRkqiQ0HmqgsoKgWChXmLu
Q5RqKJi1qxOG+WoTvdYY/KB2q9mTDj0X8+OGlkSCZPRTkGIK
jD7rw4Vvml7nKlqWg/NhCLWCQFWZ
      </Modulus>
      <Exponent>
      AQAB
      </Exponent>
    </RSAKeyValue>
   </KeyValue>
  </KeyInfo>
 </Signature>
</Notas>
```

As this example illustrates, the digital signature is applied by encrypting a *digest* of the XML message with a private key. The digest is usually the result of applying a mathematical function called *hash* to the XML document we wish to sign. This encrypted digest is included along with a key to be able to verify the operation with the XML message. As we are the only ones with the key to encrypt it, we are clearly the only senders of the message. Moreover, since the result of the *digest* depends on every single byte composing the message, it is clear that the latter cannot have been altered on its way.

With XML, given its particular nature and the processing usually undergone by a web service message during its existence and taking into account the *parsing*, etc., that can alter its form (changing a single space can produce a different *digest*), implementation of the *digest* needs to take into account these valid alterations. Therefore, before calculating the *digest* we need to perform a process to ensure that none of the possible changes that our XML may undergo, which do not really affect the content (deleting spaces, summarising *tags* without content, etc.) do not cause non-validation of the *digest*. This process, called W3C-XML-Canonicalization (xml-c14n), includes the following rules, which a document must comply with before the *digest*:

- Encoding must be UTF-8.
- Standardisation of line breaks (to ASCII 10).
- Standardisation of attributes.
- Empty elements converted into start-end pairs.
- Standardisation of meaningless spaces.
- Elimination of superfluous namespace declarations.
- Default attributes are listed.

- Lexicographic reordering of declarations and attributes.

5.1.2. Data encryption

Besides their digital signature capabilities, the new security standards of XML have encrypting features that can be used to hide parts or all of the XML document from the elements of the intermediate process.

An example will illustrate how the previous document would look if we encrypted the data on student grades as well as signing it.

```
<Grades xmlns="urn:uocedu">
   <student id="1293">
      <name>Joan Oro</name>
      <city>Lleida</city>
   </student>
   <grades>
   <EncryptedData Id="ED" Nonce="16"
     Type=http://www.w3.org/2001/04/xmlenc#Content
     xmlns="http://www.w3.org/2001/04/xmlenc#"
     xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
     <EncryptionMethod
     Algorithm ="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
        <ds:KeyInfo>
             <ds:KeyName>uoc</ds:KeyName>
        </ds:KeyInfo>
        <CipherData>
            <CipherValue>
dRDdYjYs11jW5EDy0lucPkWsBB3NmK0AFNxvFjfeUKxP75
cx7KP0PB3BjXPg14kJv74i7F00XZ5WhqOISswIkdN/pIVe
qRZWqOVjFA8izR6wqOb7UCpH+weoGt0UFOEkIDGbemm23e
u812Ob5eYVL8n/DtO81OhYeCXksSMGUZiUNj/tfBCAjvqG
2jlslQM6n4jJ3QNaR4+B2RisOD6Ln+x2UtNu2J7wIYmlUe
7mSgZiJ5eHym8EpkE4vjmr2oCWwTUu91xcayZtbEpOFVFs
6A==
        </CipherValue>
       </CipherData>
   </EncryptedData>
   </grades>
   <Signature Id="EnvelopedSig"
      xmlns="http://www.w3.org/2000/09/xmldsig#">
         .....
   </Signature>
</Notas>
```

Use and maintenance

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Carles Mateu

PID_00148403

UOC
Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Configuring security options

1.1. User authentication

User authentication is used in Apache to ensure that the people viewing or accessing certain resources are the people we want them to be or who know a certain access "key". In Apache, we can define which resources require authentication of the user accessing them and we can identify the mechanism used to authenticate this user.

Note

Authentication. The weakness of authentication is always the human factor. We commonly "lend" our usernames and passwords to friends to *make life* easier.

The names of user authentication modules usually begin with `mod_auth_` followed by an abbreviation of the search mechanism and user verification, such as `ldap, md5`, etc.

We will now explain how to configure Apache to require user authentication for accessing a given directory:

```
<Directory /web/www.uoc.edu/docs/secret>
   AuthType Basic
   AuthName "Restricted"
   AuthUserFile /home/carlesm/apache/passwd/passwords
   Require user carlesm
</Directory>
```

With this configuration, we are telling Apache to only display the specified content to the user `carlesm` after verifying his identity with a password. The file containing the usernames and passwords is specified with `AuthUserFile`.

This configuration uses the `mod_auth` module, which provides basic authentication based on the use of a plain text file in which we will store a list of users and encrypted access words (`passwords`). Apache also has a tool for administrating this password file (`htpasswd`). To create a file like this, which will give a password to our user, we need to execute:

```
# htpasswd -c /home/carlesm/apache/passwd/passwords carlesm
New password: <password>
Re-type new password: <password>
Adding password for user carlesm
```

Where we see `<password>` we need to enter our password.

Staying with the configuration we used to illustrate authentication, we can see that, besides the required password file, we tell Apache the name of the authentication field; in our case, "Restricted". The client browser will now always send the password that we provide to requests that are identified (the identification is known as `Realm`) as "Restricted". This allows us to share passwords among several resources, doing away with the need to type in countless passwords.

1.1.1. Other authentication modules

There are several authentication modules for Apache. These modules are used to select multiple sources to verify users and passwords. In our example, we used the basic `mod_auth` module, which produces a plain text file of users and their passwords using the directive `AuthUserFile` to indicate the file.

We also have the following modules:

- `mod_auth_dbm`: allows the use of Berkeley type databases (DBM) to store usernames and passwords.
- `mod_auth_digest`: similar to `mod_auth`, but it allows the use of DigestMD5 to encrypt passwords.
- `mod_auth_ldap`: allows us to use an LDAP directory to store usernames and passwords.
- `mod_auth_samba`: allows us to use a Samba domain server (SMB/-CIFS) for user verification.
- `mod_auth_mysql` and `mod_auth_pgsql`: used to store usernames and passwords in an SQL database such as `mysql` or `postgresql`.
- `mod_auth_radius`: allows us to use a RADIUS server to verify usernames and passwords.

1.2. Communications security

Another of Apache's features is the use of tight cryptography to encrypt and sign communications. For this, it has a module called `mod_ssl` that se ves as an interface to the cryptographic library OpenSSL, thus providing Apache with the mechanisms for using secure connections based on the SSL/TLS protocol.

To use the cryptographic module, we need to install `mod_ssl` (we will need OpenSSL installed on our system). Once installed, we need to go to the Apache configuration file to configure the operation of `mod_ssl`. To do so, we will specify the SSL requirements on a virtual server or in a directory:

```
# Enable SSL
SSLEngine on
# Specify Protocol
SSLProtocol all -SSLv3
# Specify cryptographic algorithms
SSLCipherSuite ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL
# Password files and certificates
SSLCertificateFile /etc/httpd/ssl.key/server.crt
SSLCertificateKeyFile /etc/httpd/ssl.key/server.key
```

As we can see, to enable a secure server with SSL, we need a pair of public/private keys and a digital certificate. We can use the programs provided with the OpenSSL library to create certificates and key pairs.

2. Configuring load balancing

Apache has several options for the use of load balancing solutions. The complexity of the solutions varies greatly and they have very wide-ranging features, which means that the situations they are designed for are very different.

2.1. DNS balancing

The simplest load balancing solution is to install a group of systems with web servers and ensure that they all have access to the files that make up our website. We can then programme the name server (the DNS) to return a different address (from one of the machines) each time it is prompted for the name of our web server.

Note

The DNS (*Domain Name Server*) is the service that "resolves" (converts) the names of servers to IP addresses at the request of the clients. For example, the DNS is in charge of converting `www.uoc.edu` to 213.73.40.217.

We have seven computers to configure our web server (www.uoc.edu). To each, we assign an IP address and a name (we are showing the data here in BIND format, the most common DNS software):

`www0`	`IN`	`A`	`1.2.3.1`
`www1`	`IN`	`A`	`1.2.3.2`
`www2`	`IN`	`A`	`1.2.3.3`
`www3`	`IN`	`A`	`1.2.3.4`
`www4`	`IN`	`A`	`1.2.3.5`
`www5`	`IN`	`A`	`1.2.3.6`
`www6`	`IN`	`A`	`1.2.3.7`

We will also add the following entry:

`www`	`IN`	`CNAME`	`www0.uoc.edu.`
	`IN`	`CNAME`	`www1.uoc.edu.`
	`IN`	`CNAME`	`www2.uoc.edu.`
	`IN`	`CNAME`	`www3.uoc.edu.`
	`IN`	`CNAME`	`www4.uoc.edu.`
	`IN`	`CNAME`	`www5.uoc.edu.`
	`IN`	`CNAME`	`www6.uoc.edu.`

If we now ask the name server to tell us the address of `www.uoc.edu`, it will respond each time with all of the addresses of the seven machines, but in a different order. This means that the client requests will be alternated between the different server machines.

However, this balancing schema is not ideal. On the one hand, it does not take into account whether any of the machines are down. On the other, the global DNS system requires DNS servers, such as that of our Internet provider, to create a *cache* of the addresses, so some DNS servers will "learn" one of the addresses and always resolve this same address. In the long run, however, this is not a problem because the total queries are split among the different servers.

There are other DNS server programs that offer more effective solutions for real load balancing between servers as they use auxiliary tools to check the status of the different servers.

2.2. Proxy balancing

Another balancing option in Apache is to use a module called `mod_rewrite` which redistributes requests among the different servers.

To do this, we must modify the name server so that our website, called `www.uoc.edu`, corresponds to just one of the machines (`www.uoc.edu`).

`www`	`IN`	`CNAME`	`www0.uoc.edu.`

We will then convert `www0.uoc.edu` into a dedicated `proxy` server. This machine will then resend all of the requests it receives to one of the other five machines. To do this, we will need to configure the `mod_rewrite` module to redirect the balanced requests to the other machines. We use a program (written in Perl this time) providing `mod_rewrite` with the server to which the request needs to be redirected.

```
RewriteEngine on
RewriteMap     lb       prg:/programs/lb.pl
RewriteRule    ^/(.+)$   ${lb:$1}             [P,L]
```

This program is as follows:

```
#!/usr/bin/perl
##
## lb.pl -- Load balancing
##

$ = 1;
```

```
$name = "www";       # Name base
$first = 1;          # First server (we will begin with www1)
$last = 5;           # Last server (www5).
$domain = "uoc.edu"; # Domain

$cnt = 0;
while (<STDIN>)
{
   $cnt = (($cnt+1) % ($ult+1-$prim));
   $server = sprintf(" %s %d. %s", $name, $qnt+$first, $domain);
   print "http://$server/$_";
}
```

This solution has the slight disadvantage that all requests pass through one machine (`www0.uoc.edu`). Although this machine does not perform heavyweight processes (CGI, etc.) and only redirects requests (much faster and more lightweight), in the event of overloading, the machine can become saturated. To solve this, we can use *hardware* solutions that carry out the same tasks as the option we have described here with `mod_rewrite`. These *hardware* solutions, however, are usually very expensive.

2.3. mod_backhand balancing

When DNS balancing or balancing based on a *hardware* resource is insufficient, we can use a much more sophisticated resource allocation mechanism. This consists of an Apache module called `mod_backhand` used to redirect requests.

HTTP from one web server to another, depending on the load and use of resources on the servers.

`mod_backhand` has a major disadvantage: at the time of writing, the version for Apache 2 was still unavailable, so it could only be used with Apache 1.2/1.3.

The configuration of `mod_backhand` is extremely straightforward. The module needs to be installed on each of the Apache servers in the machine cluster. We must then configure the different Apache servers to communicate with one another for synchronising purposes. We will use a *multicast* address (239.255.221.20) to communicate. The configuration will be similar to this:

```
<IfModule mod_backhand.c>
 # Working directory of mod_backhand. Make sure
 # that the permissions allowing us to write.UnixSocketDir /var/backhand/backhand
 #   We will use IP Multicast with TTL=1 to report statistics
 #   We could use Broadcast.
   MulticastStats 239.255.221.20:4445,1
 #   We will accept notifications from our servers:
   AcceptStats 1.2.3.0/24
```

```
</IfModule>
```

We can configure mod_backhand to display the operating status on a page:

```
<Location "/backhand/">
     SetHandler backhand-handler
</Location>
```

To enable mod_backhand for a directory, we can configure it inside an Apache directory module. We must then indicate that the cgi-bin directo y containing the shared CGI files, those with the biggest system requirements, is distributed among the cluster machines:

```
<Directory "/var/backhand/cgi-bin">
     AllowOverride None
     Options None
     Order allow,deny
     Allow from all
     Backhand byAge
     Backhand byRandom
     Backhand byLogWindow
     Backhand byLoad
</Directory>
```

Example

Example of a criterion for re-sorting: by least CPU load.

As we can see, mod_backhand is configured with Backhand directives that we shall call *candidate functions*. The operation of mod_backhand is: when we need to serve a resource from the specified directory, /var/backhand/cgi-bin, we pass mod_backhand the list of candidate servers (initially all those configured with mod_backhand). This passes through each of the specified candidate functions (byAge, byRandom etc). These functions eliminate the servers they do not consider appropriate from the list or they rearrange the list using the corresponding criterion.

Then, after evaluating all of the candidate functions, mod_backhand resends the request to the server that tops the list of candidates.

mod_backhand has numerous default candidate functions and also includes the tools so that we can build new functions ourselves.

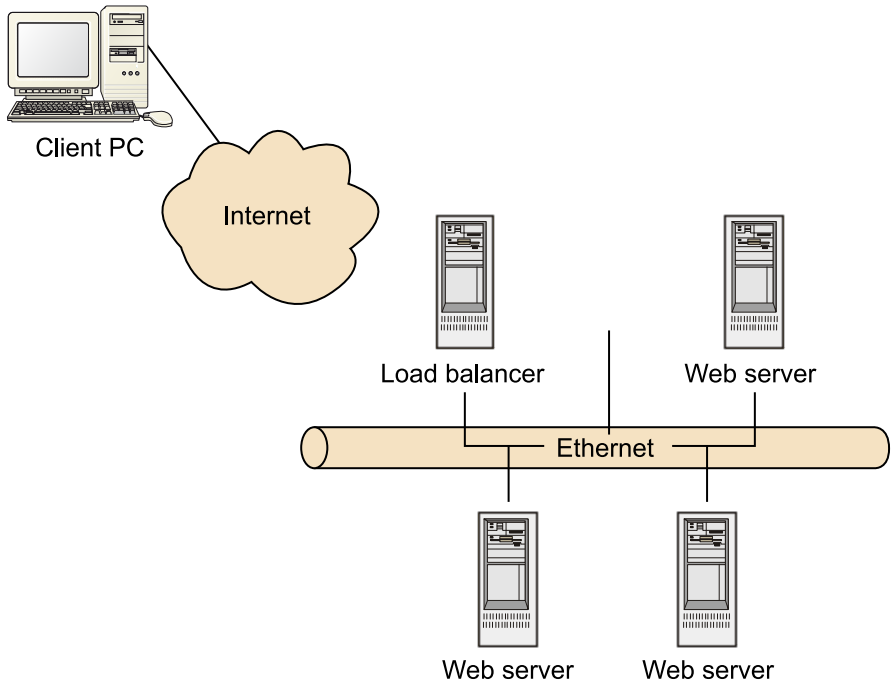
2.4. Balancing with LVS

Linux servers have a high-performance tool for load balancing and high availability configurations. This tool, or group of tools, is based on the LVS project (Linux *Virtual Server*) and uses NAT (*Network Address Takeover*) and IP/Port *forwarding* mechanisms.

Configurations based on LVS are usually complex and expensive. They usually involve several servers because they are normally aimed more at generating high availability than at facilitating high performance (load balancing) and require at least a system for load balancing.

The minimum configuration of a system with LVS is similar to this:

Figure 21.

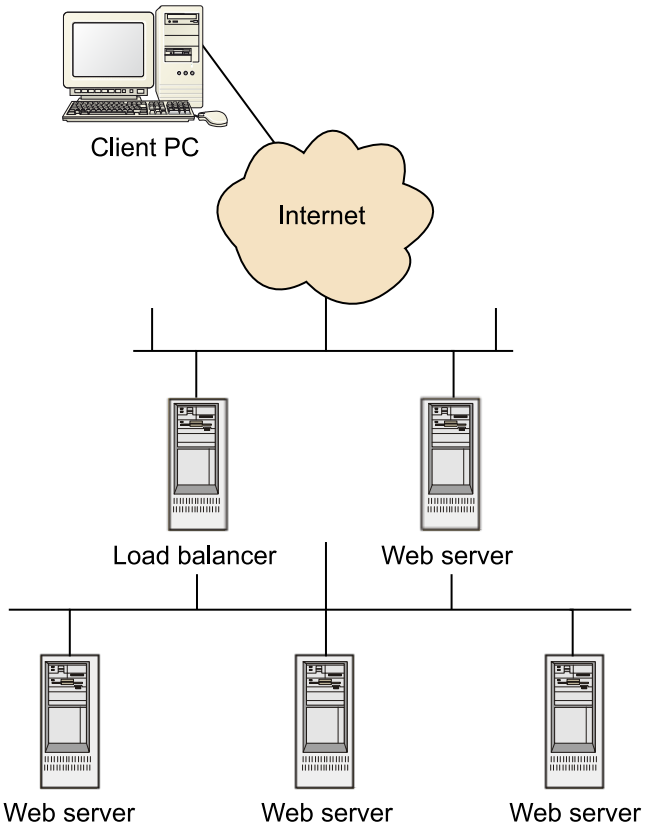


The main difference between this approach and the use of `mod_rewrite` for load balancing is that in this case balancing is done at IP level; in other words, the balancer does not have a copy of Apache running to receive requests and resend them. Instead, they are resent at IP level, generating a performance hundreds of times superior.

LVS is very similar, in fact it is virtually identical, to *hardware* solutions such as Cisco, etc. The advantage of using LVS is that, because it is based on Linux, we can use standard (cheaper) computers and even "reuse" obsolete computers, servers, etc. The performance of a system configured to perform balancing tasks is so high that a *Pentium III 1Ghz* computer can be used to saturate links of 100 Mbps. This allows us to reduce the costs of our clusters and extend the useful life of our systems.

Solutions based on LVS can also be used in two ways: as load balancing solutions for increasing the efficiency of our web servers and as systems with high availability and resistance to failure. A typical configuration of this type would be:

Figure 22.



In this configuration, the two balancing systems would monitor each other, so that if the one acting as the primary system failed, the other would adopt this role. They would also distribute the load among different web servers in the cluster, monitoring their operation and, if failures were detected in one, they would divert the requests to the other servers.

Something to take into account when implementing a balancing solution is session management. If the web servers or a servlet container serving these web servers has a session management system, database connections, etc., we need to be very cautious about implementing a load balancing solution. We need to study our application to ensure that requests from a single client to our application (an on-line store, for instance) do not become problematic if these requests are attended alternately by different web servers.

Many of these solutions, like LVS, offer methods for ensuring a minimum of persistence in connections, i.e. that all requests from the same client are assigned to the same server for the approximate duration of a session. These mechanisms, however, are not infallible because they are only network mechanisms, i.e. mechanisms operating at IP level.

2.5. Other load balancing solutions

There are other approaches to load balancing on Apache servers that are not general but they may represent the perfect solution for a specific problem.

2.5.1. OpenMOSIX or balancing and migration of processes

For Linux servers, there is a tool called OpenMOSIX, which consists of a series of modifications in the operating system kernel, allowing us to distribute the execution of certain processes among several machines in a group configured for this purpose (to the machine with the least load at this time).

This solution does not really balance the load because the processes are simply "migrated" rather than being distributed. Moreover, each process is executed on a single machine. However, it has been shown to be a good solution when the performance problems of the web application are not caused by Apache but by the length of time needed to execute a CGI or external program. In these cases, a solution like OpenMOSIX, which migrates the specific process taking too long to a machine with a lighter load, thus unloading the main server, could be a very valid one. Its advantages include the fact that it has none of the disadvantages of other solutions seen previously (persistence control etc).

2.5.2. Handling URLs and links

There is an experimental solution, available only for Apache 1.2/1.3, that can be used for efficiently scaling services between machines.

This solution is based on a hypothesis that is usually verified in most cases, indicating that users usually enter web servers by means of a set series of entry points.

The solution, then, can handle the links that appear in the documents, rewriting them to point to one of the cluster servers. It can also replicate the document by sending it to the latter.

This module, called DC-Apache (*Distributed Cooperative* Apache *Web Server*, http://www.cs.arizona.edu/dc-apache/), allows us to add to our web server a series of machines to facilitate the subsequent diversion of most of our requests to these secondary machines.

In our example, we would configure the main server as follows:

```
...
DocumentRoot "/web/www.uoc.edu/docs/"
...
  <IfModule mod_dca.c>
  SetHandler DCA-handler
  # Directory to collect replicated documents
  ImportPath "/home/www/&#732;migrate"

  # Directory with the documents we want
```

```
  # to distribute in the cluster
  ExportPath "/web/www.uoc.edu/docs/"

  # Support servers
  Backend 1.2.3.2:80
  Backend 1.2.3.3:80
  Backend 1.2.3.4:80
  Backend 1.2.3.5:80
  Backend 1.2.3.6:80
  Backend 1.2.3.7:80
</IfModule>
```

For secondary servers, if they do not have a local document to share in the cluster, the configuration would be:

```
<IfModule mod_dca.c>
   SetHandler dca-handler
   ImportPath /var/dcamigrated
   # Disk quota available for migrating
   # e.g. 250 MB
   DiskQuota 250000000
</IfModule>
```

2.5.3. Manual load distribution

There is obviously a manual solution for load balancing which, due to its simplicity and surprisingly good results, is one of the most used. It consists of manually dividing the contents of our website among several servers and adapting the URLs of our website to match.

For example, we can put all of the web images on a separate server to the one with the documents and CGIs, and use the following on all pages like `IMG_SRC` :

```
<IMG SRC="http://images.uoc.edu/logos/logo.gif">
```

This will divert the required traffic for logos, etc., to a specific server, releasing the main server from these tasks.

3. Configuring a *caching proxy* with Apache

One of the possible uses of Apache is to operate it as a *caching proxy*. The server can operate both as a *forward proxy* and a *reverse proxy*, with *proxy* capabilities for both HTTP (protocol versions 0.9, 1.0 and 1.1) and for FTP and CONNECT (necessary for SSL). The module that implements the *proxy* functionalities for Apache is very straightforward because it offloads part of its capabilities on other modules.

For example, the storage capacity (the *cache*) is delegated to the `mod_cache`, etc.

3.1. Introduction to the *proxy*

A *proxy* is a representative server located between the client making the request and the server that must resolve it. There are countless situations advising the use of *proxy* servers between our clients and the servers that need to attend the requests.

For example, the use of *proxy* is recommended to speed up browsing (this is the case of *forward proxy caches*) to control which addresses are accessed (*forward proxy*) to accelerate the response of a web server (*reverse proxy*), etc.

3.1.1. The *forward proxy*

A *forward proxy* is an intermediate server that sits between the client and the servers that need to attend to the request.

For a client to receive content from a server, it must make the request to the *proxy* server, telling it that it wants to obtain the content and the request that it wants to be met. Then, the *proxy* makes the request on behalf of the client and, once resolved, delivers the results to the latter. In the case of *forward proxy*, clients must generally be specifically configured to use the services of the *proxy*.

A typical use of *forward proxy* is to provide Internet access to the clients of an internal network isolated from the latter for security reasons and only allowing access through the *proxy*, which is easier to secure. Another very common use is to *cache* pages. This time, the *proxy* server builds up the pages visited by clients locally. If a client requests a visited page, the *proxy* server can serve it directly from its local store, thus saving on bandwidth and reducing the time to access pages that are visited frequently (clearly, *caching proxy* must have an information storage/discarding policy).

One effect of the use of a *proxy* is that, for the servers, access to resources will appear to come from our *proxy* system rather than from the client's real address. It is therefore essential to configure *proxy* servers very securely before connecting them to the Internet, as they could be used as a hiding system by malicious users.

3.1.2. The *reverse proxy*

A *reverse proxy*, unlike *forward proxy*, are located in front of a server and only control access to this server. For clients, the *reverse proxy* will look like a normal web server and will not require any specific configuration.

The client makes its requests to the *reverse proxy* server, which decides where to redirect these requests and, once resolved, returns the result as if the *reverse proxy* had been the source of the content.

One typical use of a *reverse proxy* is to filter and control access by Internet users to a server that we want to remain very isolated. Other uses of *reverse proxy* include balancing loads between servers and providing *cache* mechanisms for slower servers. They can also be used to unify the URL addresses of different servers under a single URL namespace: that of the *proxy server*.

3.2. Configuring a *forward proxy*

To configure a *forward proxy* we must first tell Apache that it needs to use certain modules. These are:

- `mod_proxy:` the module that will provide the *proxy*.
- `mod_proxy_http:` services for HTTP protocols *proxy*.
- `mod_proxy_ftp:` services for FTP protocol *proxy*.
- `mod_proxy_connect:` services for SSL *proxy*.
- `mod_cache:` cache *module*.
- `mod_disk_cache:` auxiliary disk *cache* module.
- `mod_mem_cache:` auxiliary memory *cache* module.
- `mod_ssl:` auxiliary SSL connections module.

After loading the necessary modules, we will configure the *proxy*. The first configuration requires us to indicate that Apache will act as a *forward proxy*.

```
LoadModule proxy_module modules/mod_proxy.so
```

```
<IfModule mod_proxy.c>
     LoadModule http_proxy_module modules/mod_proxy_http.so
     ProxyRequests On
     ProxyVía On

     <Proxy *>
          Order deny,allow
          Deny from all
          Allow from 172.16.0.0/16
     </Proxy>
</IfModule>
```

This example shows how easy it is to configure the `mod_proxy`. We must first enable the handling of *proxy* requests by Apache. The `ProxyRequests` directive, when enabled, indicates that Apache must act as a *forward proxy*. The second directive, `ProxyVia` tells the module that it must mark the requests made with the field `Via:` aimed at controlling the flow of requests in a chain of *proxies*.

The `Proxy` block is used to configure the security restrictions of the *proxy*. In this case, we only allow use of the *proxy* server by all of the machines in our internal network (172.16.0.0). Here, we can use all of Apache's access control directives.

After configuring the *proxy* server, we will need to configure the Apache *cache* module. To do so, we need to define a disk storage of 256 Mbytes:

```
Sample httpd.conf
#
# Sample cache Configuration
#
LoadModule cache_module modules/mod_cache.so
<IfModule mod_cache.c>
     LoadModule disk_cache_module modules/mod_disk_cache.so
     <IfModule mod_disk_cache.c>
          CacheRoot /var/cache
          CacheSize 256
          CacheEnable disk /
          CacheDirLevels 5
          CacheDirLength 3
     </IfModule>
</IfModule>
```

3.3. Configuring a *reverse proxy*

To configure Apache as a *reverse proxy*, we need to load the same modules as for when we use it as a *forward proxy*. One difference, however, will be that the configuration of the security directives will be much less critical, as we will explicitly tell the *proxy* which servers it can access.

The configuration of a *reverse proxy* for accessing an internal server we have on another TCP port and mapping it to a subdirectory of our web space would be:

```
ProxyRequests Off
     <Proxy *>
          Order deny,allow
          Allow from all
     </Proxy>
ProxyPass /internal               http://internal.uoc.edu:8181/
ProxyPassReverse /internal        http://internal.uoc.edu:8181/
```

The `ProxyPass` directive tells Apache that all of the requests addressed to the specified URL (`/internal`) must be converted internally into requests to the specified target server. The `ProxyPassReverse` directive also indicates that the response received from the specified target server will be rewritten as coming from the *reverse proxies* server and originating from the URL space indicated (`/internal`).

As part of the configuration of *reverse proxies*, we can add the features of `mod_cache` to store requests locally, just as we would for a *forward proxy*.

3.4. Other configuration directives

The `mod_proxy` module also has other configuration directives that we can use to adapt the server operation.

3.4.1. ProxyRemote/ProxyRemoteMatch directive

The `ProxyRemote` directive is used to take received requests and redirect them to other *proxy* servers, such as

```
ProxyRemote http://www.uoc.edu/manuals/ http://manuals.uoc.edu:8000
ProxyRemote * http://fastserver.com
```

These two configurations will redirect all requests matching `http://www.uoc.edu/manuals/` to another server and all other requests to a specific server.

There is a variant of `ProxyRemote` called `ProxyRemoteMatch` that allows the use of regular expressions to indicate the URL to be checked.

3.4.2. ProxyPreserveHost directive

This directive is used to tell Apache that the requests must maintain the `host` field instead of replacing it with that indicated in the configuration of `ProxyPass`. This is only necessary when the server hidden by a *reverse proxy* is a name-based *virtual host* .

3.4.3. NoProxy directive

This directive is used to exclude from `mod_proxy` processing any computer, domain, address, etc., we do not require.

4. Other Apache modules

Apache has many additional modules that can be included on our server. Some of these modules are distributed with the official Apache package. To use them, simply check that they are on the server and enable them in the configuration files. Other modules contributed by hundreds of developers are not supplied officially and will need to be downloaded separately and installed on the server.

4.1. mod_actions

This module incorporates methods for executing actions based on the file type requested. A simple configuration for this is:

```
Action image/gif/cgi-bin/images.cgi
```

Bear in mind that the URL and file name requested are passed to the program that provides the actions through environment variables: CGI PATH_INFO and PATH_TRANSLATED.

4.2. mod_alias

This is used to define URL areas that are located on the server disk outside the area defined by `DocumentRoot`. It provides a number of directives, including `Alias`, for defining new directories, `Redirect` for defining redirections and `ScriptAlias` for defining new directories containing CGIs.

```
Alias /images /web/images
<Directory /web/images>
     Order allow,deny
     Allow from all
</Directory>

Redirect permanent /manuals http://manuals.uoc.edu/
Redirect 303 /document http://www.uoc.edu/underway.html.
```

4.3. mod_auth, mod_auth_dbm, mod_auth_digest, mod_auth_ldap

Together with other modules not distributed as standard, these allow us to use different sources of data to authenticate our users.

4.4. mod_autoindex

This allows us to control how Apache will generate file lists for directories without an index file, how to define formats, columns, orders, whether all files will be visible, etc.

4.5. mod_cgi

This is the main driver allowing Apache to serve CGI type files.

4.6. mod_dav y mod_dav_fs

This provides the functionalities of classes 1 and 2 of the WebDAV standard, a system that allows web content handling way beyond that specified by the HTTP standard. WebDAV converts the web server into a virtual disk server with the same features as a normal disk server: copying, reading, moving, deleting, etc.

Note

WebDAV is a HTTP extension for advanced content handling and authoring features.

4.7. mod_deflate

This is used to compress contents before sending them to the client, thus increasing the capacity of our communication lines. Note that not all browsers support compression, so it is a good idea to check the documentation of this module as you will obtain some very useful tips for detecting and avoiding problems with certain browsers. One configuration that avoids certain problems is:

```
<Location />
 # Enable filter
 SetOutputFilter DEFLATE
 # Problem: Netscape 4.x
 BrowserMatch &#710;Mozilla/4 gzip-only-text/html
 # Problem: Netscape 4.06-4.08
 BrowserMatch &#710;Mozilla/4\.0[678] no-gzip
# MSIE
 BrowserMatch \bMSIE !no-gzip !gzip-only-text/html
 # Do not compress images
 SetEnvIfNoCase Request_URI.(?:gif jpe?g png)$ no-gzip dont-vary.
 # Avoid modifications by proxies
 Header append Vary User-Agent env=!dont-vary
</Location>
```

4.8. mod_dir

Provides the necessary support for serving directories such as URLs by performing the right *redirects* and serving the index files.

4.9. mod_env

This is used to define new environment variables to pass to the CGI programs. It provides two directives: `SetEnv` and `UnsetEnv`.

```
SetEnv SPECIAL_VARIABLE value
UnsetEnv LD_LIBRARY_PATH
```

4.10. mod_expires

This allows us to generate HTTP headers indicating the expiry of content in line with a criterion defined by us:

```
# Enable the module
ExpiresActive On
# expire images GIF 1 months after modification
ExpiresByType image/gif "modification plus 1 month"
# HTML documents 1 week after modification
ExpiresByType text/html "modification plus 1 week"
# The rest 1 month after the last access
ExpiresDefault "access plus 1 month"
```

4.11. mod_ldap

Provides a *pool* of LDAP connections and makes a *cache* of the results for use in other modules requiring LDAP. It is essential when using LDAP as a source of authentication, etc., as it ensures that this does not cause a bottleneck.

4.12. mod_mime

This decides which type of MIME file (the standard that marks content types on the web) is associated with each file served by Apache. Based on the file extension, it can decide which `Content-type` to associate with it and it can even decide to take actions or send the file to special modules for processing. It includes a `mod_mime_magic` module complementing `mod_mime` for files where `mod_mime` has been unable to determine their type.

4.13. mod_speling

This offers a mechanism for correcting addresses (URL) that users may have entered incorrectly. If the requested resource is not found, Apache will try to correct the error. For example, upper-case may have been mistakenly used, etc.

If more than one possible page is found, a list will be displayed to the user from which to choose.

Note that this module has a substantial impact on performance.

4.14. mod_status

Displays information on server status, how busy it is and its activity level. The information provided is:

- The number of child processes.
- The number of unoccupied child processes.
- The status of each child, the number of requests and bytes served by each child.
- Number of total accesses and bytes served.
- When the server was booted and the time it has been running for.
- Average requests per second, bytes per second and bytes per request.
- Current CPU use per child and the total for Apache.
- Requests currently being processed.

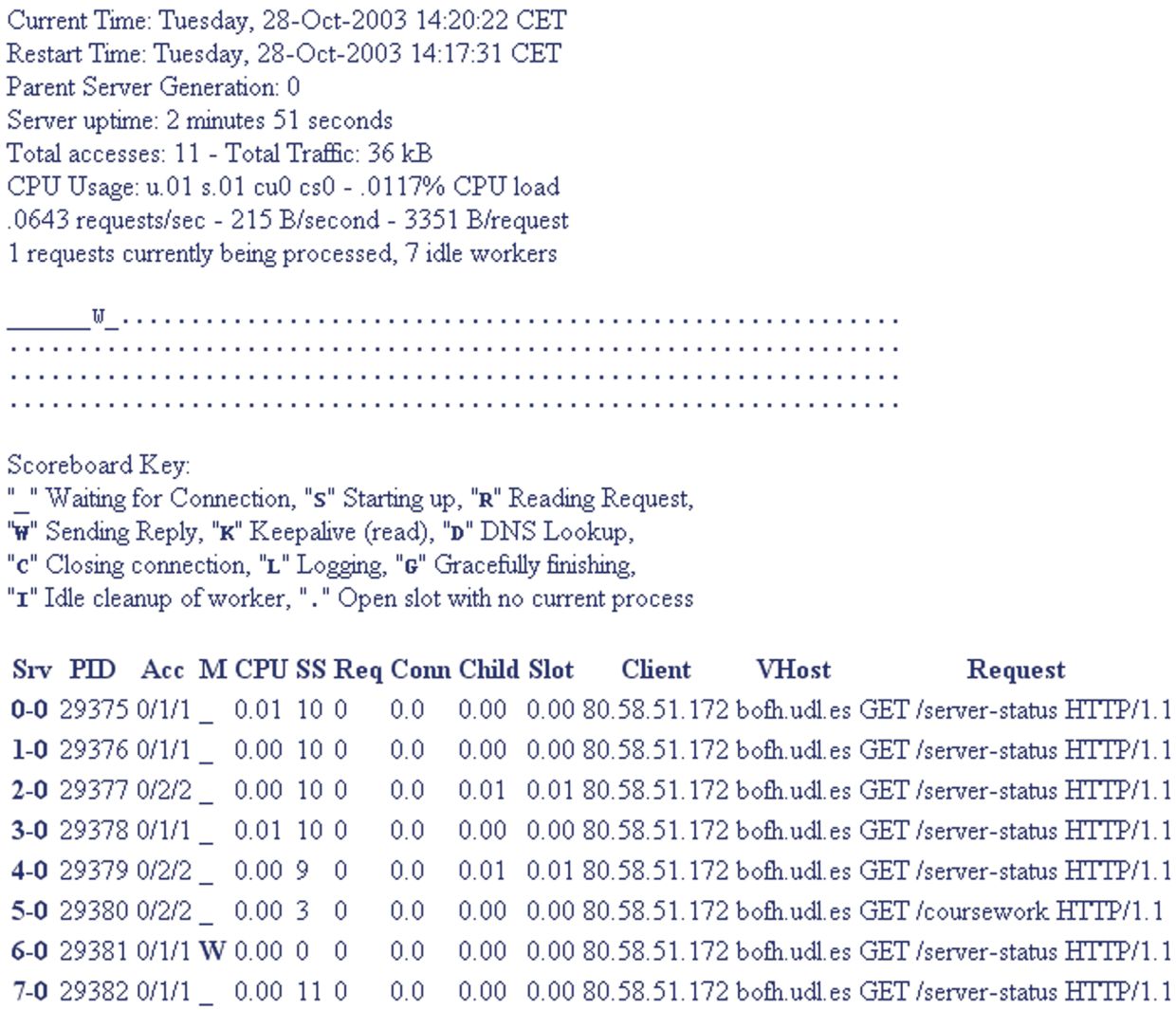
Some of this information may be disabled during compilation of the server.

The following example shows how to enable this module:

```
<Location /status>
     SetHandler server-status
     Order Deny,Allow
     Deny from all
     Allow from .uoc.edu
</Location>
```

We can see part of the result of the module in the figure below:

Figure 23.



4.15. mod_unique-id

This module provides an environment variable with a unique identifier for each request, guaranteed to be unique to a machine cluster. The module does not run under Windows. The environment variable provided is: `UNIQUE_ID`.

For this, it uses a value generated from (*ip_server, process pid, time stamp, counter16*). The `Counter16` is a 16-bit counter that rotates to 0 every second.

4.16. mod_userdir

This allows us to offer personal pages to the users of our system. This module has a basic functionality. It maps a subdirectory of the working directory of our system users to a specific URL, generally: http://www.uoc.edu/ ~user/.

The most common configuration,

```
UserDir public_html
```

would resolve requests to http://www.uoc.edu/ ~user/ from the contents of a subdirectory `public_html` in the working directory of `user`.

4.17. mod_usertrack

Provides a module that uses a *cookie* to monitor user activity over the web.

Monitoring and analysis

David Megías Jiménez (coordinator)
Jordi Mas (coordinator)
Carles Mateu

PID_00148401

Universitat Oberta de Catalunya
www.uoc.edu

Index

1. Analysis of HTTP server logs

Web servers (and those of FTP, *caching proxy*, etc.), if configured for the task, save files to the system where they record all events occurring during normal operation of the service. These files are called *log*. Here, we can find the record of failed operations, sometimes with the cause of the failure. We will also find the record of anomalous operations and a record of all operations performed correctly.

1.1. Format of the *log*

Web servers generally save logs in a format called *Common Log Format*. Servers that do not use this format by default generally include an option for using it. The *Common Log Format* is:

```
65.61.162.188 - - [14/Dec/2003:04:10:38 +0100] "GET /exec/rss HTTP/1.1" 200 9356
66.150.40.79 - - [14/Dec/2003:04:18:46 +0100] "HEAD / HTTP/1.1" 302 0
69.28.130.229 - - [14/Dec/2003:04:36:59 +0100] "GET /robots.txt HTTP/1.1" 404 1110
69.28.130.229 - - [14/Dec/2003:04:37:00 +0100] "GET /space/start HTTP/1.1" 200 17327
64.68.82.167 - - [14/Dec/2003:05:23:32 +0100] "GET /robots.txt HTTP/1.0" 404 1110
64.68.82.167 - - [14/Dec/2003:05:23:32 +0100] "GET / HTTP/1.0" 304 0
66.196.90.246 - - [14/Dec/2003:05:36:14 +0100] "GET /robots.txt HTTP/1.0" 404 1110
66.196.90.63 - - [14/Dec/2003:05:36:14 +0100] "GET /exec/authenticate HTTP/1.0" 302 0
66.196.90.63 - - [14/Dec/2003:05:36:19 +0100] "GET /space/start HTTP/1.0" 200 17298
69.28.130.222 - - [14/Dec/2003:05:50:32 +0100] "GET /robots.txt HTTP/1.1" 404 1110
69.28.130.222 - - [14/Dec/2003:05:50:33 +0100] "GET / HTTP/1.1" 302 14
69.28.130.222 - - [14/Dec/2003:05:50:34 +0100] "GET /space/start HTTP/1.1" 200 17327
```

As we can see, each line of the log file uses the following format:

Table 20.

Name	Description
remote client	IP address or name of the remote client that made the request
rfc931	Remote user identifier, if defined □ if it has not been defined
user	User identifier validated against our server □ if it has not been defined
date	Date of request
request	Request (method and URL) sent by the client
status	Numerical code of the result
bytes	Size of the result in bytes (0 if not applicable)

1.1.1. The Extended Common Log Format

There is an extended variant of the *Common Log Format* called *Extended Common Log Format*, more commonly known as the *Combined Log Format*, which adds two additional fields to the above format:

```
65.61.162.188 - - [14/Dec/2003:04:10:38 +0100] "GET /exec/rss HTTP/1.1"
     200 9356 "http://www.google.com" Mozilla/4.5[en]
66.150.40.79 - - [14/Dec/2003:04:18:46 +0100] "HEAD / HTTP/1.1"
     302 0 "http://www.altavista.com" Mozilla/3.1[en]
```

The fields added by this extension are:

Table 21.

Name	Description
referrer	The address from which the client comes. If it has not been defined, we will use –
User agent	The version of the browser software used by our client. If it cannot be determined, we will use –

1.2. Analysing the *errors*

The *log* files will provide us with some very useful information with important data on the visitors to our website. However, we will be unable to find lots of relevant data in our *log*, so we will need to estimate this based on the information in these files.

The data we can find in *log* are:

- Number of requests received (*hits*).
- Total volume in bytes of data and files served.
- Number of requests by file type (e.g. HTML).
- Different client addresses attended and requests for each.
- Number of requests per domain (from the IP address).
- Number of requests per directory or file.
- Number of requests per HTTP return code.
- Source addresses (*referrer*).
- Browsers and the versions used.

Although we can obtain a lot of information from the analysis of *log* there are some things that we cannot find out. Of these, the most important are:

- User identity, except where the user is identified by a server request.

- Number of users. Although we have the number of different IP addresses, we cannot know for certain the number of users, particularly if we take into account the existence of *caching proxy*. An IP address can represent:

  - A robot, spider or other automated browser program (such as those used by browsers such as Google).
  - An individual user with a browser on their computer.
  - A *caching proxy* server that can be used by hundreds of users.

- Qualitative data: user motivations, reactions to content, use of the data obtained, etc.

- Unseen files.

- What the user visited after leaving our server. This data will be recorded in the *log* of the server where the user went after leaving ours.

Other information is recorded but only partially, so we could interpret this data incorrectly. Many of these inconsistencies come from the *cache* made by browsers from that created by intermediate *caching proxy* servers, etc.

1.2.1. Common log interpretation *errors*

The information in *log* files does not allow us to obtain the following information, although most programs that analyse *log* generally do it:

- XML *hits* are not the same as visits. A page can generate more than one *hit*, because it contains images, style sheets, etc., that correspond to another *hit*.

- User sessions are easy to isolate and count. If there is no specific monitoring mechanism for sessions (*cookies*, etc.), they are normally obtained by considering all accesses from the same address over a consecutive period of time to be from the

- Same session. This does not take into account the existence of *caching proxy* servers or the possibility that a user may pause for a time (while consulting other sources of information, etc.).

- Data such as average pages per visit and lists of the most visited pages are obtained from user sessions. Given the difficulties in calculating these, the values obtained are not very reliable. Moreover, the existence of *caching proxy* servers has a very negative effect on lists of most visited pages. Precisely because they are the most visited, they are more likely to be stored on *cache*.

- It is difficult to gauge the geographical location of users from IP addresses. We will often locate an entire block of addresses in the city where the Internet services provider of a user has its head offices, while the user may be in a completely different place.

1.3. Log analysis programs

There are many free software programs available to analyse *log* that we can use to obtain information on the logs of visits to our website. Most of these generate their reports as web pages that can even be published on the site.

1.3.1. Webalizer

Webalizer is no doubt one of the most widespread. So much so that even some Linux distributions include it preconfigured.

If Webalizer is not installed in our server system, it is not too difficult to configure from the source code.

We first need to download the program from the website hosting it, where we can also obtain further documentation and some contributions (`http://www.mrunix.net/webalizer`). After downloading, we need to decompress it:

```
[carlesm@bofh k]$ tar xvzf webalizer-2.01-10-src.tgz
webalizer-2.01-10/
webalizer-2.01-10/aclocal.m4
webalizer-2.01-10/CHANGES
webalizer-2.01-10/webalizer_lang.h
webalizer-2.01-10/configure
[...]
webalizer-2.01-10/sample.conf
webalizer-2.01-10/webalizer.1
webalizer-2.01-10/webalizer.c
webalizer-2.01-10/webalizer.h
webalizer-2.01-10/webalizer.LSM
webalizer-2.01-10/webalizer.png
```

After decompressing it, we can configure its compilation in the building directory:

```
[carlesm@bofh webalizer-2.01-10]$ ./configure \
     --with-language=spanish --prefix=/home/carlesm/web
creating cache ./config.cache
checking for gcc... gcc
[...]
creating Makefile
linking ./lang/webalizer_lang.spanish to webalizer_lang.h
[carlesm@bofh webalizer-2.01-10]$
```

There is one important option, `with-language`, which is used to specify the language in which we want to build and install Webalizer. To choose the language, look in the `lang` subdirectory to see the available languages.

We now build the program as normal in these cases with the `make`.

```
[carlesm@bofh webalizer-2.01-10]$ make
gcc -Wall -O2 -DETCDIR=\"/etc\" -DHAVE_GETOPT_H=1
-DHAVE_MATH_H=1 -c webalizer.c
[...]
gcc -o webalizer webalizer.o hashtab.o linklist.o preserve.o
  parser.o output.o dns_resolv.o graphs.o -lgd -lpng -lz -lm
rm -f webazolver
ln -s webalizer webazolver
[carlesm@bofh webalizer-2.01-10]$
```

And, once built, we install it:

```
[carlesm@bofh webalizer-2.01-10]$ make install
/usr/bin/install -c webalizer /home/carlesm/web/bin/webalizer
[...]
rm -f /home/carlesm/web/bin/webazolver
ln -s /home/carlesm/web/bin/webalizer \
    /home/carlesm/web/bin/webazolver
[carlesm@bofh webalizer-2.01-10]$
```

To generate a *log* report, we can run it by passing a *log* file as a parameter and it will leave the files containing the report in the current directory.

```
[carlesm@bofh log]$ &#732;/web/bin/webalizer access_log
Webalizer V2.01-10 (Linux 2.4.20-8) Spanish
Using history access_log (clf)
Creating report in current directory
The machine name in the report is 'bofh'
History file not found...
Generating report on December 2003
Generating report summary
Saving file information...
45 records in 0.03 seconds
[carlesm@bofh log]$ ls
access_log
ctry_usage_200312.png
daily_usage_200312.png
hourly_usage_200312.png
index.html
usage_200312.html
usage.png
```

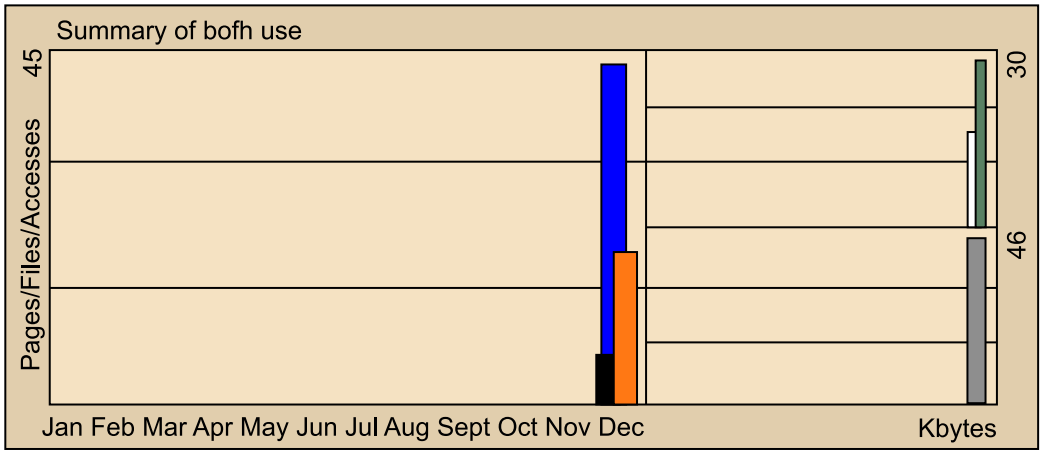
```
webalizer.hist
[carlesm@bofh log]$
```

This will give us a usage report based on the *log* of our web server:

Figure 24.

Statistics of use for bofh

Period summarised: last 12 months
Generated on 12-Dec-200 3:14 CET



Summary by months										
Month	Daily average				Monthly totals					
	Accesses	Files	Pages	Visits	Clients	Kbytes	Visits	Pages	Files	Accesses
Dec 2003	9	0	4	3	30	46	17	20	3	45
Totals						46	17	20	3	45

Figure 25.

(Daily statistics) (Statistics by hours) (URLs) (Input) (Output)
(Clients) (Source links) (Search) (Browsers) (Countries)

Monthly statistics for December 2003		
Total Accesses		45
Total Files		3
Total Pages		20
Total Visits		17
Total Kbytes		46
Total Clients		30
Total URLs		2
Total Source links		1
Total Browsers		6
	Average	Max.
Accesses per Hour	0	6
Accesses per Day	9	13
Files per Day	0	1
Pages per Day	4	5
Visits per Day	3	5
Kbytes per Day	9	14
Accesses by response code		
200 – OK		3
400 – Wrong request		1
403 – Forbidden		2
404 – Not found		30
405 – Method not allowed		8
413 – Request entity too large		1

1.3.2. Awstats

AWstats is a *log* statistics and analysis program with a comprehensive list of features and capabilities. Besides web server statistics, AWstats can generate statistics on mail servers, file servers, etc.

This *log* analyser can operate both as a CGI module and from the command line. The information it provides includes:

- Number of visits and visitors.
- Duration of visits.
- Authenticated users and visits.
- Time and date of the most traffic (pages, *hits*, bytes etc).
- Domains and countries of origin of the visits.
- Most visited pages and input/output pages.
- Types of files requested.
- Visitor bots (from search engines, etc.).
- Search engines for visitor origin, including the words and phrases used for the search.
- HTTP return errors and codes.
- Browser features (Java, Flash, etc.) and screen size.

- Compression statistics (if used).
- Browsers used to visit us (browser versions, pages served for each browser, bytes per browser, etc.).
- Operating systems used to visit us.

Note that all of these reports and statistics are obtained from the data in the *log* file, for which we have already discussed the problems associated with the use of *proxy caches*, etc.

If our system does not come with the program pre-installed, we can install it so long as we have at least a Perl interpreter. We will begin installation by downloading the program code from its web server (`http://awstat.sf.net`).

- After downloading, decompress as usual with the `tar`.

```
[carlesm@bofh aws]$ tar xvzf awstats-5.9.tgz
awstats-5.9/
awstats-5.9/docs/
awstats-5.9/docs/awstats.htm
[....]
awstats-5.9/wwwroot/icon/other/vv.png
awstats-5.9/wwwroot/js/
awstats-5.9/wwwroot/js/awstats_misc_tracker.js
[carlesm@bofh aws]$
```

- The first step of installation is to configure the web server to use the *log* format called *NCSA combined/XLF/EL* F. In the Apache configuration file, for example, *httpd.conf* we need to change:

```
CustomLog /var/log/httpd/access_log common
```

to

```
CustomLog /var/log/httpd/access_log combined
```

- The next step is to copy the contents of the subdirectory `wwwroot/cgi-bin/`, including the subdirectories it contains, to the directory where our web servers will find the CGI files.

- We must now copy the contents of `wwwroot/icon` to a subdirectory of the directory where we store the contents of our web server.

- Copy the file `awstats.model.conf` to another file that we will call `awstats.servername.conf` and locate it in one of the following

directories: `/etc/awstats`, `/etc/opt/awstats` etc. or in the directory where we find `awstats.pl`.

- We will need to configure this file by editing at least the following variables:
  - **LogFile:** path to the *log*.
  - **LogType:** type of *log*: **W** for web, **M** for mail, **F** for FTP, and **O** for other cases.
  - **LogFormat:** check that this is 1.
  - **DirIcons:** path to the directory where the icons are located.
  - **SiteDomain:** name of the web server.

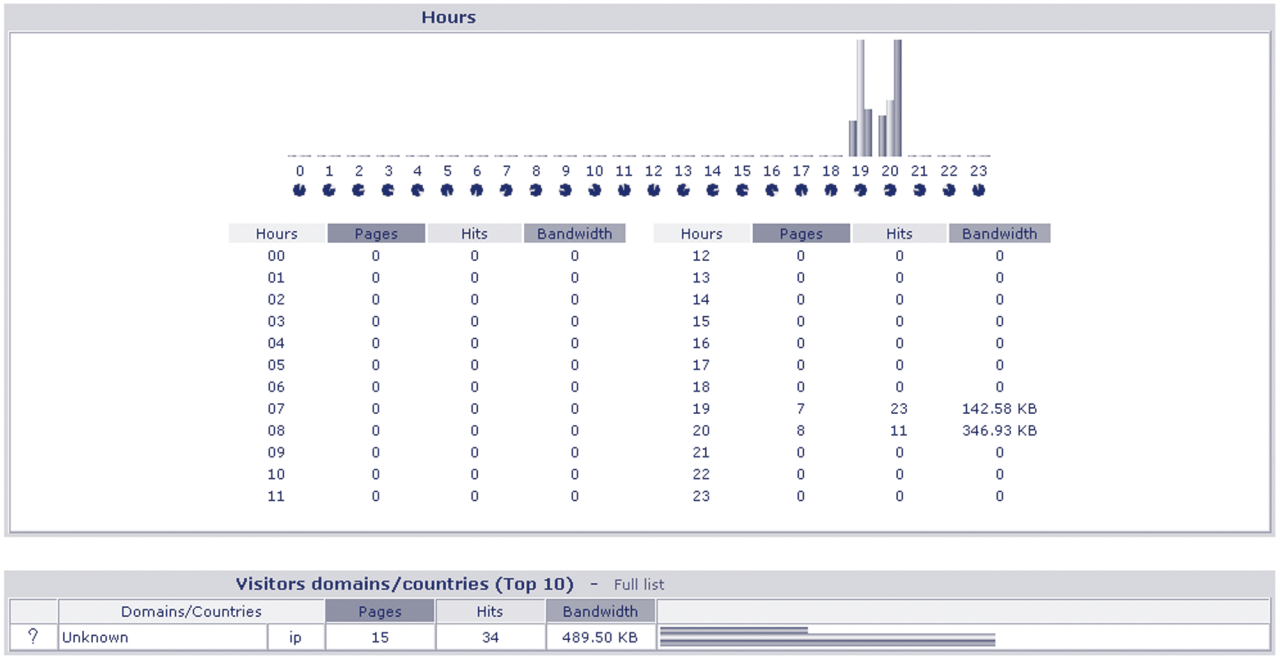
- Once the program is configured, we run AWStats from the command line, from the `cgi-bin` directo y, with:
  `awstats.pl -config=servername -update`

- We can now allow visits to our statistics from the following URL:
  `http://servername/cgi-bin/awstats.pl`
  This will display the statistics generated dynamically. Another option is to generate statistics as static HTML pages and to access these. To do this, we need to execute:
  `awstats.pl-config=servername-output-staticlinks>`
  `awstats.server.html`
  We should move the file generated (`awstats.server.html`) to a directory that can be accessed by the web server and we can now access it from a browser.

- Although we could use dynamic updating from the browser, we need to programme the server to regularly update the statistics with the command used previously to create them. To do so, we can use the `cron` facilities of our Linux systems, complementing them with `logrotate`, if we have it, so that when the time comes to change `log` files (to save disk space) these will be incorporated into the statistics.

We can now see part of what the generated statistics look like:

Figure 26.



Hours

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Hours	Pages	Hits	Bandwidth	Hours	Pages	Hits	Bandwidth
00	0	0	0	12	0	0	0
01	0	0	0	13	0	0	0
02	0	0	0	14	0	0	0
03	0	0	0	15	0	0	0
04	0	0	0	16	0	0	0
05	0	0	0	17	0	0	0
06	0	0	0	18	0	0	0
07	0	0	0	19	7	23	142.58 KB
08	0	0	0	20	8	11	346.93 KB
09	0	0	0	21	0	0	0
10	0	0	0	22	0	0	0
11	0	0	0	23	0	0	0

Visitors domains/countries (Top 10) - Full list

	Domains/Countries		Pages	Hits	Bandwidth
?	Unknown	ip	15	34	489.50 KB

Figure 27.

Pages-URL (Top 10) - Full list - Entry - Exit

23 different pages-url	Viewed	Average size	Entry	Exit
/cgi-bin/awstats.pl	10	66.24 KB	1	1
/manual/	3	4.57 KB		
/	2	728 Bytes	1	
/manual/mod/	2	12.73 KB		
/index.html.en	2	728 Bytes	1	
/manual/de/new_features_2_0.html	1	14.67 KB		1
/manual/en/	1	6.85 KB		
/manual/sections.html	1	24.81 KB		
/icons/	1	18.65 KB		
/manual/en/new_features_2_0.html	1	13.16 KB		
Others	13	22.25 KB	1	

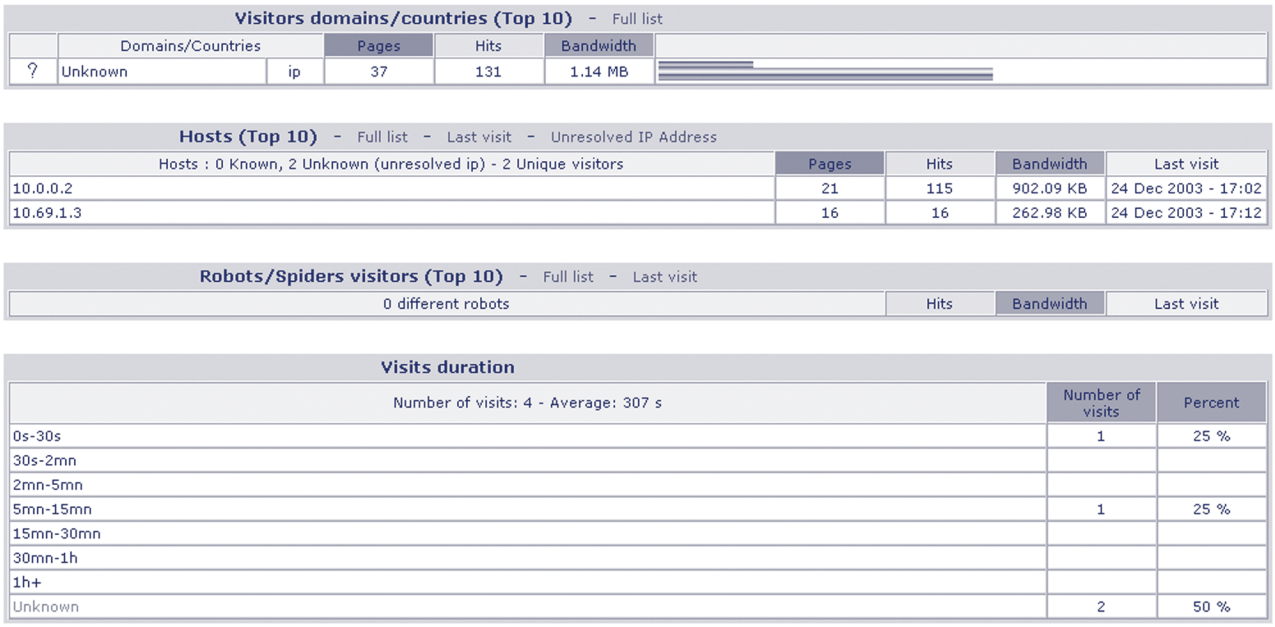
Operating Systems (Top 10) - Full list/Versions - Unknown

	Operating Systems	Hits	Percent
	Windows	115	87.7 %
?	Unknown	16	12.2 %

Browsers (Top 10) - Full list/Versions - Unknown

	Browsers	Grabber	Hits	Percent
	Firebird	No	70	53.4 %
	Mozilla	No	45	34.3 %
	Lynx	No	16	12.2 %

Figure 28.



Visitors domains/countries (Top 10) - Full list

	Domains/Countries		Pages	Hits	Bandwidth
?	Unknown	ip	37	131	1.14 MB

Hosts (Top 10) - Full list - Last visit - Unresolved IP Address

Hosts : 0 Known, 2 Unknown (unresolved ip) - 2 Unique visitors	Pages	Hits	Bandwidth	Last visit
10.0.0.2	21	115	902.09 KB	24 Dec 2003 - 17:02
10.69.1.3	16	16	262.98 KB	24 Dec 2003 - 17:12

Robots/Spiders visitors (Top 10) - Full list - Last visit

0 different robots	Hits	Bandwidth	Last visit

Visits duration

Number of visits: 4 - Average: 307 s	Number of visits	Percent
0s-30s	1	25 %
30s-2mn		
2mn-5mn		
5mn-15mn	1	25 %
15mn-30mn		
30mn-1h		
1h+		
Unknown	2	50 %

1.3.3. Analog

Analog is perhaps the oldest and most widely-used free software *log* analysis program. It is usually used in combination with another program, ReportMagic, which complements the report display features of Analog.

To install it, go to the program website at: http://www.analog.cx. Here, you will find precompiled versions for most platforms and versions with source code. Download the source code version and install it.

- Decompress the program code:

```
[carlesm@bofh l]$ tar vxzf analog-5.91beta1.tar.gz
analog-5.91beta1/
analog-5.91beta1/docs/
analog-5.91beta1/docs/LicBSD.txt
[...]
analog-5.91beta1/anlgform.pl
analog-5.91beta1/logfile.log
[carlesm@bofh l]$
```

- Now enter the `src` directory and edit the file called `anlghead.h`. This file defines certain configuration variables: server name, etc. We change the values we require.

- We compile it with `make`:

```
[carlesm@bofh src]$ make
gcc -O2 -DUNIX -c alias.c
```

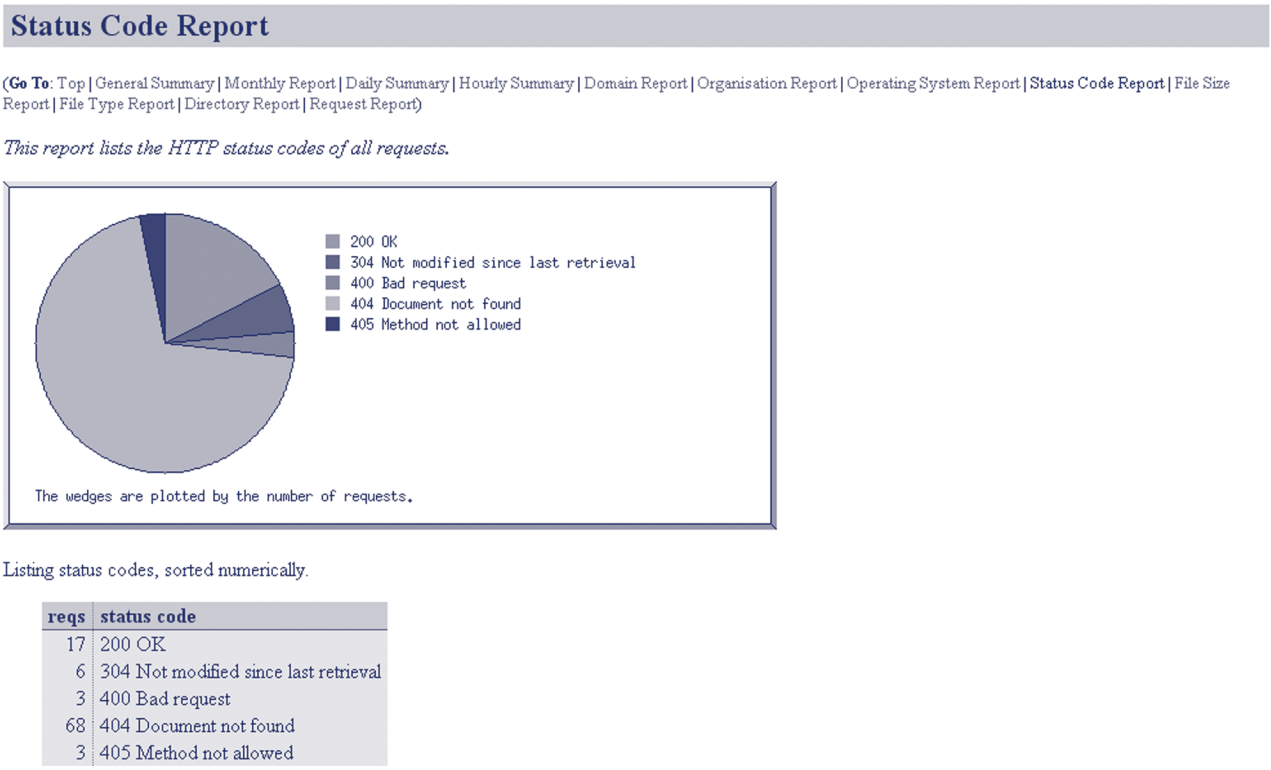
```
gcc -O2 -DUNIX -c analog.c
[...]
bzip2/huffman.o bzip2/randtable.o -lm
***
***IMPORTANT: You must read the licence before using analog
***
[carlesm@bofh src]$
```

- We can edit the `analog.cfg` file to define the output format of Analog and some of its operating parameters.

- When you are finished editing, execute Analog with `analog` to generate the statistics file.

The statistics generated will look like this:

Figure 29.



Status Code Report

(**Go To**: Top | General Summary | Monthly Report | Daily Summary | Hourly Summary | Domain Report | Organisation Report | Operating System Report | **Status Code Report** | File Size Report | File Type Report | Directory Report | Request Report)

This report lists the HTTP status codes of all requests.

Listing status codes, sorted numerically.

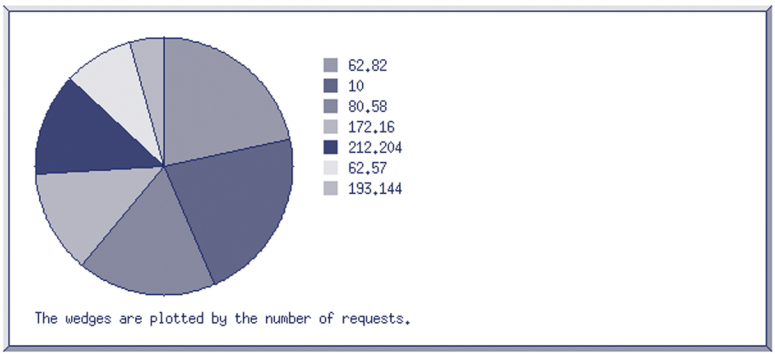
reqs	status code
17	200 OK
6	304 Not modified since last retrieval
3	400 Bad request
68	404 Document not found
3	405 Method not allowed

Figure 30.

Organisation Report

(**Go To**: Top | General Summary | Monthly Report | Daily Summary | Hourly Summary | Domain Report | **Organisation Report** | Operating System Report | Status Code Report | File Size Report | File Type Report | Directory Report | Request Report)

This report lists the organisations of the computers which requested files.



Listing organisations, sorted by the number of requests.

reqs	%bytes	organisation
5	17.65%	62.82
5	29.41%	10
4		80.58
3	17.65%	172.16
3	17.65%	212.204
2	11.76%	62.57
1	5.88%	193.144

Figure 31.

General Summary

(**Go To**: Top | **General Summary** | Monthly Report | Daily Summary | Hourly Summary | Domain Report | Organisation Report | Operating System Report | Status Code Report | File Size Report | File Type Report | Directory Report | Request Report)

This report contains overall statistics.

Successful requests: 23
Average successful requests per day: 3
Successful requests for pages: 23
Average successful requests for pages per day: 3
Failed requests: 74
Distinct files requested: 1
Distinct hosts served: 16
Data transferred: 4.67 kilobytes
Average data transferred per day: 816 bytes

Monthly Report

(**Go To**: Top | General Summary | **Monthly Report** | Daily Summary | Hourly Summary | Domain Report | Organisation Report | Operating System Report | Status Code Report | File Size Report | File Type Report | Directory Report | Request Report)

This report lists the activity in each month.

Each unit (+) represents 1 request for a page.

month	reqs	pages	
Dec 2003	23	23	++++++++++++++++++++++

Busiest month: Dec 2003 (23 requests for pages).

2. Statistics and counter tools

2.1. Counters

Web counters are visual indications to visitors to our page of the number of visits we have had. They are the visual indicator with the most aesthetic, as opposed to useful, value, since many of these counters have no value other than statistical, given that they only count *hits* (requests from the page to the server, which do not always correspond to real visits).

There are a number of ways to add a counter to your site:

- Use a CGI or servlet to generate the counter (using images or references to images).
- Use a counter service displaying the counter or references to the images.
- Use, where available, a server extension for the counter.

2.1.1. Using a CGI to generate the counter

To add a visitor counter to our site, we can use an external program such as a CGI or servlet, which will count visits and generate the counter. We can use one of the many counters available: `Count`.

To start with, we need to download `Count` from the website:

`http://www.muquit.com/muquit/software/Count/Count.html`

Once downloaded, we can begin the installation.

- The first step is to decompress the program code. To do so, we need to use `tar`:

```
[carlesm@bofh n]$ tar xvzf wwwcount2.6.tar.gz
./wwwcount2.6/
./wwwcount2.6/DIR/
[...]
./wwwcount2.6/utils/rgbtxt2db/rgb.txt
./wwwcount2.6/utils/rgbtxt2db/rgbtxt2db.c
[carlesm@bofh n]$
```

- We can now compile the program:

```
[carlesm@bofh wwwcount2.6]$ ./build \
```

```
    --with-cgi-bin-dir=/home/carlesm/apache/cgi-bin/ \
    --prefix=/home/carlesm/apache/counter
```

The `prefix` parameter indicates where we want Counter to save its files.

- After compiling, we can install with:

```
[carlesm@bofh wwwcount2.6]$ ./build --install
```

The program will allow us to confirm the installation directories before copying the files.

- We need to configure Counter. To do this, we will edit the `count.cfg` file located in the `conf` directory of Counter's installation; in our case: `/home/carlesm/apache/counter`.

- We can add the following HTML fragment, which references our counter CGI to use Counter:
  `<img src="/cgi-bin/Count.cgi" alt="Counter">`
  The counter will look like this:

Figure 32.

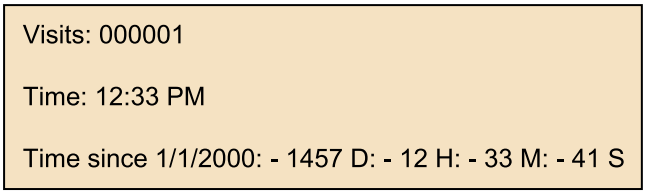
- We can use the many parameters of Counter to modify what and how it is displayed:

```
Visits:

<img src="/cgi-bin/Count.cgi?dd=C&ft=6&frgb=ff0000" alt="******">

<br>

Time:

<img src="/cgi-bin/Count.cgi?display=clock&pad=F" alt="Count">

<br>

Time since 1/1/2000:

<img
src="/cgi-bin/Count.cgi?cdt=2000;1;1;0;0;0&dd=cd&ft=2&frgb=000000"

alt="Count">
```

```
<br>
```

This graphic shows how the three counters look:

Figure 33.



2.1.2. Counter services

There are numerous commercial counter services, though many also have free options that allow us to add a counter to our website without the need to install additional programs on our server. Many of these counters also offer statistical analysis of visits.

These services include:

Table 22. Counter services

Name	Address
123 Counter	http://www.123counter.com/
Admo Free Counters	http://www.admo.net/counter/
BeSeen: Hit Counter	http://www.beseen.com/hitcounter/
BoingDragon: AnimatedCounters	http://www.boingdragon.com/types.html
Dark Counter	http://www.lunamorena.net/counter/
Digits.com	http://www.digits.com/create.html
Easy Counter	http://www.easycounter.com/
i-Depth	http://www.i-depth.com/X/guru3#hcnt
LBInet Counters	http://www.lbi.net/c50000/
LunaFly: Free Counter	http://www.freecount.co.uk/
MyComputer Counter	http://counter.mycomputer.com/
Spirit Counters	http://www.thesitefights.com/userv/

We will now create a counter. To do so, we will use the service offered by Digits.com. First, we need to visit their site and fill in the form requesting the service.

After filling in the form, Digits.com will provide us with a fragment of HTML code to include on our page. The code will look something like this:

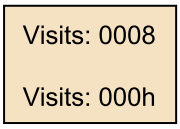
```
<IMG SRC="http://counter.digits.com/wc/-d/4/carlesm"
   ALIGN=middle
WIDTH=60 HEIGHT=20 BORDER=0 HSPACE=4 VSPACE=2>
```

The counter will look like this:

Figure 34.

Using the parameters passed to the URL, we can change the appearance of the counter to obtain this:

Figure 35.



with the following HTML code:

```
<p>
Hits:
<IMG
     SRC="http://counter.digits.com/wc/-d/4/-z/-c/8/carlesm"
     ALIGN=middle
     WIDTH=60 HEIGHT=20 BORDER=0 HSPACE=4 VSPACE=2>
</p>
<p>
Hits:
<IMG
     SRC="http://counter.digits.com/wc/-d/4/-z/-c/26/carlesm"
     ALIGN=middle
     WIDTH=60 HEIGHT=20 BORDER=0 HSPACE=4 VSPACE=2>
</p>
```

2.1.3. Server extension (Roxen)

The free software web server Roxen has a HTML extension that can be used to implement a counter easily without the need to install additional software on our system.

To do this, we have two new HTML tags: `accessed` and `gtext`, used to indicate the number of *hits* obtained by a page and to display text as graphics, respectively.

Note

The `<counter>`. To maintain compatibility with previous versions, in which there was a specific tag for counters, `<counter>`, Roxen still provides this tag, now implemented as a combination of `accessed` and `gtext`.

An example of the use of this extension is the following code:

```
Visitas:
  <gtext2bshadow=1bevel=2><accessed/></gtext><br/>
Hits:
<b><accessed /></b>
```

which produces the following result:

Figure 36.

2.2. Visitor statistics

Another option for monitoring the number of visitors to our website, where they come from and other similar data, without using a *log* analysis program, is to use one of the statistics and visitor counting services available, some free of charge, on the Internet.

The following list details some of these services:

Table 23. Visitor counting services

Name	Address
Counted	http://www.counted.com
Cyber Stats	http://www.pagetools.com/cyberstats/
Gold Stats	http://www.goldstats.com
Hit Box	http://www.websidestory.com
IPSTAT II	http://www.ipstat.com
NedStat	http://www.nedstat.com
RealTracker	http://www.showstat.com
Site-Stats	http://www.site-stats.com
Site Tracker	http://www.sitetracker.com
Stats 3D	http://www.stats3d.com
Stat Trax	http://www.stattrax.com
The-Counter.net	http://www.the-counter.net

Name	Address
WebStat.com	http://www.webstat.com
WebTrends Live	http://www.webtrendslive.com/default.htm
WhozOnTop	http://world.icdirect.com/icdirect/hitTracker.asp

Most of these services work in much the same way. When we sign up (whether for free of paid services), we will be given a HTML code to include on our pages. This code generally references an image from the website of the statistics service. Some of these services offer an image used as a counter.

An example of this code, in this case, for NedStat, is:

```
<!-- Begin Nedstat Basic code -->
<!-- Title: Carlesm Homepage -->
<!-- URL: http://carlesm/ -->
<script language="JavaScript"
     src="http://m1.nedstatbasic.net/basic.js">
</script>
<script language="JavaScript">
<!--
nedstatbasic("AA7hmw77L/vVx928ONUhsGLjd6mQ", 0);
// -->
</script>
<noscript>
<a target=_blank
     href="http://v1.nedstatbasic.net/stats?AA7hmw77L/vVx928ONUhsGLjd6mQ">
<img
     src="http://m1.nedstatbasic.net/n?id=AA7hmw77L/vVx928ONUhsGLjd6mQ"
     border=0 nosave width=18 height=18>
</a>
</noscript>
<!-- End Nedstat Basic code -->
```

Once we have included this code in our page, the statistics service will monitor the times our page is visited. We can then display the statistics for our page, as in these examples:

Figure 37.

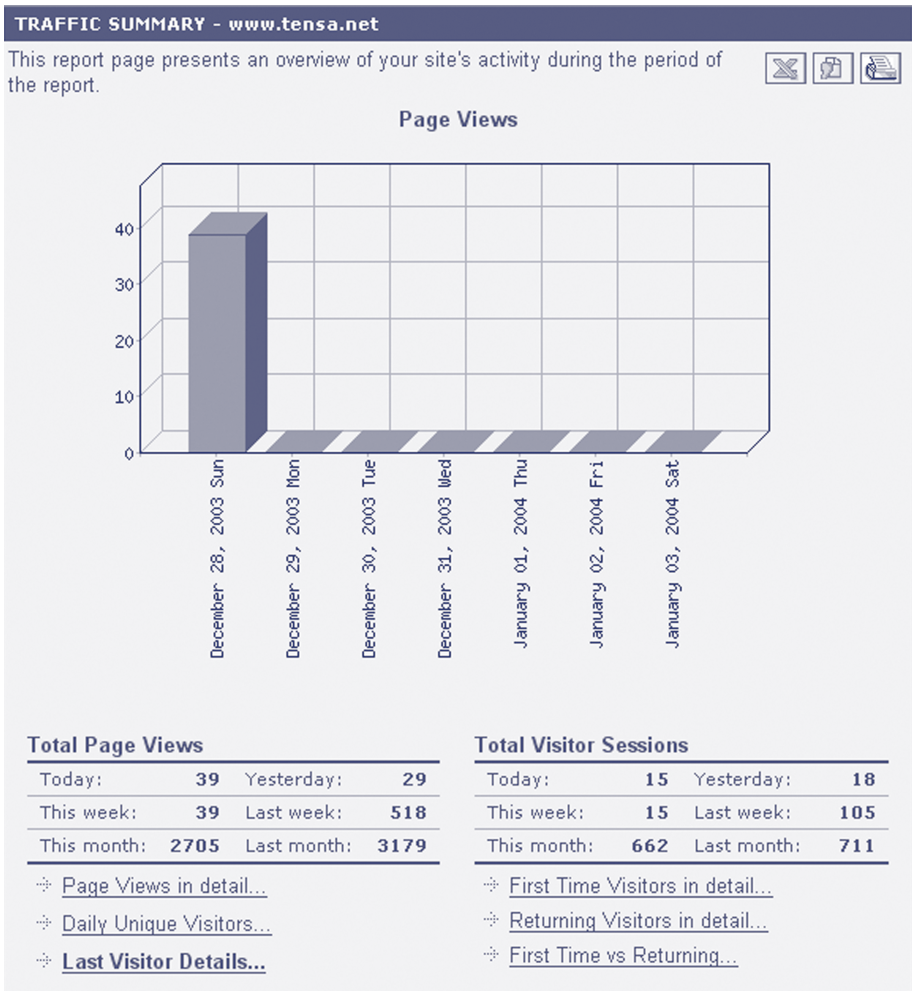


Figure 38.

Figure 39.

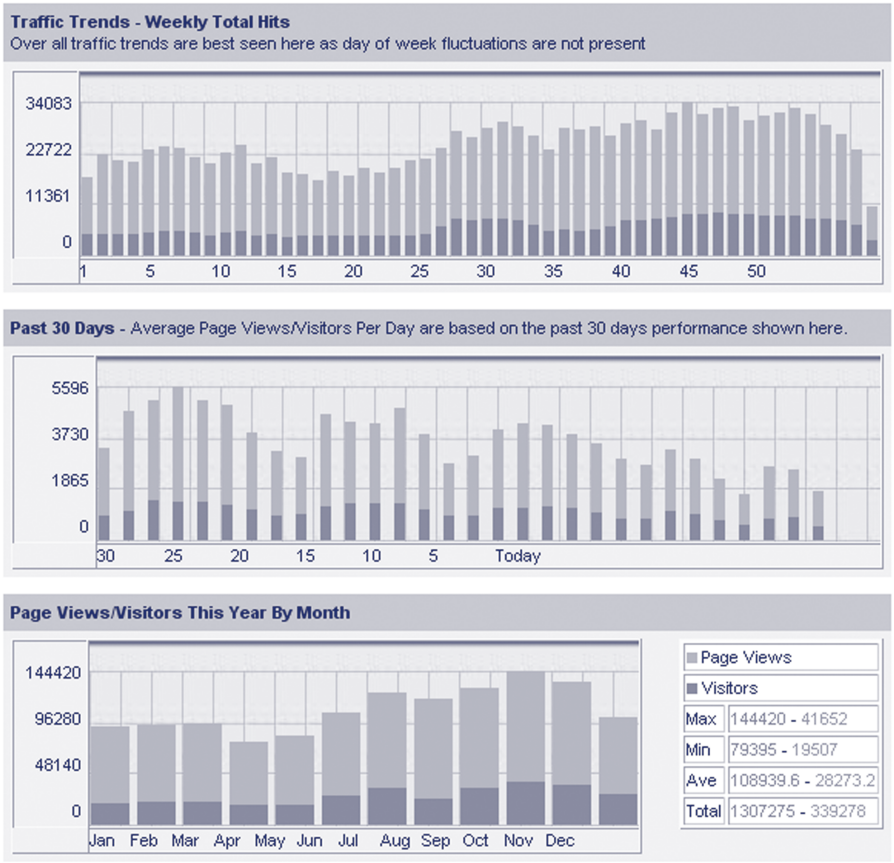
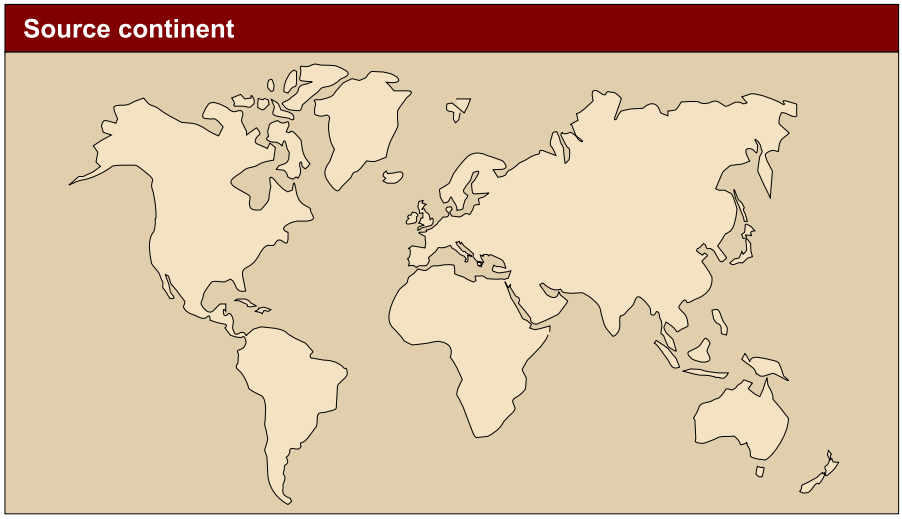

Traffic Trends - Weekly Total Hits
Over all traffic trends are best seen here as day of week fluctuations are not present
34083
22722
11361
0
1 5 10 15 20 25 30 35 40 45 50
Past 30 Days - Average Page Views/Visitors Per Day are based on the past 30 days performance shown here.
5596
3730
1865
0
30 25 20 15 10 5 Today
Page Views/Visitors This Year By Month
144420
96280
48140
0
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Page Views
Visitors
Max 144420 - 41652
Min 79395 - 19507
Ave 108939.6 - 28273.2
Total 1307275 - 339278

Figure 40.



Source continent		
1. Europe	4747	76.1%
2. South America	388	6.2%
3. USA and Canada	384	6.2%
4. Central America	267	4.3%
5. Asia	108	1.7%
6. .net and .org	19	0.3%
7. Australia	18	0.3%
8. Africa	16	0.2%

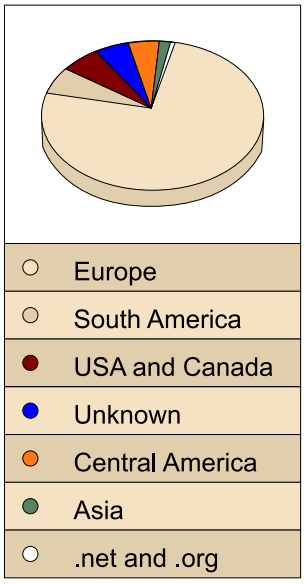
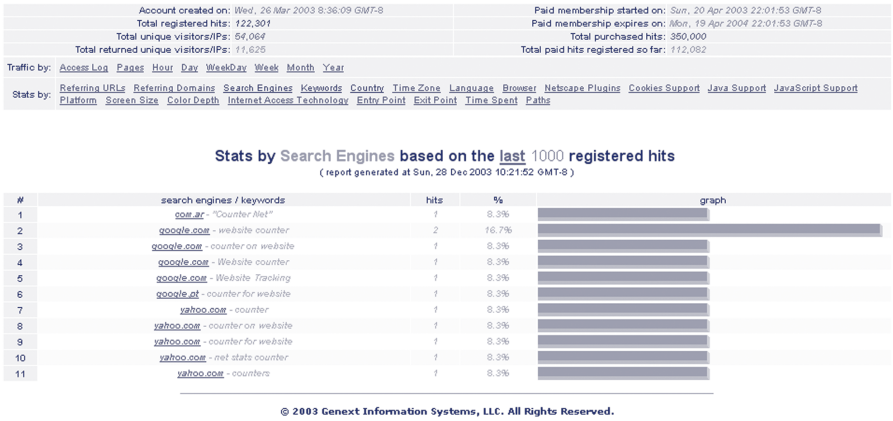


Figure 41.



Account created on: *Wed, 26 Mar 2003 8:36:09 GMT-8*	Paid membership started on: *Sun, 20 Apr 2003 22:01:53 GMT-8*
Total registered hits: *122,301*	Paid membership expires on: *Mon, 19 Apr 2004 22:01:53 GMT-8*
Total unique visitors/IPs: *54,064*	Total purchased hits: *350,000*
Total returned unique visitors/IPs: *11,625*	Total paid hits registered so far: *112,082*

Traffic by: Access Log Pages Hour Day WeekDay Week Month Year

Stats by: Referring URLs Referring Domains Search Engines Keywords Country Time Zone Language Browser Netscape Plugins Cookies Support Java Support JavaScript Support Platform Screen Size Color Depth Internet Access Technology Entry Point Exit Point Time Spent Paths

Stats by Search Engines based on the last 1000 registered hits

(report generated at Sun, 28 Dec 2003 10:21:52 GMT-8)

#	search engines / keywords	hits	%	graph
1	cow.ar - "Counter Net"	1	8.3%	
2	google.com - website counter	2	16.7%	
3	google.com - counter on website	1	8.3%	
4	google.com - Website counter	1	8.3%	
5	google.com - Website Tracking	1	8.3%	
6	google.pt - counter for website	1	8.3%	
7	yahoo.com - counter	1	8.3%	
8	yahoo.com - counter on website	1	8.3%	
9	yahoo.com - counter for website	1	8.3%	
10	yahoo.com - net stats counter	1	8.3%	
11	yahoo.com - counters	1	8.3%	

3. Performance analysis

One of the keys to the success of a website is the level of comfort of our users, that they have a pleasant visiting experience on our site, that they receive a fluid response to their actions, without delayed responses, etc. Another of these key points is the performance we obtain from our systems. The greater the performance, the more we get out of our investment. Moreover, this often translates into a smoother and more pleasant response for our users, with reduced access times, etc.

3.1. Obtaining performance information on Apache

The first point offering information on how the web server is running is the web server itself.

3.1.1. mod_status

As we have seen, Apache has a module called `mod_status` that di plays a page of information on the performance of the web server at a given moment. This page looked like this:

Figure 42.

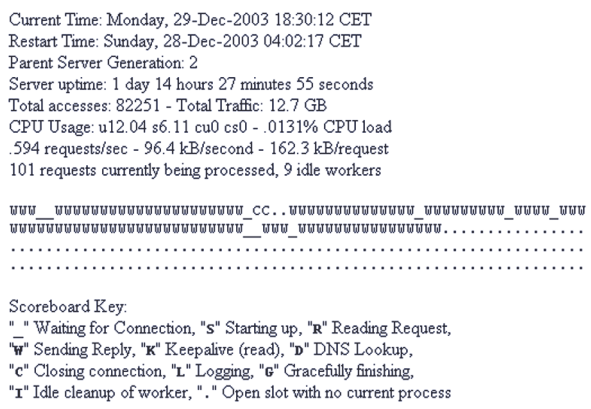
Current Time: Tuesday, 28-Oct-2003 14:20:22 CET
Restart Time: Tuesday, 28-Oct-2003 14:17:31 CET
Parent Server Generation: 0
Server uptime: 2 minutes 51 seconds
Total accesses: 11 - Total Traffic: 36 kB
CPU Usage: u.01 s.01 cu0 cs0 - .0117% CPU load
.0643 requests/sec - 215 B/second - 3351 B/request
1 requests currently being processed, 7 idle workers

_____W_.....
.....
.....
.....

Scoreboard Key:
"_" Waiting for Connection, "S" Starting up, "R" Reading Request,
"W" Sending Reply, "K" Keepalive (read), "D" DNS Lookup,
"C" Closing connection, "L" Logging, "G" Gracefully finishing,
"I" Idle cleanup of worker, "." Open slot with no current process

If we overload the server (we can use specialist programs or Apache's own `ab` for this), we will see how the result of `mod_status` gets complicated:

Figure 43.



In the information provided by `mod_status`, we can see what Apache calls the *scoreboard* which represents all of the *slots* or request processors and their status. From the information on the screen we can deduce that many of these *slots* are busy sending information (those marked with W). In theory, this indicates that our server is responding well to the requests it receives. Note that this test was carried out by simultaneously sending a total of one hundred requests to the server, until 100,000 requests were reached, and the client machine of these requests was the server itself. We can also see in the information displayed that the server will has some free *slots* in it. This tells us that we can still receive more requests (it does not tell us that we can attend to them at the desired speed).

3.2. Obtaining information on system performance

Another source of information on server performance is the operating system. Operating systems generally come with a broad range of tools that we can use to find out their status at any time. These tools tell us the level of CPU usage, memory, etc.

For our analysis, we will use the tools usually available in Unix systems. If your system does not have these tools, many can be found free on the Internet. Similar tools are available for most operating systems.

3.2.1. CPU load

This term is used to indicate how busy the CPU or central processing unit of our server system is. Web servers like Apache make considerable use of the processor for their normal functions, and this increases when we use dynamically generated pages, etc.

If, as is sometimes the case, the web server system provides services in addition to web services (mail server, for example), we need to be very careful about the level of processor used.

We can obtain a rough estimate of the use of our system with the command: `vmstat`.

```
[carlesm@bofh carlesm]$ vmstat 2
```

procs						memory		swaps		io		system			cpu
r	b	w	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id
1	0	0	26364	46412	178360	206072	0	0	4	13	29	41	7	0	46
0	0	0	26364	46412	178360	206072	0	0	0	0	107	22	0	0	100
0	0	0	26364	46412	178360	206072	0	0	0	0	108	22	0	0	100

In this example, we can observe the distribution of the CPU load with the three values to the right, labelled `us, sy and id`, which correspond to: *user, system* and *idle* respectively. These values indicate the percentage of time that the processor remains in each of these states:

- *user*: the processor remains in user space while executing programs.

- *system*: the processor is in this state while it executes code forming part of the operating system kernel or while attending calls to the system, such as those from communications drivers, etc.

- *idle*: is the time the processor is free, not busy.

A constantly high `us` value indicates intensive use of the processor. In this case, the machine is reaching its response limit and we need to find a solution to ensure that increased load does not result in a loss of responsiveness. These solutions must be geared towards reducing processor consumption (rewriting code, optimising code) or increasing processing capacity.

A high `sy` value indicates that the system is busy for long periods with system kernel tasks. We should try to find out the causes of this (wrong or faulty drivers, inadequate *hardware*, etc.) and solve them.

A high `id` value (if we have performance problems) would indicate that the problem does not lie with the processor and that we need to look at other aspects.

Other important values revealed by `vmstat` are the two columns:

- `in` (*interrupts*): the number of interruptions occurring per second (including those for the system clock).

- `Cs` (*context switches*): the number of context switches (for processes or active threads in the processor) occurring per second.

Too high a `cs` value usually indicates that there are too many system processes running. If this excess is caused by the processes generated by the web server, we need to lower this figure. Another possible cause might be a high and poorly optimised level of inter-process communications (IPC), which could lead to excessive context switches.

The first three columns indicate other values that we should also look at:

- `r`: number of processes ready to execute.

- `b`: number of blocked processes.

- `w`: number of processes passed to *swaps* memory but which are executable.

In load situations, these indicators on the number of processes can give us an idea as to the contention level to enter in the processor that we need to execute:

23	0	1	27328	7944	169696	210932	0	0	0	480	116	13715	41	59	0
21	0	0	27336	7576	169624	211376	0	4	0	4	104	13724	43	57	0
17	0	1	27336	7096	169448	211924	0	0	0	474	113	13726	40	60	0
13	0	0	27344	6624	169444	212296	0	4	0	4	105	13753	38	62	0

In this example, obtained at a point of heavy server load, we have a high number of processes available for execution and a processor availability of 0. Since these data were taken from a uniprocessor machine, and given the number of context switches made, we can conclude that it is executing too many processes.

There are other tools, including `top, ps`, that provide the same or complementary information. It is very important to know the tools we have available in our operating system and their capacity.

3.2.2. Memory use

The same command, `vmstat` also pr vides basic data on memory use. The following columns contain information on memory use:

The columns `swpd, free, buff` and `cache` indicate, respectively:

- `swpd`: memory use *swaps* (swap memory).

- `free`: the amount of free physical memory (RAM).
- `buff`: the amount of memory used as *buffers*.
- `cache`: the amount of memory used as *cache*.

We need to keep a close eye on these values when the web server is under heavy loads. A very high `swpd` and a very low `free, buff` and `cache` would suggest that our system does not have enough memory and has to resort to using the virtual disk memory, which is much slower than RAM.

The two columns `si` and `so` tell us the amount of memory sent to the virtual disk memory or the memory recovered from there in kB/s. Values other than zero sustained over time would suggest that the system lacks memory and thus has to continually discard and recover data from the disk.

3.2.3. Disk access

One of the points often overlooked when sizing up equipment for web servers is disk access. We need to take into account that a web server constantly sends data that it reads from the disk to remote clients (pages, images etc). Thus, short disk access times and high transfer speeds could give the web server high page-serving performance.

For an idea of how our disks are responding to requests, we can use the `vmstat` command as well as a more specialised command: `iostat`. The result of executing `iostat` is as follows:

avg-cpu:	%user	%nice	%sys	%idle		
	67.50	0.00	18.50	14.00		
Device:		tps	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn
dev3-0		32.00	208.00	844.00	416	1688

In this result, we can see the number of disk access transactions that have occurred, the number of blocks (sectors) read and written, and the total blocks during the time measured.

Note

disk use can be motivated either by access to data from programs or by the use of *swaps*. In the second case, the best...

Very high values would suggest high disk usage. We must therefore make sure that the system has fast disks with the lowest possible access time, the highest transfer speed available and sufficient memory to perform disk *cache* efficiently or to avoid excessive use of *swaps*.

3.3. Configuration improvements

We can make some improvements by adjusting the configuration of the web server. Different versions of Apache incorporate these adjustments by default. However, we will need to be clear on the values being used by the web server, since changing these could have drastic effects on system performance.

3.3.1. DNS queries

One area that usually creates a bottleneck when processing requests is the fact that, in certain circumstances, Apache sends queries to the DNS for each access. This behaviour is disabled by default since version 2.0. However, there is one case in which we should still make DNS queries for each request received: when we are using access control directives, such as `Allow`. In this case, wherever possible, it is advisable to use IP addresses instead of names.

3.3.2. Symbolic links and overrides

If we use the `FollowSymLinks` or `SymLinksIfOwnerMatch` options for each request, Apache must check whether it is a link and if any of the parent directories in the directory hierarchy is a symbolic link. This takes up a considerable amount of time for each access. Thus, we need to disable these options where possible or, if we need them on a specific disk space, limit their scope using the Apache configuration directives (`Directory` etc).

Additionally, if we use `AllowOverride` type di ectives, for each file access, Apache will look for a `.htaccess` file in the directory hierarchy preceding this file. As in the above case, we need to limit the scope of application of this directive as much as possible.

3.3.3. Memory mapping and sendfile

If our platform allows, we must check that Apache is using the operating system memory mapping capabilities to access file contents (`mmap`). This will generally increase performance considerably. However, you will need to consult the Apache documentation for your platform, as the performance of some operating systems is reduced with the use of `mmap`. Remember also that files accessible through units shared over a network (NFS for example) should not be mapped to memory.

Another operating system capability that substantially increases Apache's performance is use of the `sendfile` system all, which is a function provided by some operating systems, characterised by delegating the task of sending a file over the network to the operating system kernel. If we have this directive,

it is a good idea to check that Apache is using it in compilation time. However, we need to take the same precautions as `mmap`, that is, check that our platform is supported and that these files are not accessible from network disk drives.

3.3.4. Creating processes and threads

Another area where we can control performance of Apache is in the creation and instantiation of processes. On start-up, Apache creates a series of processes to attend requests. When a process has attended a certain number of requests, it finalises and another starts in its place. We can adjust this behaviour with:

- `MinSpareServers`: minimum number of server processes we need to have running.

- `MaxSpareServers`: maximum number of server processes not attending a request that we can have running.

- `StartServers`: number of server processes we can start.

- `MaxRequestsPerChild`: maximum requests that a process can attend before being recycled.

Another feature that can be used to control the operation of Apache is the processing module (MPM). By default, Apache works with a processing module based on system processes, called `prefork` but we an change it for one called `worker`, which also launches a series of threads for each system *process*. The latter is a good choice for systems with high loads.

With support from the
Education and Culture DG
Lifelong Learning Programme

This course book starts with an introduction to the Internet, including a brief history of the TCP/IP protocol and Worldwide Web. It defines the basic concepts for web servers and studies the case of Apache, the most used webserver, while other free software webservers are not forgotten. The course continues with webpage design focussing on HTML and JavaScript. XML schemas, their validation and transformation are covered as well as dynamic webpages built with CGI, PHP or JSP and database access.

Webservices are software components that are accessible through SOAP and have their interface described with WSDL (Web Service Description Language). In this section the XML-RPC protocol is discussed among other things.

The last part of the course deals with configuration, maintenance, monitoring and security aspects.

freeknowledgeinstitute
UOC Universitat Oberta de Catalunya www.uoc.edu
www.ou.nl
UNIVERSITETET I AGDER