PART I

TECHNOLOGY OVERVIEW

Chapter 1
Technologies for Web Applications

This Page Intentionally Left Blank

CHAPTER 1 Technologies for Web Applications

1.1 Introduction

Web applications are complex systems, based on a variety of hardware and software components, protocols, languages, interfaces, and standards. This chapter proposes a "guided tour" through the ingredients that characterize Web applications; the progression of arguments is such that every new development can be interpreted as the natural consequence of problems exhibited by previous technological solutions. This approach may help you in consolidating within a clear reference framework some background knowledge you might have already, but perhaps not systematically organized.

The review starts with the basic technologies for building Web applications: HTTP—the resource transfer protocol—and HTML—the language for writing

hypertexts. The first section also describes client-side scripts and components that make a Web interface more interactive. HTTP and HTML alone enable the development of simple Web sites, but are insufficient for large and complex applications; in particular, HTML is limited in the number and expressive power of its tags, which embed presentation features inside pages. However, the latest version of HTML (HTML 4), and specifically the introduction of cascading style sheets, moves in the direction of separating content from presentation, which is essential for large scale applications.

The second section of this chapter presents the *eXtensible Markup Language* (XML) as an evolution of the concepts proposed by HTML 4; it shows that XML is totally independent of "look and feel" aspects, and that presentation can be superimposed on XML documents by means of the eXtensible Stylesheet Language (XSL), a technology comparable to cascading style sheets. We also explain the structure of XML documents, originally represented by *Document Type Definitions* (DTDs) and more recently by XML schema definitions (XSDs), and introduce the notion of an XML document being *well-formed* and valid with respect to a DTD or XML schema.

While Sections 1.2 and 1.3 illustrate technologies not specifically related to data-intensive applications, the subsequent sections drill down into the data-intensive aspects. Section 1.4 briefly addresses data management with relational databases and the SQL language, two very consolidated technologies in use for over 30 years. An interesting aspect of the integration of relational databases into Web applications is the existence of interoperability standards, such as Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC), which simplify the access to data repositories produced by different vendors, by providing a uniform interface to developers.

Finally, Section 1.5 presents a broad spectrum of technologies specifically designed to support the dynamic construction of pages. We start by describing the Common Gateway Interface (CGI), as an extension of the HTTP protocol; then we discuss Java-based Web server extensions (the so-called servlet architecture) and server-side scripting (embodied in such languages as Java Server Pages [JSP] and Active Server Pages [ASP]); next, we present tag libraries as an evolution of server-side scripting; and finally we focus on complex, three-tier architectures, introducing the notion of Web Application Servers, and briefly comparing the two popular architectures Java 2 Enterprise Edition and Microsoft .NET. The chapter closes with a discussion of the next challenge of Web applications: the publishing of content for multiple devices with different rendition capabilities.

1.2 HTTP and HTML: The Foundation of Web Technology

We start the technology tour from the foundations, which lie in the HTTP protocol and in the HTML markup language.

1.2.1 Accessing Remote Resources: The Hypertext Transfer Protocol

The fundamental technology at the origin of Web applications is the well-known *HyperText Transfer Protocol* (HTTP), an application-level protocol for allowing users to make requests of resources to remote servers. HTTP is the invention of Tim Berners-Lee and Robert Cailliau, two researchers working at CERN, the European Nuclear Research Centre (formerly, Centre European pour la Recherche Nucleaire), located in Geneva, Switzerland. The origin of HTTP can be traced back to the World Wide Web (WWW) project, started in 1990 and aimed at building a distributed hypermedia system for accessing with a simple interface documents, reports, images, and online help stored in servers spread over a TCP/IP network.

Technically speaking, HTTP is a *client-server application protocol,* which defines the rules by which a client program, called a *browser* or user agent, and a server program, called a *Web server,* may interact in order to exchange requests and responses. In HTTP terminology, the user agent sends a request for a given resource to the Web server, which is a process running continuously and listening to requests coming from the network; upon receiving the request, the server locates or builds the resource and sends a response to the client (Figure 1.1). The basic resource requested by the client is an HTML page, which is a piece of text representing a multimedia hypertextual document. More generally, a request may address a file of any format stored in the Web server, or even the invocation of a program to be executed at the server side.

The HTTP protocol is conceived for exchanging resources distributed over the Internet and therefore exploits a standard resource addressing system. HTTP resources are identified by means of *Uniform Resource Locators* (URLs), which are structured strings of the format

```
http: // <host> [: <port>] [ <path> [? <query>]]
```

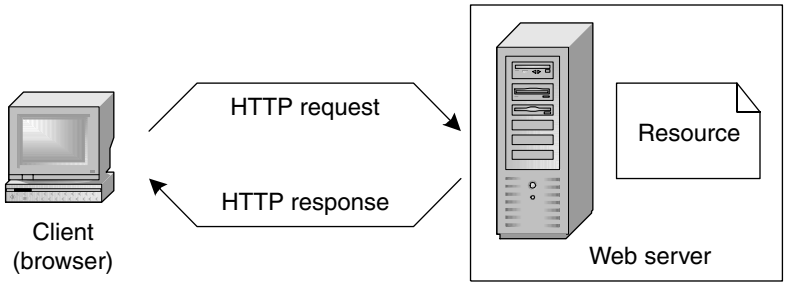

Figure 1.1 Request-response cycle of HTTP.

After the fixed prefix `http://`,[1] the URL contains the host name or IP address of the server, possibly followed by a port number (denoting the access to a specific server port), optionally followed by a pathname in the Web server file system pointing to the requested resource, optionally followed by a set of parameters, called the *query string*.

For example, the URL *http://www.theatlantic.com/unbound/flashbks/computer/bushf.htm* denotes the file named *bushf.htm,* stored in the directory named *unbound/flashbks/computer/,* in the file system managed by the Web server installed in the host named *www.theatlantic.com*.

HTTP requests are issued by the browser, either explicitly when the user types a URL in the "open page" command of the browser, or implicitly when the user clicks on an anchor within a page.

HTTP requests have a fixed format, which consists of three parts: a request line, some optional message headers, and the request body (also optional).

The request line is a formatted string, which consists of three parts; the HTTP method, the URL of the requested resource, and the protocol version. The most important HTTP methods are GET and POST, which respectively perform

- The submission of a plain request for some resource to the Web server.
- The submission of a request including sizeable user's input (e.g., a long text or a file) to be processed by the server. In such a case, the user's input is packaged as an attachment to the request, and constitutes the so-called request body.

For example, in order to display an HTML copy of the article "As You May Think," by Vannevar Bush published by the *Atlantic Monthly,* the browser may send the following GET request to the Web server installed in the host *www.theatlantic.com:*

```
GET /unbound/flashbks/computer/bushf.htm HTTP/1.1
```

The request includes the name of the method (GET), the URL of the requested resource, and the version of the HTTP protocol. Methods GET and POST will be further discussed in Section 1.5.1, in the context of the CGI protocol.

After receiving and interpreting a request message, a server responds with an HTTP response message, which is structured in three parts: a status line, a set of optional headers, and a message body.

---

[1] In a Web browser, the prefix may also refer to a different protocol (for instance, *ftp://*) supported by the browser.

The status line consists of the protocol version followed by a numeric status code and its associated message (for example: `HTTP/1.1 404 Not found`).

Because the HTTP protocol has only two very generic request methods and one kind of response, several optional fields (called headers) can be added to requests and responses to convey auxiliary information that the browser and the Web server must exchange. There are four kinds of headers: general, request-specific, response-specific, and entity headers, as follows:

- General headers are applicable both to requests and responses. For example, the Date header represents the date and time at which the request or response was originated.
- Request headers apply specifically to requests and not to responses. For example, the Accept-Language header establishes the set of languages that are preferred by the user as a response to the request.
- Response headers apply specifically to responses and not to requests. For instance, the WWW-Authenticate response-header field is used for access control; it is included in response messages having status code = 401 (which means "unauthorized"), emitted by the Web server when the user tries to access a resource stored in a password-protected domain.
- Finally, entity headers apply to the content transferred in the body of the request or response.[2] For example, the Content-Length header specifies the number of bytes associated with the body of the request or response.

Since its origin, HTTP has been subject to a deep revision process, which has produced two versions of the protocol: HTTP/1.0, the pioneering version, and HTTP/1.1, the stable version, approved as Draft Standard by the *Internet Engineering Task Force* (IETF) in July 1999.

Some important observations apply to HTTP, which are at the base of the evolution of Web application architectures:

- HTTP is *stateless:* Each HTTP request is treated by the Web server as an atomic and independent call. There is no difference between a sequence of two requests by different users or by the same user. As a consequence, HTTP is not capable of maintaining any information between two

[2]In the HTTP terminology, the word "entity" refers to the content transferred in the body of an HTTP request or response, which can be, for example, the HTML page returned to the browser, and the text submitted by the user in a form.

successive requests by the same user. In other words, there is not a notion of user session in HTTP. If the history of user interaction must be preserved, this feature must be implemented outside HTTP.

- HTTP is *pull-based:* Interaction occurs only when the client calls the server. There is no way in which the server may call back the client. Therefore, notification of clients cannot be implemented using HTTP alone.

1.2.2 Writing Web Documents: The Hypertext Markup Language

The most popular resources requested on the Web are HTML pages. An HTML page is a textual file written in the *Hypertext Markup Language,* a language based on a document composition style known as "markup."

Markup document processing is centered on the idea of inserting special tokens, called *tags,* in textual documents, in order to delimit selected portions of the text and express some properties of such text portions, like font color and size. With markup, the rendering of the document is separate from the creation of its content, and is delegated to a processor, which receives in input the marked-up text and transforms it into a rendered document, by interpreting the meaning of tags (Figure 1.2). In the Web context, the editing of HTML documents is done by the content producer using any text editor, and the processing is performed by the browser.

Syntactically, HTML tags are symbols delimited by angle brackets, like <FONT> or <TABLE>. Most often tags are used in pairs, because they must delimit a portion of text. In this case the same tag has two variants, the start tag, positioned at the beginning of the text to delimit, and the end tag, positioned at the

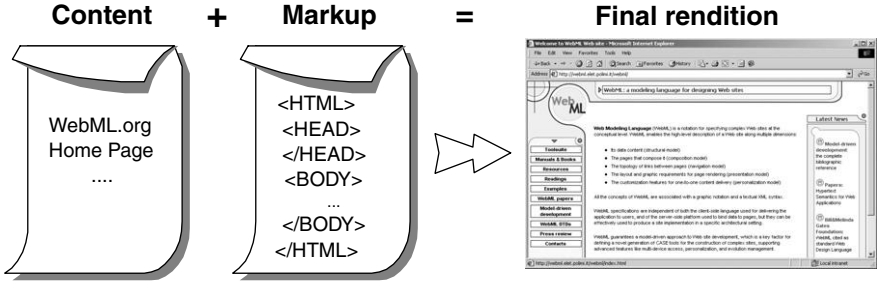

Figure 1.2 Processing of a marked-up text.

end; the end tag has the same name of the start tag, but is preceded by a "/" character.

The primary use of HTML tags is to give a general structure to the document. The entire document is delimited by the <HTML> tag, and contains two major sections: the header, delimited by the <HEAD> tag, and the body, delimited by the <BODY> tag, as shown in Figure 1.3.

The header section includes information about the document, for example the <TITLE> tag, which specifies the document title used by the browser to name the window where the document is rendered, or the <META> tag, which can be used to list keywords for document indexing by search engines. The body section is where the actual content is placed. In the example in Figure 1.3, the body

```
<HTML>
  <HEAD>
    <TITLE>A Simple Document</TITLE>
    <META name="keywords" content="HTML, tutorial">
  </HEAD>
  <BODY>
    <P align="center"><FONT size="+6">Hello world!</FONT></P>
  </BODY>
</HTML>
```

Figure 1.3 A simple HTML page and its rendition in the browser.

contains just the text "Hello world!", wrapped inside a paragraph and formatted with a larger font.

In the early versions of HTML, prior to HTML 4, the most common use of tags is to assign formatting properties to the text. The following example shows a portion of text delimited by the <B> (bold) tag, which sets the font style to bold:

```
The effect of inserting a bold tag pair <B> is to make the
included text bold.</B>
```

The effect of inserting a bold tag pair **is to make the included text bold.**

Sometimes a tag must be able to express a property having many possible values, like the size or the color of the font. To this end, tags may have attributes, which are label-value pairs. For example, the <FONT> tag has several attributes for setting the font properties, such as size and face:

```
The attributes of the FONT tag <FONT size="+1" face="Arial">
assign various properties to the font</FONT>
```

The attributes of the FONT tag assign various properties to the font.[3]

Table 1.1 summarizes some of the HTML tags for text formatting.

The most popular features of HTML are the two tags <A> and <IMG>, which are used to build multimedia hypertexts.

The anchor tag <A> delimits a portion of text, which is interpreted by the browser as the starting point (anchor, in the hypertext terminology) of a hypertext reference. The text is rendered in a special way (for example, underlined and in a different color) and clicking on it makes the browser issue an HTTP request for a resource to be displayed. The URL of the new resource, which is the target of the hypertext reference, is specified in the `href` attribute of the anchor tag, as highlighted in the example in Figure 1.4 on page 12.

A possible use of the anchor tag, besides expressing a hypertext link, is to open the default mailer with a click. To achieve such an effect, the `href` attribute specifies an e-mail address instead of an HTTP URL, as shown in Figure 1.5 on page 13.

[3]As will be explained later, the use of the <FONT> tag, and of similar tags that express graphic and formatting properties, is deprecated in HTML 4. The examples illustrate the original meaning of HTML primitives, not the most correct way of using HTML.

Table 1.1 HTML tags for text formatting.

Tag	Attributes	Meaning
<BODY>	bgcolor, background, text, link	Sets properties of the entire document, such as background color, background image, text, and link color.
<B>		Sets the font style to bold.
<I>		Sets the font style to italic.
<U>		Sets the font style to underlined.
<PRE>		Tells the browser to render the text in a fixed-pitch font, to preserve whitespace, and not to wrap long lines.
<FONT>	size, color, face	Assigns font properties.
<P>	align	Delimits a paragraph and sets text horizontal alignment (left, center, right, justify).
<H1>	align	Delimits a heading of level 1 (there are six levels of headings) and sets the horizontal alignment.

If the <A> tag introduces hypertext features into HTML, the <IMG> tag adds multimedia aspects. The <IMG> tag inserts an inline image into the document, the URL of which is specified using the `src` attribute, as shown in Figure 1.6 on page 14.

When the browser renders the HTML page and encounters the <IMG> tag, it sends an additional HTTP request to the Web server for the file mentioned in the `src` attribute, and displays the image received by the Web server inside the HTML document.

Table 1.2 (page 14) illustrates the most common attributes of the <IMG> tag. The image tag can be nested into the anchor tag, to produce clickable images, which act as anchors of a hypertext reference.

1.2.3 Client-Side Scripting for Enhancing Page Interactivity

HTML was originally conceived for writing hypertexts. However, the development of interactive applications, for example applications supporting form-based data entry, requires additional capabilities, like the possibility of defining events produced by the interaction of the user with the HTML page, and procedures for reacting to such events.

```
<HTML>
  <HEAD>
    <TITLE>A Simple Hypertext Anchor</TITLE>
    <META name="keywords" content="HTML, tutorial">
  </HEAD>
  <BODY bgcolor="#FFFFFF" text="#000000">
    <P align="left">
      <FONT size="+6"> Click
        <A href="http://www.webml.org">HERE</A> to open the WebML home page
      </FONT>
    </P>
  </BODY>
</HTML>
```

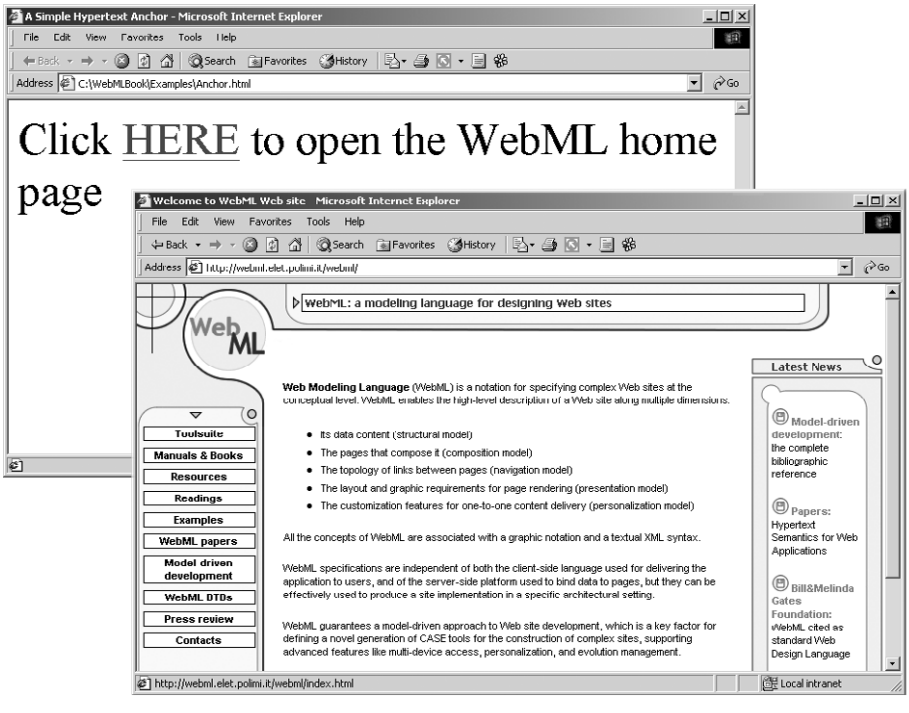

Figure 1.4 HTML page containing a hypertext anchor.

```
<HTML>
  <HEAD>
    <TITLE>A Simple Hypertext Anchor</TITLE>
    <META name="keywords" content="HTML, tutorial">
  </HEAD>
  <BODY bgcolor="#FFFFFF" text="#000000">
    <P align="left">
      <FONT size="+5"> Click <A href=
"mailto:piero.fraternali@polimi.it">HERE</A>to send me email</FONT>
    </P>
  </BODY>
</HTML>
```

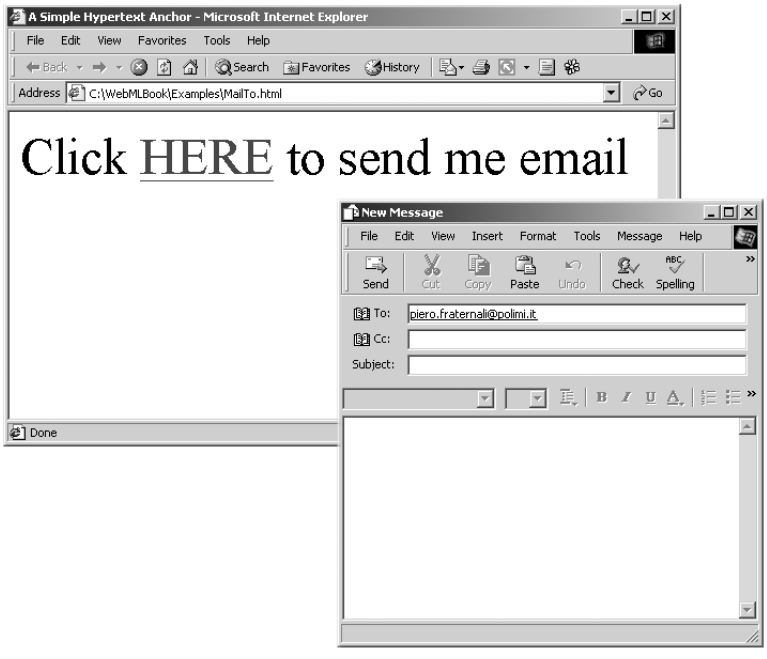

Figure 1.5 HTML page containing a "mailto" anchor.

```
<HTML>
  <HEAD>
    <TITLE>A Simple Image</TITLE>
    <META name="keywords" content="HTML, tutorial">
  </HEAD>
  <BODY bgcolor="#FFFFFF" text="#000000">
    <P align="center">
      <IMG src="http://www.polimi.it/images2000/home/testata.gif">
    </P>
  </BODY>
</HTML>
```

Figure 1.6 HTML page containing an image tag.

Table 1.2 Essential attributes of the IMG tag.

Attributes of the image tag	Meaning
`width, height`	The screen space reserved to the image, in pixels.
`align`	The image alignment (top, middle, bottom, left, right).
`border`	Width of the image border in pixels.
`alt`	Alternative text, to be displayed when the image is not available, or in textual browsers.

HTML pages can be made more interactive by inserting into their code small programs, called *client-side scripts,* which are executed by the browser when the page is loaded or when the user produces some event.

The most popular languages for creating client-side scripts are JavaScript, by Netscape, and VisualBasic script, by Microsoft. Scripts are inserted inside a page using the `<SCRIPT>` tag, placed either in the head section of the HTML document, or in its body. Scripts in the head section are executed when they are explicitly called, or when an event is triggered; scripts in the body section are executed when the HTML page is loaded.

The example in Figure 1.7 uses JavaScript code to write the current date. The script is placed in the document body and is executed when the page is loaded in the browser. The instruction `document.write` in the `<SCRIPT>` tag makes the argument of the write function (for example, the current year calculated by the function call `d.getFullYear()`) to be output in the document, in the position where the `<SCRIPT>` tag appears. The result of loading the page in the browser appears in Figure 1.7 on page 16.

Client-side scripts can be activated after the occurrence of specific events. In this case, the scripts are placed in the head section of the document, and the event that triggers script execution is declared as an attribute of the appropriate HTML element. For example, the page in Figure 1.8 on page 17 contains the anchor text "Click here to enter my favorite Web site." The anchor tag includes the declaration of the `onClick` event, which is raised when the anchor text is clicked. The event triggers a JavaScript function named `dialog`, which opens a confirmation dialog box asking the user if he/she really wants to access the page linked to the anchor.

Client-side scripting is particularly useful in conjunction with the `<FORM>` tag, an HTML element for constructing data entry forms. In this case, events can be attached to the form fields and may trigger functions for validating the user input, for example, for controlling the validity of a date. Forms are treated in Section 1.5.1.

1.2.4 Client-Side Components for Enhancing Client Functionality

Client-side scripting is not the only way to add interactive functions to an HTML page. A more powerful feature is offered by client-side components, which are executable objects plugged into the page. Examples of pluggable components include Java applets, Microsoft ActiveX controls, Macromedia Flash movies, and more.

Similarly to scripts, client-side components are executed by the browser; unlike scripts, client-side components are full-fledged executable objects,

```
<HTML>
<BODY>
This is the date of today: <SCRIPT type="text/javascript">
var d = new Date()                  // create an object of type date
document.write(d.getMonth() + 1)    // get the month and print it
document.write("/")                 // print a separator
document.write(d.getDate())         // get the day and print it
document.write("/")                 // print a separator
document.write(d.getFullYear())     // get the year and print it
</SCRIPT>
</BODY>
</HTML>
```

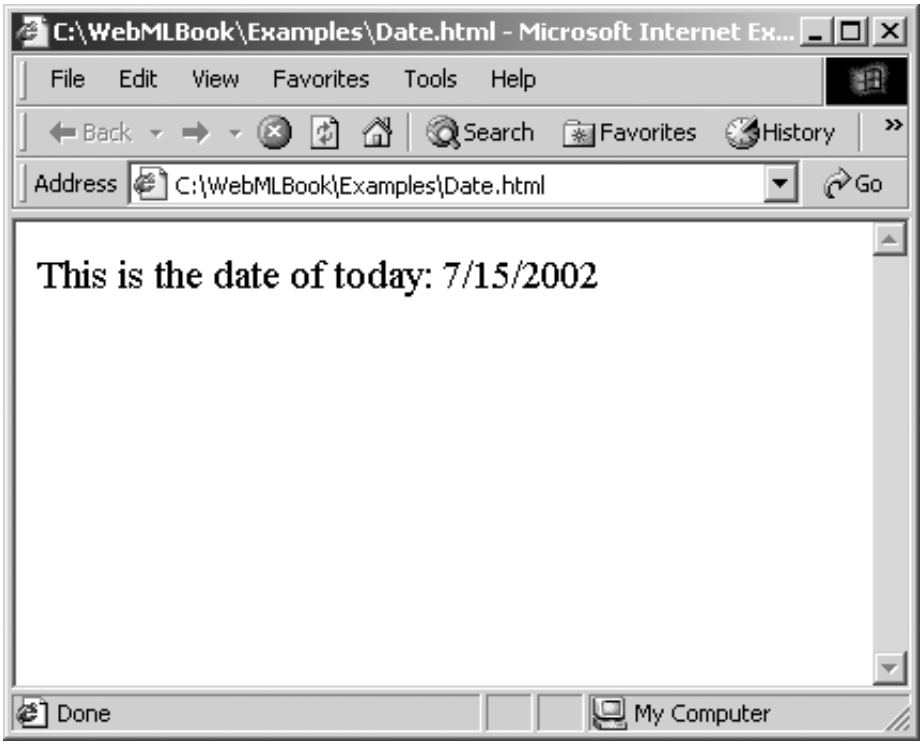

Figure 1.7 Example of client-side scripting in JavaScript.

```
<HTML>
<HEAD>
  <SCRIPT type="text/javascript">
    function dialog()
      {if (confirm('Are you sure you want to enter?'))
        window.location.href='http://home.netscape.com/'}
</SCRIPT>
</HEAD>
<BODY>
  <A href="http://home.netscape.com/" onClick="dialog()">
        Click here to enter my favorite Web site</A>
</BODY>
</HTML>
```

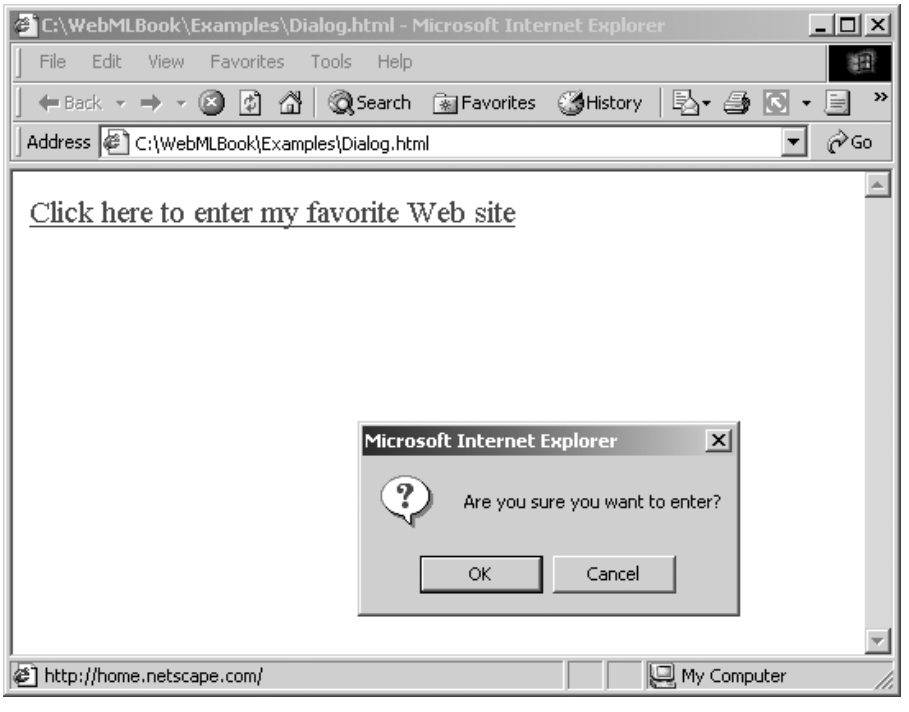

Figure 1.8 Example of client-side scripting with the script in the head section.

programmed by a technically skilled developer using a language like Visual Basic or Java and stored in a Web server, wherefrom they are downloaded by the browser. For example, client-side components written in Java, also known as *Java applets,* can be downloaded from the network and run inside the browser; they may implement complex functions, like showing in real time a graph of stock quotations, but are prevented for security reasons from executing potentially dangerous instructions, like reading and writing files on the file system, and making network connections to hosts different from the one from which they have been downloaded.

Client-side components are inserted into an HTML page using the `<OBJECT> tag`. For example, Figure 1.9 shows an HTML page containing an ActiveX component, a text rotator, which constructs an animation by taking an existing file and scrolling its content in a delimited portion of the screen. As shown in the code reported in Figure 1.9, the `<OBJECT>` tag used to insert the component in the HTML page is similar to the `<IMG>` tag, in the sense that it reserves a certain amount of space in the page for the component, using the attributes `width` and `height`.

The `classid` attribute is analogous to the `src` attribute of an image, in the sense that it references the actual object that implements the client-side component. The `<OBJECT>` tag also admits some initialization parameters, expressed by one or more nested `<PARAM>` tags. In the example of Figure 1.9, the client-side component scrolls the content of a Web page, whose URL is specified in the `<PARAM>` element named szURL, along the horizontal or vertical axis, according to the values of the `ScrollPixelsX` and `ScrollPixelsY` parameters. Y=10 and X=0 result in scrolling a text vertically from top to bottom.

1.2.5 The Evolution of HTML: HTML 4 and Cascading Style Sheets

The HTML language has been subject to some criticism by the community of professional graphic designers, due to its fixed set of tags and limited graphic capabilities. Achieving sophisticated graphic effects with the standard HTML tags is hardly possible; on the other hand, introducing new tags in the language requires a worldwide standardization process, which may take several years.

In response to these problems, the World Wide Web Consortium (W3C), the governing body devoted to the Web standards, issued the latest version of the HTML language, called HTML 4, in December 1997. HTML 4 introduces several enhancements over the previous versions, including a better separation of document structure and presentation, improved support for accessibility of documents and internationalization, and a better definition of complex constructs like

```
<HTML>
<HEAD>
  <TITLE>Marquee Control</TITLE>
</HEAD>
<BODY>
  <H1>The OBJECT Tag </H1>
  <P>
  <OBJECT id="Marquee1" width="500" height="300" align="top" border="1"
          classid="CLSID:1A4DA620-6217-11CF-BE62-0080C72EDD2D">
    <PARAM name="szURL" value="text.htm">
    <PARAM name="ScrollPixelsY" value="10">
    <PARAM name="ScrollPixelsX" value="0">
  </OBJECT>
</BODY>
</HTML>
```

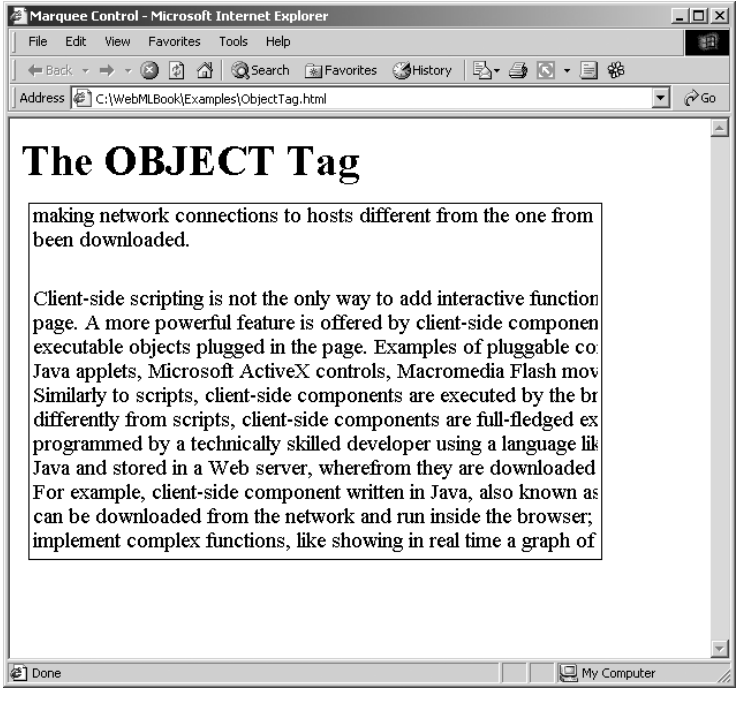

Figure 1.9 Example of ActiveX component: The text in the box rotates vertically.

frames and tables. However, the most innovative feature of HTML 4 is the introduction of *cascading style sheets* (CSS), a new technique for defining the presentation of HTML documents, separating the look and feel from markup and textual content.

A style sheet is a set of rules that tell a browser how to render a document. A rule is made of two parts: a selector, which specifies the HTML tag to which the style rule applies, and a style declaration, which expresses the style properties to be attached to the HTML tag mentioned in the selector. As an example, the following code fragment defines the `color` and `font-size` properties for the `<H1>` and `<H2>` tags:

```
<HEAD>
  <TITLE>CSS Example</TITLE>
  <STYLE type="text/css">
    H1 { font-size: 20pt; color: red }
    H2 { font-size: 18pt; color: blue }
  </STYLE>
</HEAD>
```

The CSS standard includes more than 60 rendition properties, which permit the designer to define the appearance of HTML documents in a much more precise way than with HTML 3.

In the above example, style rules are embedded in the document to which they apply, by means of the `<STYLE>` tag. The same effect can be obtained by storing the two style rules in a separate file, and by linking the HTML document to the style file, using the `<LINK>` tag in the document head. In the following example, the HTML document is linked to a style file named style.css:

```
<HEAD>
  <TITLE>CSS Example</TITLE>
  <LINK rel=StyleSheet href="style.css" type="text/css" >
</HEAD>
```

The `rel` attribute in the `<LINK>` tag expresses the meaning of the link between the document and the resource specified in the `href` attribute. The value `StyleSheet` declares that the external file must be used as a presentation style sheet.

Separating the presentation rules from the document content has an immediate advantage: if multiple documents share the same style rules, these rules can be placed in a single file and linked to all documents. As a consequence of this separation, updating the style rules requires modifying the single style file, instead of updating all documents.

1.3 XML: eXtensible Markup Language

Although HTML 4 is a great improvement over the preceding versions of HTML in the direction of better presentation, the imposition of a fixed tag set prevents developers from defining their own tags that could match the requirements of specific application domains. At the same time, the advent of cascading style sheets, and the consequent separation of the presentation rules from the HTML markup, makes HTML obsolete for content formatting.

A stronger position with respect to the separation of presentation, content, and markup is taken by the *eXtensible Markup Language (XML),* whose base specification (XML 1.0) became a W3C Recommendation in February 1998.

1.3.1 User-Defined Tags

XML is a standard syntax for defining custom tag collections. Unlike HTML, which consists of a fixed set of tags, XML is a meta-language (that is, "a language for defining languages"), which standardizes the syntactic rules whereby users can define their own sets of tags, suited to the needs of a specific application domain.

A *well-formed* XML document is a piece of marked-up content that obeys a few syntactic rules:

- The document must start with a standard line, declaring the language version, such as: `<?xml version="1.0"?>`
- All tags, called *elements* in XML terminology, can enclose some content, which can be text or other tags. XML elements have an opening tag and a closing tag. The latter is obtained by prefixing the opening tag by means of the symbol "/". Exception to this rules are the tags with no content, which may have no closing tag—but in such a case they must have the "/" symbol at the end of the tag name, as in `<emptytag/>`.
- The document must have one root element, and the nesting of elements must be well-formed, which means that any element containing an inner element must not be closed before the closing of the inner element.
- Elements may have attributes with given values, and attribute values must be delimited by quotes (" ").

The following example presents a short, but well-formed, XML document:

```
<?xml version="1.0"?>
  <root>
    <child>
      <subchild>..some content...</subchild>
```

```
    </child>
  </root>
```

The document starts with a standard line that declares the XML version, and then contains some custom tags. As another example, a fragment of the outline of this book could be represented in XML as shown in Figure 1.10. As illustrated in the example, XML elements may have different kinds of content:

- *Element content:* Contains other elements, like the <book> element.
- *Text content:* Contains character data, like the <chapter> element.

```
Designing Data-Intensive Web
Applications

Stefano Ceri, Piero Fraternali,
Aldo Bongio, Marco Brambilla,
Sara Comai, Maristella Matera

Part I:    INTRODUCTION
Chapter 1  Technologies for Web
           Applications

PART II:   CONCEPTUAL MODELING
Chapter 2  Data Model
Chapter 3  Hypertext Model
Chapter 4  Content Management
           Model
Chapter 5  Advanced Hypertext
           Model
...
```

```
<book>
<publishing schedule="10-31-2002"/>
<title> Designing Data-Intensive Web
Applications </title>
<author> Stefano Ceri </author>
<author> Piero Fraternali </author>
<author>Aldo Bongio </author>
<author> Marco Brambilla </author>
<author> Sara Comai </author>
<author> Maristella Matera </author>
<part> Technology Overview
<chapter> 1.Technologies for Web
            Applications </chapter>
</part>
<part> Models for Designing Web
Applications
<chapter> 2.Data Model </chapter>
<chapter> 3.Hypertext Model </chapter>
<chapter> 4.Content Management
            Model </chapter>
<chapter> 5.Advanced Hypertext Model
</chapter>
</part>
..
</book>
```

Figure 1.10 An example of XML tags for representing the outline of a book.

- *Mixed content:* Contains other elements and/or character data, like the <part> element.
- *Empty content:* No content, like the <publishing> element.

Besides content, XML elements may have attributes, like the schedule attribute in the <publishing> element.

An XML document may be associated with a *Document Type Definition (DTD),* prescribing the common format of a class of XML documents. A DTD includes the description of the elements that can be used in the document, and for each element specifies the admissible content and attributes.

A DTD contains three categories of declarations: element, attribute, and entity declarations. An element declaration introduces an element and specifies its admissible content; an attribute declaration specifies which attributes can be put inside an element and expresses a few properties of such attributes; an entity declaration introduces a sort of "constant," which is a reference to some fixed piece of content. We do not further discuss entity declarations, although we next illustrate a few examples of element and attribute declarations.

A DTD for structuring documents about books may include *element declarations* like the ones in the following example:

```
<!ELEMENT book (publishing, title, editor?, author+,
                (chapter*|part*))>
<!ELEMENT publishing EMPTY>
<!ELEMENT title            (#PCDATA)>
<!ELEMENT editor           (#PCDATA)>
<!ELEMENT author           (#PCDATA)>
<!ELEMENT chapter          (#PCDATA)>
<!ELEMENT part             (#PCDATA|chapter)*>
```

The above rules declare seven elements: book, publishing, title, editor, author, chapter, and part. Element book has a complex content model: it may contain a sequence of subelements, denoted by the comma-separated list of element names. Specifically, the book element must contain one subelement of type publishing, one subelement of type title, zero or one (denoted by the "?" symbol) subelement of type editor, one or more (denoted by the "+" symbol) subelements of type author, and zero or more (denoted by the "*" symbol) chapters or parts. Chapters and parts are in *alternative* (denoted by the "|" symbol): either the book is organized in parts or in chapters. The publishing element has no content (EMPTY), the title, editor, author, and chapter elements have text data (PCDATA) as content. Finally, the part element contains zero or more chapters mixed with text data.

An *attribute declaration* lists all the attributes that an element may include, and poses some constraints on their values. For example, the attributes of the publishing element may be declared as follows:

```
<!ATTLIST publishing
  schedule    CDATA                  #REQUIRED
  editor      CDATA                  #IMPLIED
  format      (paperback|hardback)   "paperback"
>
```

The ATTLIST clause introduces three attributes for the publishing element: schedule, editor, and format. The schedule attribute consists of character data (CDATA) and is mandatory (#REQUIRED). The editor attribute has also character data as a value, but is optional (#IMPLIED). Finally, the format attribute is optional and may have a value chosen from a fixed set of options (paperback, hardback), with paperback as the default value assigned to the attribute when the user does not include the attribute in the publishing element.

A document that conforms to a given DTD is said to be *valid* with respect to that DTD. For example, the document of Figure 1.10 is valid with respect to the DTD expressed by the above clauses defining elements and attributes for describing books.

The DTD can be either placed inside the XML document, or stored in a separate file, as shown by the following example:

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
..
</book>
```

The line: <!DOCTYPE book SYSTEM "book.dtd"> defines the type of the document by referring to the file book.dtd, where the DTD declarations are stored.

DTDs present several limitations in expressing the structure of documents: they do not allow you to specify data types for the content of elements and attributes other than character data, and are unable to express several useful constraints on the nesting of elements. To improve the document structure specification, DTDs can be replaced by *XML schema definitions (XSDs)*. An XML schema definition is an XML document, which dictates the structure of a family of XML documents, using a standard set of tags for element declaration, defined by the XML Schema specification. XML Schema became a recommendation of the World

Wide Web Consortium in May 2001 and is gradually replacing DTDs in those applications that require a more precise description of XML document structure.

Figure 1.11 shows an example of an XSD, corresponding to the structure of the XML document of Figure 1.10. Being an XML document, the XSD starts with the XML version declaration (line 1), followed by the <schema> element, which encloses all the element definitions. The xmlns attribute of the <schema> element also imports the definition of the XML Schema tags used to describe the

```
1  <?xml version="1.0"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <!-- definition of book element-->
5  <xs:element name="book">
6    <xs:complexType>
7      <xs:sequence>
8        <xs:element ref="publishing"/>
9        <xs:element ref="title"/>
10       <xs:element ref="editor" minOccurs="0"/>
11       <xs:element ref="author" maxOccurs="unbounded"/>
12       <xs:choice>
13         <xs:element ref="chapter" minOccurs="0" maxOccurs="unbounded"/>
14         <xs:element ref="part" minOccurs="0" maxOccurs="unbounded"/>
15       </xs:choice>
16     </xs:sequence>
17   </xs:complexType>
18 </xs:element>
19
20 <!-- definition of chapter, title, editor and author elements -->
21 <xs:element name="chapter" type="xs:string"/>
22 <xs:element name="title" type="xs:string"/>
23 <xs:element name="editor" type="xs:string"/>
24 <xs:element name="author" type="xs:string"/>
25
```

(continued)

Figure 1.11 XML Schema definition.

Figure 1.11 *(continued)*

```
<!-- definition of part element->
<xs:element name="part" mixed="true">
  <xs:complexType>
    <xs:element ref="chapter" minOccurs="0" maxOccurs="unbounded"/>
  </xs:complexType>
</xs:element>

<!-- definition of publishing element -->
<xs:element name="publishing">
  <xs:complexType>
    <xs:attribute name="schedule" type="xs:date" use="required"/>
    <xs:attribute name="editor" type="xs:string"/>
    <xs:attribute name="format" default="paperback"/>
      <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="paperback"/>
        <xs:enumeration value="hardback"/>
        </xs:restriction>
      </xs:simpleType>
    <xs:attribute/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

document structure. These tags are organized in a so-called XML *namespace,* specified in the document *http://www.w3.org/2001/XMLSchema*. All the tags belonging to the same namespace have a name starting with a common prefix; in the case of XML Schema tags, the prefix is `xs`.

The `<schema>` element encloses the definition of the element types for describing books. The root element `<book>` (lines 4–18) is declared as a complex type, because it contains several subelements (`publishing, title, editor` and so on). The `<sequence>` element inside the declaration of element `<book>` specifies the required order of the nested subelements (`publishing`, followed by `title`, and so on). The `<choice>` tag is used for specifying that an element can

contain one of a set of subelements; in the example, element <book> may contain as subelement either chapter or part. For each element, occurrence indicators define how often an element can appear. In particular, the maxOccurs attribute specifies the maximum number of times an element can occur, whereas the minOccurs indicator specifies the minimum number of times an element can occur. The default values of the occurrence indicators are 1.

Elements chapter, title, editor, and author (declared at lines 21–24) are of type string, one of the basic types provided by XML schema, which also includes the decimal, integer, Boolean, date, and time types. Element part (lines 27–31) may contain both plain text and chapter elements: this feature is specified by setting the mixed attribute to true (line 27), and by defining the nontextual subelement of part (lines 28–30).

Finally, element publishing (lines 33–47) has empty content and three attributes. The three attributes are declared in the same way as elements; they are grouped into the definition of a complex type, which does not include the <sequence> element, to denote that the attributes can occur in any order. Attribute schedule is of type date and is required; editor is string-typed and optional (which needs not be explicitly specified); the type of attribute format is a string, whose content is restricted to a set of predefined values, enumerated inside the <restriction> tag (lines 40–43). The acceptable values are "paperback" and "hardback", with "paperback" as the default (line 38).

Like DTDs, XML schemas can be placed inside the XML document or in a separate file, referenced inside the document.

1.3.2 Presenting XML Documents Using XSL

XML is a platform-independent standard for describing and exchanging data, and not a markup language for presenting multimedia hypertext on the Web. The intended uses of XML cover a variety of applications, including electronic data exchange, document storage and transformation, document exchange in e-commerce and B2B transactions, and more. However, XML documents may also be presented to human readers, for example to Web users using a browser.

The best approach to displaying an XML document is to transform it into another document, encoded using a presentation-oriented markup language such as HTML. The transformation of XML documents is supported by the *eXtensible Stylesheet Language* (XSL).

XSL is a language for writing pattern-matching rules, similar to the rules of HTML 4 cascading style sheets. An XSL rule contains a matching part, for

selecting the target XML elements, and an action part, to transform or format the matched elements.

XSL rules are much more powerful than CSS rules, both in matching primitives and in transformation capability. As a consequence, XSL is a much more complex language than CSS; actually, XSL is the union of three sub-languages:

- *XPath:* A language for writing expressions that define a portion of an XML document.
- *XSL Transformations* (XSLT): A language for defining transformations of XML documents.
- *XSL Formatting Objects* (XSL-FO): A language for defining the rendition of a document.

XSL is now a standard recommended by the W3C. The first two parts of the language became a W3C Recommendation in November 1999. The full XSL Recommendation including XSL Formatting Objects became a W3C Candidate Recommendation in November 2000.

The most common approach for presenting an XML document is to transform it into HTML; for this transformation, the XPath and XSLT sub-languages are sufficient. The transformation applies to an input document and produces an output HTML document. The processing is specified by writing an XSL style sheet, which consists of several rules written in XSLT. Each XSLT rule uses XPath to define parts of the source document that match one or more predefined templates. When a match is found, the XSLT rule transforms the matching part of the source document into some content in the resulting document.

The XSL style sheet in Figure 1.12 illustrates the XSLT rules that transform the XML document of Figure 1.10 into the HTML page shown in Figure 1.13 on page 30. We will show the flavor of XSL by walking through the example.

An XSL style sheet is a particular kind of XML document, containing pattern-matching rules called *templates*. The example in Figure 1.12 contains five templates. We will describe the first one completely (lines 5–18), and omit the detailed illustration of the other rules, for brevity.

The template starts with the `<xsl:template match="book">` clause (line 5), which introduces an XSL rule matching all the book elements of the input XML document. The meaning of such template is that if the match succeeds, then the document fragment contained inside the template (lines 6–17) is inserted into the output document. The document fragment inside the template may include plain tags in some document formatting language (HTML in Figure 1.12), and XSL tags denoting calls to other XSL rules. For example, the `<TITLE>` tag (line 8) contains a rule invocation (`<xsl:value-of select="title"/>`), which extracts

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3                  version="1.0">
4
5    <xsl:template match="book">
6      <HTML>
7        <HEAD>
8          <TITLE><xsl:value-of select="title"/></TITLE>
9        </HEAD>
10       <BODY>
11         <CENTER><H1><xsl:value-of select="title"/></H1></CENTER>
12         <CENTER><B><xsl:apply-templates select="author"/></B></CENTER>
13         <xsl:apply-templates select="publishing"/>
14         <HR/>
15         <xsl:apply-templates select="part"/>
16       </BODY>
17       </HTML>
18   </xsl:template>
19
20   <xsl:template match="author">
21     <xsl:if test="position() &gt; 1">, </xsl:if>
22     <xsl:value-of select="text()"/>
23   </xsl:template>
24
25   <xsl:template match="publishing">
26     <P align="right">
27       <I>Publishing date: <xsl:value-of select="@schedule"/></I>
28     </P>
29   </xsl:template>
30
31   <xsl:template match="part">
32     <P><H3><xsl:value-of select="text()"/></H3></P>
33     <xsl:apply-templates select="chapter"/>
34   </xsl:template>
35
36   <xsl:template match="chapter">
37     <P><H4>- <xsl:value-of select="text()"/></H4></P>
38   </xsl:template>
39 </xsl:stylesheet>
```

Figure 1.12 Document transformation in XSLT.

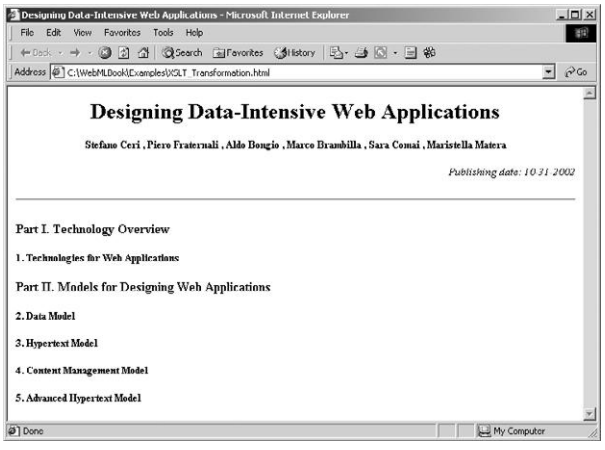

Figure 1.13 The HTML page resulting from applying an XSLT transformation to the document in Figure 1.10.

the book's title from the input XML document. As an effect of such a call, the title of the output HTML document will coincide with the book's title extracted from the input XML document. The <BODY> elements contains four more rules calls. The first rule invocation (line 11) extracts again the book's title and places it inside a pair of <H1> tags, which produces the first line of the page shown in Figure 1.13. The rule invocations at lines 12, 13, and 15, respectively, match authors, publishing information, and part elements, producing the HTML output shown in Figure 1.13.

Note that the XSL call for extracting the book title differs from those for extracting the publishing information, the authors, and the book parts. The former extracts just an atomic piece of text from an XML element, which is achieved using the basic XSL clause `xsl:value-of`; the latter require more elaborate matching and output construction, which is delegated to another template, called using the `xsl:apply-templates` clause.

The templates called for producing the output may be very simple, as the ones for extracting the publishing information (lines 25–29) and the authors (lines 20–23), which just return either the textual content of XML elements or the value of an attribute. Templates can be more complex and activate other templates, as the rule for extracting the data of a book part (lines 31–34), which contains a nested rule invocation (line 33) for finding the data of the various chapters that constitute a part.

Transformations of XML documents can be achieved also by means of the *XQuery language* (also known, in conjunction with XPath, as *XML Query*), a W3C standard proposal addressing the efficient querying of XML data. XQuery is still

```
1   <HTML>
2   <HEAD><TITLE>{//book/title/text()}</TITLE></HEAD>
3   <BODY>
4       <CENTER><H1>{//book/title/text()}</H1><CENTER>
5       <CENTER>
6         <B>
7           {//book/author[position()=1]/text()}
8           {FOR $a IN //book/author[position() > 1]
9            RETURN ", ", {$a/text()}
10          }
11        </B>
12      </CENTER>
13      <P align="right"><I>{//book/publishing/@schedule/text()}</I></P>
14      <HR/>
15        {FOR $p IN //book/part
16         RETURN
17          <P><H3>$p/text()</H3></P>
18          {FOR $c IN $p/chapter
19           RETURN <P><H4>$c/text()</H4></P>
20          }
21        }
22  </BODY>
23  </HTML>
```

Figure 1.14 Document transformation in XQuery.

a Working Draft, but several tools and databases already offer XQuery processing. The official XQuery recommendation is expected for early 2003.

Queries and document transformations are written as path expressions, specified in XPath, or as FLWR[4] expressions, a syntax for specifying iterations and for binding variables to intermediate results. To give the flavor of the language, the example in Figure 1.14 shows the XQuery version of the same transformation expressed by the XSL style sheet of Figure 1.12.

The transformation mixes HTML tags and XQuery expressions, which are replaced in the output document by the result of their evaluation. The result of

[4]FLWR stands for For-Let-Where-Return, pronounced as the word "flower."

the transformation shown in Figure 1.14 is an HTML document, wrapped inside the two HTML tags (lines 1 and 23). The HEAD tag includes the title of the book (line 2), which is determined by evaluating the XPath expression `//book/title/text()`. The BODY contains a heading where the title is repeated (line 4), and the list of authors (lines 5–12), centered and in bold face: the first author is obtained evaluating the expression `{//book/author[position()=1]/text()}` (line 7); then the subsequent authors are processed one at a time by means of the FOR clause, which iterates over the set of authors obtained by evaluating the expression `//book/author[position() > 1]`, which selects all the authors whose position is greater than 1. For each author retrieved in the loop (which is bound to variable `$a`), a comma is inserted, followed by the text of the AUTHOR element (lines 8–9). Then the value of the schedule attribute is enclosed in a paragraph (line 13), and, after a horizontal line (line 14), the parts and chapters of the book are inserted (lines 15–21): for each part of the book, bound to variable `$p` (line 15), the part name is displayed in a paragraph (line 17), and for each chapter of the part, bound to variable `$c` (line 18), the title of the chapter is displayed (line 19).

1.4 SQL: The Structured Query Language for Relational Databases

If HTTP and HTML/XML are the building blocks of the Web, SQL and relational databases are the cornerstones of information systems. Because relational databases are the leading products for building information systems, and over 80% of the pages found on the Web are actually built from database content, relational technology can be legitimately considered a key ingredient of data-intensive Web applications.

The main principle of the relational technology is extremely simple: data are formatted in tables (called relations) consisting of rows and columns. Each table stores the "facts" about a distinct concept of the application domain, like products or customers, organized as rows of elementary values. For example, the database in Figure 1.15 includes two tables: PRODUCT and CUSTOMER. The product table has three columns: CODE, NAME, and PRICE, which respectively store the code, name, and price of each distinct product.

To uniquely identify the facts stored in a table, it is a good practice to define that a group of columns form a *key* of the table. This definition implies that two rows cannot coexist in the table if they have coincident values of all the key columns. Therefore, the key of a table gives a unique "identity" to each one of the table rows. In the example of Figure 1.15, the key of table `product` is the CODE

PRODUCT		
CODE	NAME	PRICE
100	Aladdin	35
101	Blue moon	67
102	Casper	40
104	Dundee	21

CUSTOMER		
ID	NAME	ADDRESS
1	Acme Inc.	1 First St. Memphis TN
2	Gizmo Inc.	2 Second Av. New York NY
3	Widget Inc.	5 Third Av. S. Francisco CA
4	Supply Inc.	8 Fifth Rd. London UK

Figure 1.15 Tables in a relational database.

PURCHASE		
PROD-CODE	CUST-ID	QUANTITY
100	1	30
102	1	15
104	4	43
101	3	30
101	4	30

Figure 1.16 Table storing purchases of products performed by customers.

column, and the key of table `customer` is the `ID` column. These columns are underlined in Figure 1.15 to denote that they are the key of the respective table.

The notion of *key* is central in relational databases, because it permits the expression of semantic relationships between the objects of the application domain. For example, the relationship between the customers and the products they have purchased, along with the information of the total quantity of each product purchased by each customer, can be represented using the table illustrated in Figure 1.16. Each row in the table specifies the product (`PROD-CODE`), the

customer (CUST-ID), and the total purchased quantity (QUANTITY). The columns PROD-CODE and CUST-ID constitute the key of the PURCHASE table, because it does not make sense to allow multiple rows recording the total purchased quantity *for the same pair (product, customer)*.

The three tables shown in Figure 1.15 and Figure 1.16 constitute a small database. By combining the data stored in the three tables it is possible to extract complex information; for instance, which products were bought by "Acme Inc." or how many "Aladdin" lamps were bought by all the customers over time. The combination of information occurs by comparing values in the columns that constitute the keys of the three relations; such a "navigation" from one table to another one is expressed by means of query languages.

The *Structured Query Language* (SQL) is the most popular language for extracting and manipulating the information stored into relational tables. SQL offers the SELECT statement for querying data, and the INSERT, DELETE, and UPDATE statements for changing the database content.

A SELECT statement permits you to extract the desired information from one or more tables, formatted as a table of result rows. The basic form of a SELECT statement is:

```
SELECT column-list
FROM table-list
WHERE condition
```

The meaning of the statement is to build a table composed of the columns specified in the column list (SELECT clause), assembled from the values of the rows of the tables mentioned in the table list (FROM clause) that satisfy some condition (WHERE clause).

For example, to retrieve the name and price of all products costing less than $50, the following query can be formulated:

```
SELECT NAME, PRICE
FROM PRODUCT
WHERE PRICE < 50
```

Returning as result the table in Figure 1.17.

A more complex example is the query that retrieves the names of all the customers who have purchased products costing less than $50, together with the name of the purchased products. To fulfill this query, it is necessary to consider three tables: the PRODUCT, CUSTOMER, and PURCHASE tables. In the relational terminology, the PRODUCT table must be "joined" to the PURCHASE table to obtain a table that includes for every product the IDs of customers who have purchased

NAME	PRICE
Aladdin	35
Casper	40
Dundee	21

Figure 1.17 Result of a SQL query.

PRODUCT.NAME	CUSTOMER.NAME
Aladdin	Acme Inc.
Casper	Acme Inc.
Dundee	Supply Inc.

Figure 1.18 Result of a SQL query.

it, and then this intermediate table must be in turn joined to the CUSTOMER table, to add the information about the customer's name and address. The following query does the job:

```
SELECT PRODUCT.NAME, CUSTOMER.NAME
FROM PRODUCT,PURCHASE, CUSTOMER
WHERE PRODUCT.CODE = PURCHASE.PROD-CODE AND
      CUSTOMER.ID = PURCHASE.CUST-ID AND
      PRODUCT.PRICE < 50
```

The WHERE condition includes both "join conditions" (PRODUCT.CODE = PURCHASE.PROD-CODE and CUSTOMER.ID = PURCHASE.CUST-ID) and row selection condition (PRODUCT.PRICE < 50). The result of the query applied to the three tables shown in Figure 1.15 and Figure 1.16 is the table shown in Figure 1.18.

The SQL SELECT statement has more options than table joins and row selection; it includes primitives for applying aggregate functions to sets of rows, sorting and grouping query results, and building complex nested queries.

Figure 1.19 shows the various ways in which SQL queries can be submitted to the database. As a first option SQL queries can be posed directly by the user, using a dedicated graphical user interface. However, such an interface is rarely used, and queries are instead normally embedded inside programs, which interact with the database to extract or update its content according to the business logic of the application. The interaction between an external program and the database query processor takes place through the *Application Programming Interface* (API) of the database, which offers procedures for shipping queries to the database and for getting results back.

To ensure the portability of programs across databases of different vendors, which may have different APIs, special-purpose libraries are available that offer a standard set of functions masking the peculiarities of the specific database product. The most diffused interoperability libraries are the *Open Database Connectivity* (ODBC) library by Microsoft, and the *Java Database Connectivity* (JDBC) library,

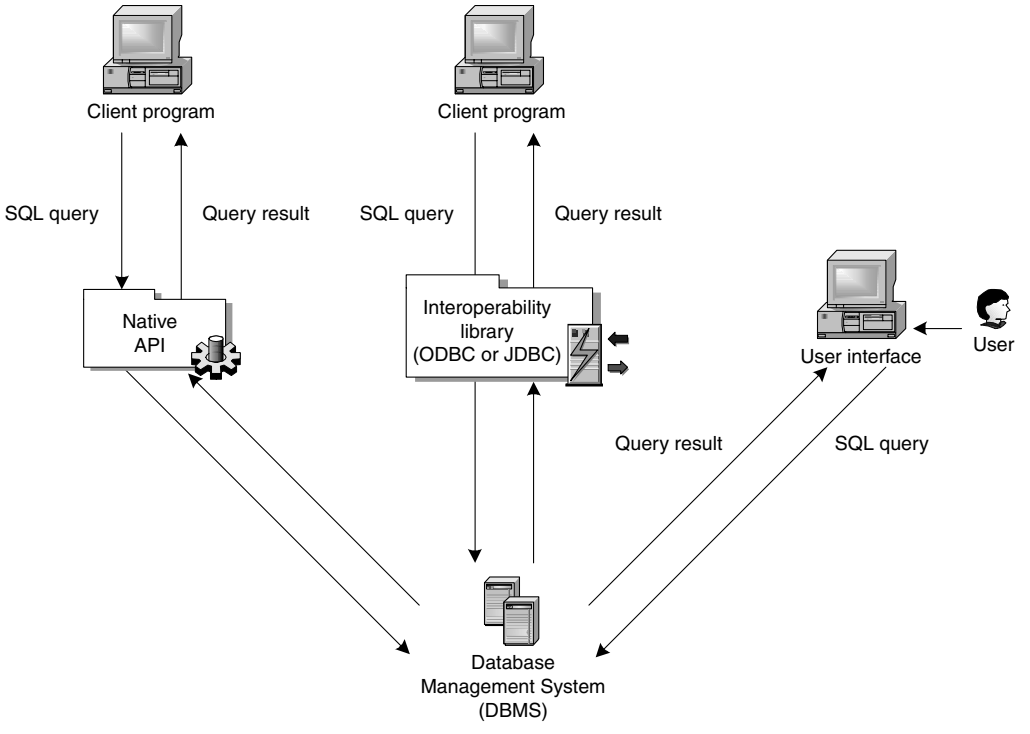

Figure 1.19 Interactions between clients and database management system.

by Javasoft, which is part of the Java 2 Enterprise Edition platform. These libraries expose a set of utility objects (for example, the "database connection" object, the "statement" object, the "result set" object), which hide the details of database interaction and facilitate the programming of database-aware applications. Programs that build dynamic Web pages from database content typically use the ODBC or JDBC libraries.

1.5 Beyond HTTP: Building Web Pages on the Fly

Real-life Web applications require the capability of serving to the users HTML pages that dynamically publish content coming from one or more data sources. For example, the content of the home page of a news magazine is refreshed daily, by extracting the latest news from the news repository. This requirement goes beyond the capabilities of the HTTP protocol, which is designed to ship requests for

resources from the browser to the server, and not to govern the process by which the desired resource is located or built. This section reviews a progression of techniques for building Web pages on the fly, extending the capability of HTTP.

1.5.1 Common Gateway Interface

The simplest way to solve the problem of dynamically building a Web page in response to an HTTP request is to let the HTTP server delegate the construction of the page to an external program, using a standard interface called *Common Gateway Interface* (CGI), as illustrated in Figure 1.20.

The dynamic computation of the page introduces a "double" client server loop (Figure 1.20):

1. The browser issues an HTTP request to the HTTP server.
2. The HTTP server invokes an external program that constructs the page to be sent back to the user.
3. The external program sends the constructed page to the HTTP server.
4. The HTTP server uses the page constructed by the external program to assemble the HTTP response and sends this response back to the browser.

The invocation of an external program occurs when the HTTP request coming from the browser includes an URL pointing to an executable program called *CGI script* instead of pointing to a document or media file. The Web server must be properly configured to be able to distinguish URLs that point to resources that must be "executed" from URLs pointing to static resources to be served back as they are.

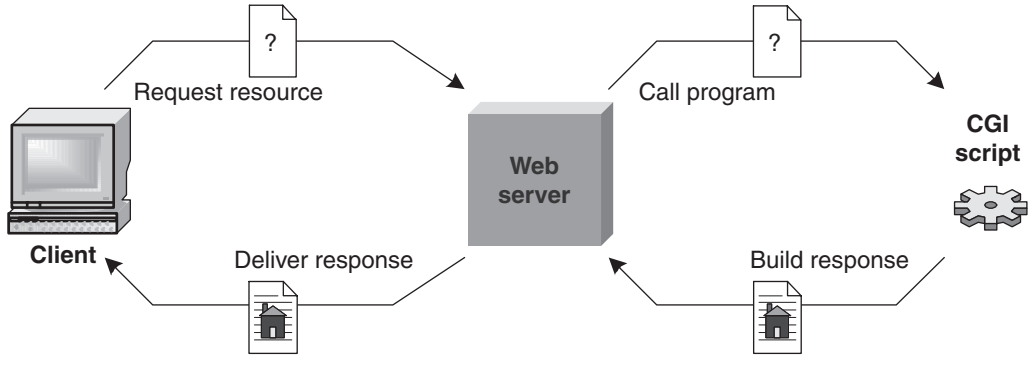

Figure 1.20 Dynamic construction of a Web page in response to an HTTP request.

When the HTTP request refers to an executable, the Web server launches it, typically by spawning a new process. The communication between the Web server and the CGI script is based upon a set of predefined system variables, which are initialized by the HTTP server upon receipt of the HTTP request. These variables describe the parameters of the HTTP request (such as the HTTP method used, the number of bytes attached to the request, and so on).

When the CGI script is launched, it reads the CGI variables and the request body (if there is one transmitted via the POST method), then it performs the elaboration required to compute the response page, and finally it outputs the page.

For an effective interaction, the user must be able to supply request parameters to the Web server, to be forwarded to the CGI script and used to elaborate the response. For example, the user must be able to supply the name of an artist to get a page listing all the music albums composed by him/her. User input coming from the browser must be transported by the HTTP request, which is the only means of interaction between the browser and the Web server. Transportation takes place by means of the two distinct HTTP methods already discussed in Section 1.2.1:

- With the *GET method,* the user input is appended to the requested URL. Parameters are encoded as label/value pairs appended to the URL, after a question mark symbol, as in the following example: *http://www.google.com/search?q=SQL&hl=it* When the GET method is used, the Web server initializes a CGI variable called "query string" with the value of the final part of the URL containing the label/value pairs, so that the CGI script can get the user input simply by fetching and decoding the content of this variable. Due to the limitations in the length of a URL (256 characters at maximum), the GET method is used when the client needs to communicate to the server only a few simple parameters.

- With the POST method, the user input is attached to the HTTP request using the message body. Therefore, the POST method supports the transmission of large amounts of data from the browser to the Web server. In order to collect user input in the browser, HTML includes the `<FORM>` element, which permits the designer to construct an HTML page to be used as a data entry mask. When the user clicks the submit button in the form, the browser packages the user's input in the body of the HTTP request and sends it to the Web server. Upon receipt of a POST request, the Web server extracts the user's input from the message body, and sends it to the standard input of the CGI script, which can use it for performing a business action and elaborating the response.

Figure 1.21 shows a simple example of HTML form for inputting the name of an artist and the desired recording medium. Syntactically, an HTML form is de-

```
<HTML>
<HEAD>
<TITLE>A simple HTML form</TITLE>
</HEAD>
<BODY>
<P align="center">An example of HTML form: please input the name of the
artist and your preferred recording medium</P>
<FORM action="html_form_action.exe" method="post">
  Artist's name: <INPUT type="text" name="name"> <BR>
  <INPUT type="radio" name="medium" value="CD" checked> CD
  <INPUT type="radio" name="medium" value="Vinyl"> Vinyl <BR>
  <INPUT type="submit" value="Submit">
</FORM>
  </BODY>
  </HTML>
```

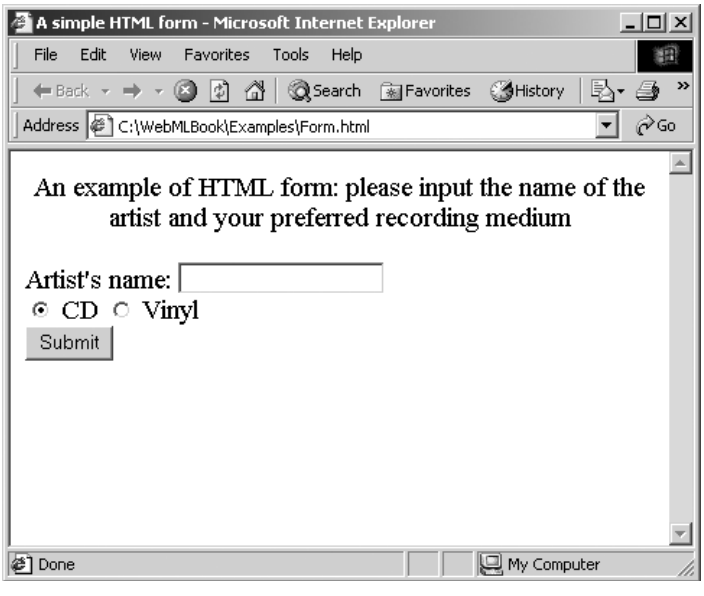

Figure 1.21 An example of an HTML form.

limited by the <FORM> element, which includes multiple <INPUT> elements for collecting input. The form of Figure 1.21 includes four inputs: a text input, for inserting a string; two radio buttons, for inputting the value of the recording

medium; and a button, for confirming the data entry and submitting it to the Web server. The enclosing <FORM> element has two attributes:

- The `action` attribute specifies the URL of the server program to be called when the submit button is pressed.
- The `method` attribute specifies the HTTP method to use for sending the input.

A CGI program can exploit the user's input to assemble a `SELECT` query for retrieving data to be placed inside an HTML page, as well as to store the user's input into the database, issuing an `INSERT` or `UPDATE` query.

CGI is the simplest way of dynamically publishing or updating content on the Web, but its architecture has severe limitations, which make it unpractical in most situations:

- At each HTTP request for a CGI script the Web server spawns a new process, which is terminated at the end of execution. Process creation and termination is a very costly activity, which may quickly become the performance bottleneck.
- Terminating the process where the CGI script is executed after each request prevents information about the user interaction to be retained between consecutive user requests, unless such information is stored in a database, which again impacts performance. In addition, terminating the process also prevents the management of shared resources, like a database or network connection, used by multiple users across multiple HTTP requests.

For these reasons, more complex architectures have been developed to overcome the downsides of CGI and cope with applications that demand a high level of performance and the retention of user session data.

1.5.2 Web Server Extensions

The limitations of the CGI architecture, which are mostly due to the forced termination of the process where the CGI script is executed, can be eliminated by extending the capabilities of the Web server. The adopted solution is shown in Figure 1.22: the Web server is extended with an application execution engine, where the programs for computing the HTTP response can be processed in an efficient way.

The application execution engine can be implemented in several different ways, from a dynamically linked library directly integrated inside the Web server,

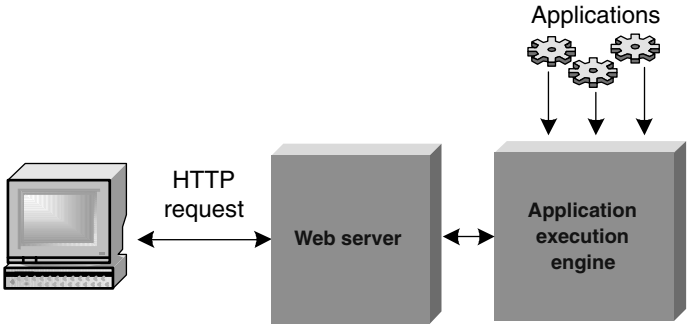

Figure 1.22 The extended Web server architecture.

to a distinct process running on the same or on a separate machine. Regardless of the technical details of its implementation, a Web server extension offers an efficient and persistent execution environment, where applications can be installed and executed in response to incoming HTTP requests, without being terminated after each request.

The extended Web server architecture offers several advantages with respect to the pure CGI approach:

- It eliminates the process creation and termination overhead.
- It permits the allocation of shared resources, associated with one or more applications and concurrently accessed by multiple users.
- It offers a main memory environment for storing session data, durable across multiple HTTP requests.

An example of extended Web server architecture is the Javasoft's Servlet API (illustrated in Figure 1.23), which associates the Web server with a *Java Virtual Machine* (JVM). The JVM supports the execution of a special Java program, called a *servlet container,* which in turn manages the execution of Java servlets and the maintenance of session data. A *Java servlet* is the counterpart in the Java world of a CGI program; it is a Java program that can be invoked to respond to an HTTP request for a dynamic page. The servlet container intermediates between the Web server and the various servlets: it is responsible for receiving the HTTP request from the Web server, creating a user session when needed, invoking the servlet associated to the HTTP request, and transmitting to the servlet the data of the HTTP request, wrapped in the form of a Java object.

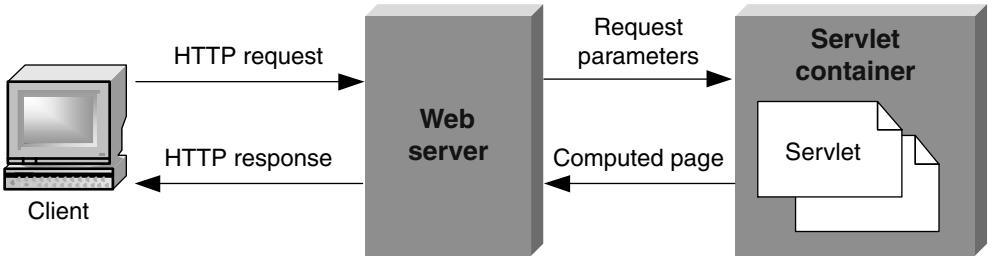

Figure 1.23 Java servlet architecture.

Each servlet is a Java class that extends the standard class `HttpServlet` included in the servlet API specification. The `HttpServlet` class offers the basic primitives for interacting with the servlet container, like the functions for inspecting the HTTP request and the session data, and for writing content to the HTTP response. The example in Figure 1.24 shows the flavor of servlet programming.

The servlet includes the declaration of class `RequestInfo`, which extends `HttpServlet` (line 5) and implements the standard function `doGet` (line 7), which is invoked by the servlet container when an HTTP request formulated with the GET method arrives. The `doGet` function has two input parameters, `request` and `response`, which are the Java objects wrapping the HTTP request and response.

By operating on these two objects, the servlet can inspect and manipulate the HTTP request and response, as shown in line 10, where the servlet sets the MIME type of the response to `text/html`, as necessary when the response is an HTML page. Then, the servlet calls the function `getWriter()` on the response object, which returns an output stream (represented by variable `out`) to write content to (line 11). Everything printed on this output stream ends up in the response sent to the user. After getting the output stream, the servlet starts the actual production of the HTML page, by printing content to the output stream. Lines 12–17 simply print some fixed HTML code and textual content. The lines 18, 20, and 22 extract information from the request object (namely, the request method, the URI[5] of the requested resource, and the version of the browser) and print it to the output stream. The result of invoking the servlet is shown in Figure 1.25 on page 44.

[5]The *Uniform Resource Identifier* (URI) is a generalization of the HTTP concept of Uniform Resource Locator (URL).

```
1:  import java.io.*;
2:  import javax.servlet.*;
3:  import javax.servlet.http.*;
4:
5:  public class RequestInfo extends HttpServlet {
6:
7:    public void doGet(HttpServletRequest request,
8:                      HttpServletResponse response)
9:                      throws IOException, ServletException {
10:     response.setContentType("text/html");
11:     PrintWriter out = response.getWriter();
12:     out.println("<HTML>");
13:     out.println("<HEAD>");
14:     out.println("<TITLE>Request Information Example</TITLE>");
15:     out.println("</HEAD>");
16:     out.println("<BODY>");
17:     out.println("<H3>Request Information Example</H3>");
18:     out.println("Method: " + request.getMethod());
19:     out.println("<BR>");
20:     out.println("Request URI:  " + request.getRequestURI());
21:     out.println("<BR>");
22:     out.println("User Agent:" + request.getHeader("User-Agent"));
23:     out.println("</BODY>");
24:     out.println("</HTML>");
25:   }
26: }
```

Figure 1.24 Example of a Java Servlet.

1.5.3 Implementing Application State over HTTP

Web server extensions, like the Java servlet architecture, offer an efficient way of implementing *stateful* Web applications; that is, HTTP-based applications capable of retaining the state of the user interaction. State information can be stored at the server side, in the form of *session data,* and at the client side, in the form of *cookies.*

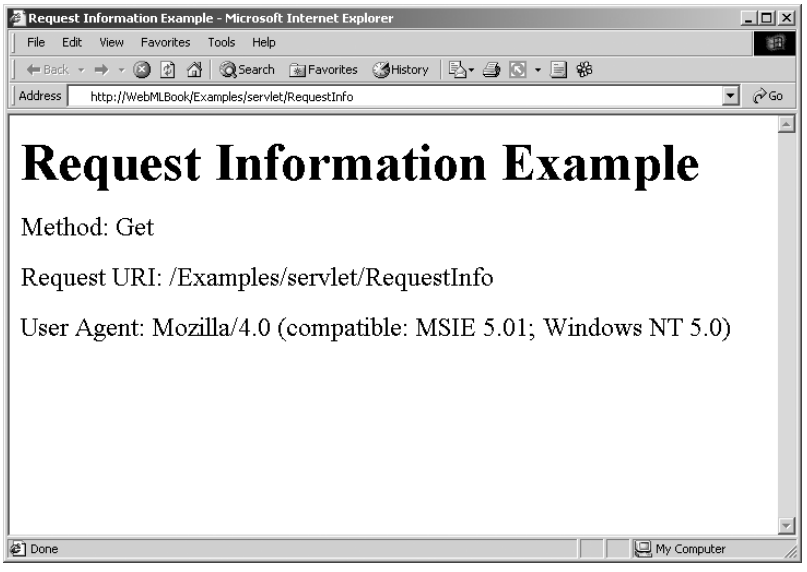

Figure 1.25 The result of executing the servlet program of Figure 1.24.

State maintenance at the server side requires the server to identify and distinguish the HTTP requests of the various clients, to associate each piece of state information to the user to which it belongs. This requirement is fulfilled by creating a *session identifier* upon arrival of the first HTTP request of a new client, and by making the browser communicate such identifier to the server in all the subsequent HTTP requests, so that the server can treat such requests as belonging to the same user's session. For example, in the servlet architecture, the servlet container creates a new session identifier for each HTTP request coming from a client not already associated to a valid session identifier and exploits the mechanisms of cookies and URL rewriting (explained later) to force the browser to communicate the session ID at each HTTP request. The server that originates the session decides when to terminate it, typically by setting a time out after the last user's request, or by offering suitable instructions whereby a server-side program can explicitly invalidate the session. When explicit invalidation or time out occurs, the next request from the client causes the server to create a new session identifier.

When a session is active, the server-side execution engine can associate to it state information. Typically, such information is temporary, and resides in some main-memory data structures, whose duration is the same as that of the user's session. In the servlet environment, session data are wrapped inside a Java object,

called *session,* similar to the request and response objects: the servlet code can access the session object, which offers functions for retrieving and updating information pertaining to a user's session.

State information can also be mantained at the client side, using cookies. A cookie is an object created by a server-side program and stored in the client (typically, in the disk cache of the browser), which can be used by the server-side program to store and retrieve state information associated to the client. A cookie is set when the server includes in the HTTP response sent to the browser a `Set-Cookie` HTTP header, filled with the piece of information to store at the client. The cookie may include a description of the range of URLs for which an associated state information is valid. Any future HTTP requests made by the client that fall in that range of URLs will transmit the content of the cookie back to the server. The duration of the state information associated with a cookie is independent of the duration of a server-side session, and is decided by the client, who may explicitly delete his cookies. A typical usage of cookies in the context of e-commerce application is to store some user preferences that are transparently communicated to the server whenever the user accesses the application.

Cookies can also be used for communicating the session identifier from the server to the client and vice versa, as required for the proper management of session data. However, a disadvantage of cookies is that the user can disable their support in the browser. To overcome this problem, the same data that would be stored in the cookie, for instance, the session identifier, can be preserved for the duration of a session using a technique called *URL rewriting,* which appends the state information as extra parameters in the URLs embedded in the page sent to the client. In this way, the client communicates the needed information to the server even in absence of cookies, but the server-side application code must take care of appending to URLs embedded inside the dynamically generated pages the extra information to be maintained. In the servlet environment, URL rewriting for the communication of the session identifier is facilitated by a utility function, which automatically appends the session identifier to a dynamically produced URL. The following instructions inside a servlet produce an automatically encoded URL:

```
out.println("<A href=");
out.println(response.encodeURL("/appdir/myServlet"));
out.println(">Click here<A>");
```

Executing such instructions produces an HTML fragment similar to the one below:

```
<A href=/appdir/myServlet;jsessionid=XY56443HY3Y>Click here<A>
```

where the URL of the anchor tag contains an extra parameter called `jsessionid`, holding the value of the session identifier. When the user clicks on the anchor, the session identifier is communicated to the server.

1.5.4 Server-Side Scripting

As shown by the example in Figure 1.24, writing a servlet (but also a CGI script) is a technical task that requires programming skills. Analyzing the code of the example shows that programming is applied not only to calculate the dynamic portions of the page (in the example, the various parameters of the HTTP request), but also the fixed textual content and HTML markup. While the former task does require programming skills, the definition of the static content and markup of the result page requires only HTML skills, and could be delegated to the graphic designer.

This observation is at the base of a different linguistic approach to the construction of dynamic Web pages, called *server-side scripting*. Server-side scripting (not to be confused with client-side scripting, treated in Section 1.2.3) is a technique for inserting into an HTML page template some programming instructions that are executed by a server program to calculate the dynamic parts of the page.

With server-side scripting, the technical skills required for writing dynamic pages are reduced, because the programmer may add the necessary scripting instructions to an HTML *page template* prepared by the graphic designer. A page template must then be deployed in a Web server, extended with a script engine capable of interpreting the server-side scripting instructions, as shown in Figure 1.26.

When an incoming HTTP request refers to a page template, the Web server passes the page template to the script engine, which processes the embedded instructions, calculates the dynamic parts of the page, and inserts them in the page template. The output of this processing is a plain HTML page, where all the expressions in the server-side scripting language have been replaced by their

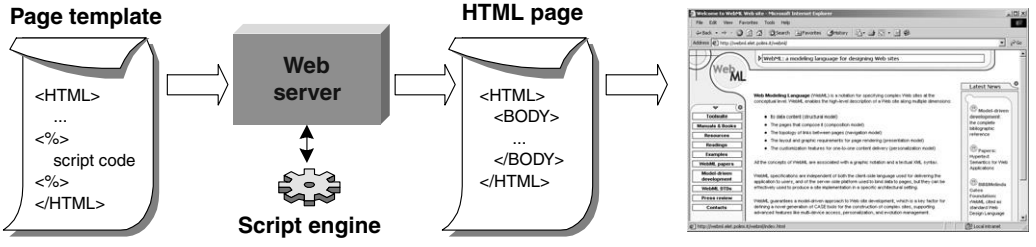

Figure 1.26 The execution of a server-side scripting page template.

calculated values. This page is handed back to the Web server, which forwards it to the client. For the browser receiving the result page, the server-side processing is completely invisible, and the HTML code received is perfectly identical to that of a manually produced, static page.

Various incarnations of the server-side scripting approach are available, among which the *Active Server Pages* (ASP) language by Microsoft, the *Java Server Pages* (JSP) language by Javasoft, and the PHP open language. Figure 1.27 shows the ASP code of a dynamic HTML page template, which displays the same information as the servlet listed in Figure 1.24. The code of Figure 1.27 can be stored in a textual file ending with the `.asp` extension, instead of the `.html` extension, and published under the Microsoft Internet Information Server Web server, for producing an HTML page similar to that shown in Figure 1.25.

The page template contains regular HTML tags and static content, and the elements that must be computed dynamically are coded as ASP statements, syntactically distinguished by special surrounding brackets (`<%..%>`). In the example in Figure 1.27, the ASP statements inside the brackets access the request object to fetch the desired information, and write it in the response. The effect of the `Response.Write` function invocation is to insert a piece of dynamic content in the output page, in the place where the instruction appears. For example, the server-side scripting instruction:

```
<%Response.Write(Request.ServerVariables("request_method"))%>
```

```
<HTML>
<HEAD>
<TITLE>Request Information Example</TITLE>
</HEAD>
<BODY>
<H3>Request Information Example</H3>
Method: <%Response.Write(Request.ServerVariables("request_method"))%>
<BR>
Request URI: <%Response.Write(Request.ServerVariables("URL"))%>
<BR>
User Agent: <%Response.Write(Request.ServerVariables("http_user_agent"))%>
</BODY>
</HTML>
```

Figure 1.27 Example of server-side scripting page template written in Microsoft ASP.

is equivalent to the instruction at line 18 of the servlet listed in Figure 1.24: it prints into the response output stream the value of variable `request_method`, which contains the HTTP method of the user's request.

Despite the similarity of the objective, the coding style of server-side scripting is completely different from the servlet style. A servlet contains programming instructions for printing the entire page, whereas a page template contains regular HTML, and programming instructions are limited to the computation of the variable part of the page. As a consequence, server-side scripting page templates are easier to write and maintain.

1.5.5 Server-Side Executable Tags

Although server-side scripting facilitates the development of dynamic Web applications, it does not eliminate the need of mixing programming with content and markup. The need remains for the programmer and the graphic designer to work jointly on the same source file, which prevents a full "separation of concerns" between the various aspects of Web development: the static content, the look and feel, and the programming logic.

The so-called *server-side tag libraries* take a further step in the direction of separating content and markup from the programming of a dynamic page template. The key idea of the tag library approach is to mask the code necessary for dynamic content production beneath "magic" tags, which can be inserted into the page as regular markup elements, but are executed by a runtime interpreter. With a tag library, the source code of the page template no longer mixes markup, content, and programming. Only content and markup remain, but the markup includes special XML tags that are executed by a server-side program to produce further content and/or HTML markup.

In this way, the roles of the programmer and of the graphic designer are cleanly separated: the programmer conceives a set of XML tags for producing the desired dynamic content, defines the tag properties to expose to the graphic designer, and writes the code for "executing" the tag; the graphic designer specifies the look and feel of the page, defines its static content, and inserts and edits the properties of the "magic" tags created by the programmer.

Tag libraries are available both in the Java world, starting with Version 1.1 of the JSP specification, and in the Microsoft .NET platform, as part of the ASP.NET language.

As an example, Figure 1.28 shows the use of a custom tag in the Microsoft ASP.NET language. In the first line of the HTML source code, an ASP.NET statement registers a custom tag library, named "Acme." Tags of this family can then

```
1:  <%@ Register TagPrefix="Acme" Namespace="Acme" Assembly="Acme" %>
2:  <HTML>
3:    <BODY>
4:      <H1>A user-defined tag for showing a calendar with the current
5:        date at the server highlighted</H1>
6:      <Acme:Calendar id="MyCal" runat=server/>
7:    </BODY>
8:  </HTML>
```

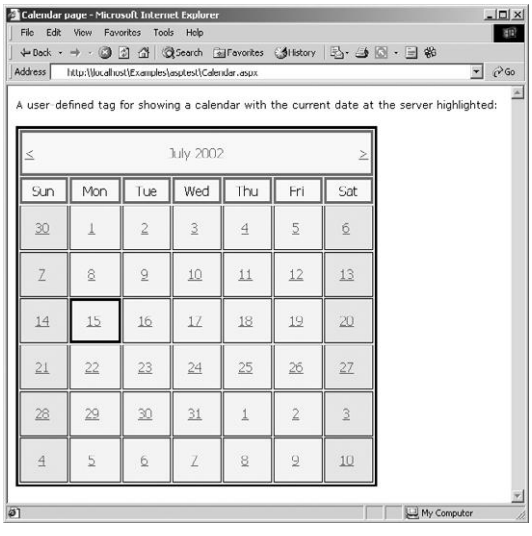

Figure 1.28 Example of server-side executable tag in Microsoft ASP.NET.

be introduced in the page, as XML tags, prefixed with the name of their family. One such tag, named `Calendar`, appears at line 6: it is a complex object, implemented by a server-side program, as declared by the `runat=server` attribute inside the tag. In this example, the behavior associated with the calendar is specified by a separate program, written in the latest version of Visual Basic, called VB.NET, or in C#, a new programming language designed by Microsoft.

The result of executing the ASP.NET template is also shown in Figure 1.28: the ASP.NET execution engine leaves the regular HTML tags intact, so that they are rendered by the browser in the usual way; conversely, the ASP.NET engine processes the `Calendar` tag, and replaces it with the content dynamically produced by the

program. As shown in Figure 1.28, the output associated with such a tag is a table, listing the days of the current month, and permitting the user to scroll to the previous or next month.

A similar example could be recast in the context of the Java 2 platform: in this case, the program associated with the `Calendar` tag would be a Java class.

1.5.6 Increasing Scalability with Application Servers

Large-scale Web applications designed for supporting the electronic business must ensure a high level of availability, security, and scalability, because they can be exposed to millions of concurrent users in the potentially hostile Internet environment. To ensure the required level of service, enterprise Web applications must have a modular architecture, where each component can be easily replicated, to increase performance and avoid single points of failure.

The requirements of scalability and reliability have fostered the commercial success of *application servers,* which are software products complementing the extended Web server architecture described in Figure 1.22. Technically speaking, an application server is a software platform, distinct from the Web server, dedicated to the efficient execution of business components for supporting the construction of dynamic pages.

The typical organization and page computation flow of a Web architecture incorporating an application server are illustrated in Figure 1.29 and Figure 1.30.

The client request (1), formatted in HTTP, is received by the Web server, which transforms it into a request to the scripting engine (2). The scripting engine

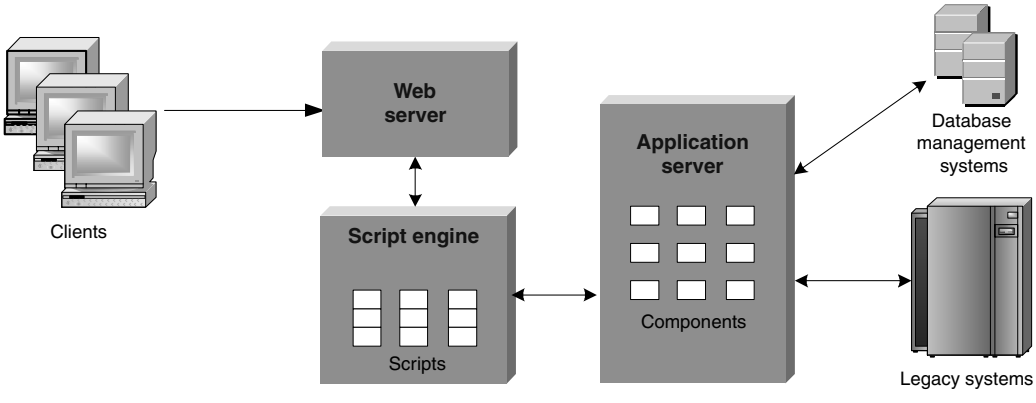

Figure 1.29 Architecture and components of an application server architecture.

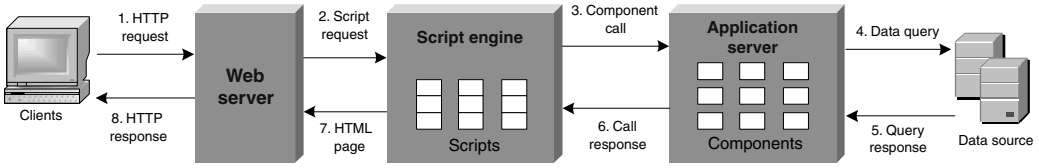

Figure 1.30 The flow of requests and responses in application server architectures.

executes the program associated with the requested URL, which may include calls to business components hosted in the application server (3). Typically, such calls involve the retrieval and elaboration of data from one or more data sources, like corporate databases or legacy systems. The components managed by the application server dispatch the query to the data source (4), collect the query result (5), possibly elaborate it and and hand it back to the scripting engine (6). Query results are integrated into the HTTP response by the scripts executed in the scripting engine, to obtain a result HTML page (7), routed by the HTTP server to the client (8).

The main purpose of the application server is to provide a feature-rich execution environment for the business components, which facilitates the construction of scalable and reliable applications. This execution environment, often called a "managed runtime environment" includes the following services:

- *Transparent component distribution, replication, and load balancing:* The business objects programmed by the user are installed into the managed runtime environment, which may be distributed on multiple processes and physical machines. The application server automatically manages the creation of processes, the replication of business objects and their allocation to the available processes, the allotment of client requests and of the calls issued by the scripting programs to the business objects, and the dynamic adaptation of the number of processes and business objects to the increase and decrease of the actual workload. The application server manages replication and load balancing in such a way that they are totally transparent to the calling client, which can behave as if interacting with a single instance of a business object.
- *Failure recovery:* The application server may monitor the active hosts, processes and business objects, detect hardware, software and network failures, and automatically avert client requests addressed to a failed component and route them to available replicas of the same business object.

- *Transaction management:* The application server may provide the capability of defining units of work (called *transactions*), which are either executed successfully from start to end, or rolled back completely in case of failure of any of the included operations. Transactions are typically offered by database management systems for sequences of database update operations. In application servers, this important service is offered for generic sequences of calls to business objects.
- *Resource pooling:* The application server may handle pools of expensive resources, like database connections, and share these resource among multiple business objects in an optimized way.
- *Interoperability with legacy applications:* The application server may be equipped with predefined gateways or software developments kits for exchanging messages and data with applications developed on obsolete platforms or with surpassed technologies.
- *Multi-protocol, multi-language application development support:* The application server may integrate multiple application distribution protocols and programming languages into a uniform development environment, and facilitate cross-platform application development and migration.

Factoring these services out of the individual applications greatly reduces their development complexity, which results in better software quality and easier maintenance. For this reason, application servers are at present the chief component of large-scale Web applications.

The application server architecture represented in Figure 1.29 has many commercial incarnations, which vary for the quality and quantity of services, and for the supported programming languages and communication protocols. The two most comprehensive solutions are Javasoft's Java 2 Enterprise Edition and Microsoft .NET.

Java 2 Enterprise Edition (J2EE) is a set of extension APIs of the Java 2 programming language, conceived to offer a portable environment for the development of enterprise-class Web applications. The core of the platform is the Enterprise JavaBeans API, which dictates the rules for defining object-oriented components managed in the application server.

The Microsoft .NET architecture is the new application development infrastructure designed by Microsoft that replaces several previous Web-related technologies, including Active Server Pages. The main innovation of the .NET architecture is the introduction of a managed runtime environment: applications written in VB.NET, C++, or C# are partially compiled into a byte code called *Microsoft Intermediate Language* (MSIL), executed by the *Common Language Runtime* (CLR).

The J2EE and .NET environments have many characteristics in common: both are software platforms conceived for multi-tier, object-oriented, distributed applications, running in a managed and feature-rich execution environment. The main difference between the two is not technical: J2EE privileges application portability with respect to operating systems, but restricts the programming language to Java; conversely, Microsoft .NET has a multi-language development environment, but it is limited to the Microsoft operating systems. Although a detailed comparison of the two approaches is outside the scope of this chapter, a parallel overview of the two platforms is given in Table 1.3.

Table 1.3 Overview of J2EE and .NET platforms.

Platform feature	J2EE	.NET
Operating systems	Any	Windows 2000, Windows XP
Browser	Any	Any (Internet Explorer, if ActiveX components are used)
Client-side components	Java applets	ActiveX components
Web server	Any	IIS
Server-side scripts	Servlet and JSP	ASP.NET and Web Forms
Server components	Enterprise Java Beans (EJB)	.NET Managed Components
Programming languages	Java	VB.NET, C++, C#
Communication protocol	Internet Inter Orb Protocol (IIOP)	Simple Object Access Protocol (SOAP)
Database access	JDBC, SQL/J	ADO.NET
Distributed transaction processing	Java Transaction Server (JTS)	Microsoft Distributed Transaction Coordinator (MS-DTC) COM+ Services
Security	Java Security Services	COM+ Security Call Context
Directory services	Java Naming and Directory Interface (JNDI)	Active Directory Services Interface (ADSI)

1.5.7 Three-tier Architectures

The application server architecture illustrated in Figure 1.29 is the "Web version" of a more general software architecture called *three-tier architecture*. The core principle of three-tier architectures is to provide an intermediate layer between the client and the data tier, which centralizes middleware services and the business logic of the application. Three-tier architectures were proposed in alternative to two-tier architectures (Figure 1.31), in which client programs interact directly with database management systems by posing queries, receiving the results, and processing them for presentation to the user.

Three-tier architectures offer a higher degree of scalability than two-tier configurations, thanks to better network utilization and to the virtually unlimited replication and load distribution capabilities of the middle tier.

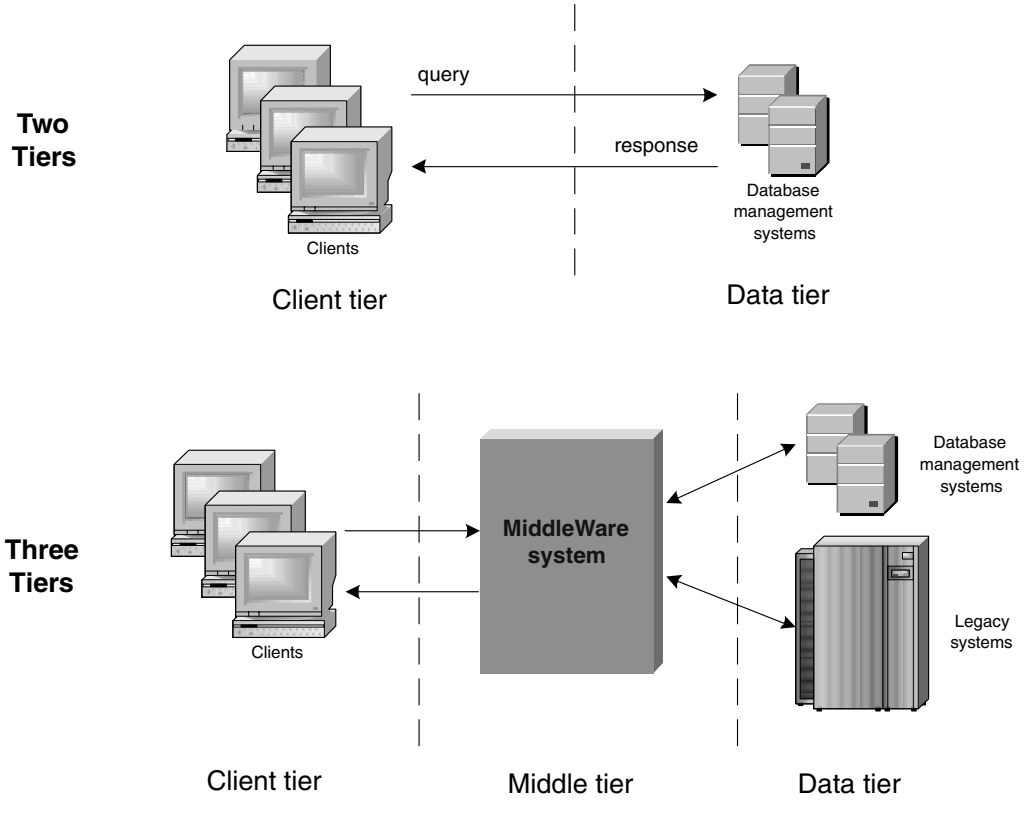

Figure 1.31 Two-tier architecture versus three-tier architecture.

Three-tier architectures existed prior to the advent of Web applications and were supported by a wide range of middleware products, including *Transaction Processing Monitors* (TP Monitors), message-based middleware systems, and distributed object request brokers. Such pre-Web middleware products have evolved into modern Web application servers by incorporating HTTP-specific features, like an HTTP server and a server-side scripting engine. Another distinctive feature of modern Web application servers with respect to their predecessors is the adoption of object-oriented components as the atomic elements of application programming and deployment, and the consequent adoption of object-based distribution protocols, like the Corba 2 Internet Inter-Orb Protocol, or SOAP (Simple Object Access Protocol).

1.5.8 Multi-Device Content Publishing with XML

We conclude this rather long technology overview with a last look at the technical challenges posed by the evolution of requirements to design and implement Web applications.

The advent of mobile terminals and their increasing popularity in the consumer market is shaping the scenario for a new generation of applications, characterised by the need for publishing the same content for a very large spectrum of access devices. A typical instance of this new scenario is mobile commerce, where users are expected to interact with the same application with such devices as their PC, mobile phone, and TV set.

Multi-device applications can be regarded as an extension of "traditional" Web applications, in the sense that content publishing must occur not only for regular Web users equipped with a normal PC screen, but also for other kinds of markup languages and access devices. Therefore, the need arises to reconsider the process of content publishing, to assess its adequacy to the broader scenario of multi-device Web applications.

As Figure 1.30 has shown, content publishing in conventional Web application architectures is characterised by a long flow of client-server request-response loops, in which data get progressively transformed from a native format to the format suitable for presentation to the user. This long chain of transformations is error-prone and difficult to maintain: any change in the requirements (such as changes in the database table format or in the page presentation) involves updating several programs, for example the SQL queries for data extraction and the servlets or JSP page templates for page assembly. The primary reason for this complexity is the mismatch between the format in which content is stored in the database, for example, as a set of relational rows, and the format in which

it must be presented to the user, namely, as an HTML page. The inconvenience of the approach based on page construction programs, like servlets and JSP page templates, increases for multi-device applications; the same content must be rendered in different markup languages or with different presentations to meet the requirements of access devices with disparate rendition capabilities, from wide-screen monitors to the tiny screens of cellular phones. In such a scenario, a different set of programs must be implemented and maintained for each family of output devices with comparable content presentation features.

To alleviate these problems, *XML can be used as an intermediate data format,* exploited by the data extraction queries to encode their result, as illustrated in Figure 1.32. In this case, HTML-aware servlet programs and page templates can be replaced by XSL programs, which transform the XML results of data queries into pages, encoded in the markup language most suitable to the specific access device issuing the request.

The advantage of this alternative approach is granted by the nature of XML, which is a format usable for both structured data and semi-structured documents and thus alleviates the "format gap" between database and Web content. Database content can be transformed from the relational format to XML, using the SQL extensions for encoding query results into XML available in most com-

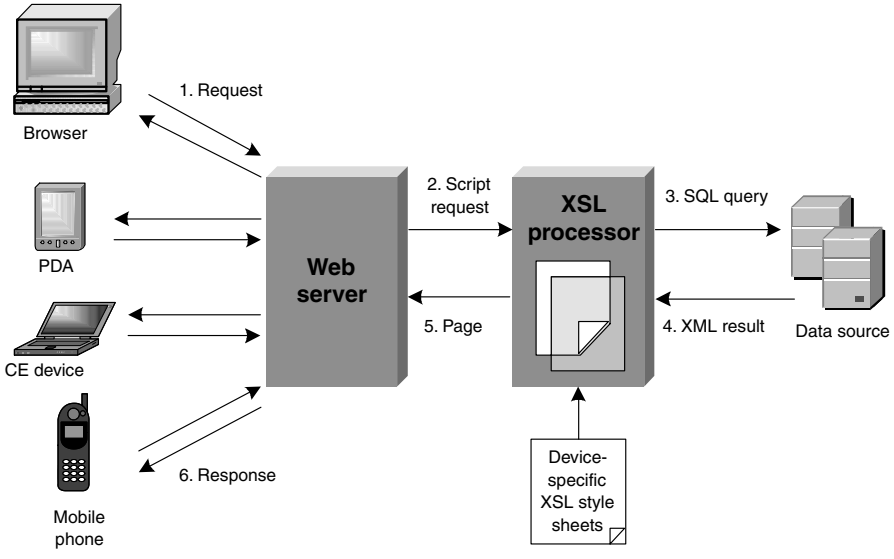

Figure 1.32 XML-based content publishing architecture.

mercial databases. Then, XML content can be converted into the desired markup language, exploiting declarative presentation rules written in XSL. This approach requires less programming and exploits better the power of such declarative languages as SQL and XSL; moreover, application evolution becomes easier, because adding yet another presentation format for a new device requires only the addition of the necessary XSL presentation rules, and not the coding of servlets or JSP templates.

Summary

This chapter has provided an overview of the technologies that constitute the foundations for building data-intensive Web applications. First, we surveyed the basic protocols and languages for the Web, such as HTTP, HTML, and client-side scripting.

Then, XML and the collateral XSL technology were introduced as the new paradigm for content definition and exchange. To cover also the "data" part of data-intensive Web applications, we have included in the chapter a brief compendium on relational databases, SQL, the most popular relational query language, and on the ODBC and JDBC database interoperability standards.

Next, the chapter discussed the architectures for constructing HTML pages on the fly, including CGI, Java servlets, server-side scripting, with a special mention of JSP and Microsoft .NET custom tag libraries, and application servers.

The concluding section addressed the challenging requirements of multi-device application development, and commented on the use of XML as an intermediate data format, and of SQL and XSL as the declarative languages for expressing the queries and transformations necessary to publish database content for multiple devices.

Bibliographic Notes

The bibliography of a survey chapter like the present one could easily become longer than the chapter itself, so many are the subjects touched to give the reader an overview of the technical issues involved. We will restrict our reading suggestions to the fundamental textbooks and publications, and to online resources where the reader may find useful materials and links for his/her personal study of the various matters.

An interesting starting point of the investigation of Web technologies is the specification of HTTP 1.0 [BLFF], which is the act of foundation of the Web. The original concept of HTTP is described by the actual words of its inventors. Other

historical documents on the birth of the Web are available in the Web site of the World Wide Web Consortium (W3C) (*www.w3.org/History.html*) [W3Cf], for example, the source code of the first browser and Web server!

The Web site of the W3C contains a wealth of resources on Web standards [W3Ca, W3Cb, W3Cc, W3Cd, W3Ce] and should be continuously monitored by the reader interested in the technology evolution. The Consortium also organizes a yearly conference (called the WWW Conference), where the research and industrial communities meet to discuss the future of the Web.

HTML [BLC, Rugget] has been the subject of a huge number of textbooks and documents. A step-by-step tutorial on HTML is available on the Web site of the W3 Schools [W3S]. The site features tutorials on many other Web-related technologies, including CSS, JavaScript, Visual Basic Script, ASP, XML, XSL, and more. A complete and easy-to-use online reference manual of all HTML 4 tags can be found at [WDG].

XML is having an editorial success similar to that of HTML, and thus XML textbooks abound. We mention the books [Laurent01] and [Harold01], which offer an extended coverage of XML and of all the most important related standards, including XSL.

The fundamentals of database management systems and of the relational model are covered by many textbooks, among which [ACPT99], which covers both the classical aspects of database technology, and the more recent issues of Web-database integration. An in-depth treatment of the SQL query language can be found in [HV00], which guides the reader in the progressive familiarization with the subtleties of SQL programming.

For developers needing materials on dynamic page generation with Java servlets and server-side scripting with JSP, Sun's Web site offers the official reference guides and technical documentation [Suna, Sunb].

The reference source for the Java 2 Enterprise Edition platform is the Web site by Sun [Sun], which contains the platform specifications and a number of tutorials and developer's guides on the different extension APIs. An in-depth introduction to the development of Web application in the J2EE platform is contained in [Kassem01].

The official reference for application development in the Microsoft .NET architecture is the section of the Microsoft Developer Network devoted to this architecture, reachable from the home page at *http://msdn.microsoft.com*. Various books have appeared on the subject since Microsoft's first announcement of the .NET platform; among the available titles, [MS01] provides an effective tutorial on the integration of .NET server-side controls and databases. An online source of materials on everything related to the .NET world is [ASPNG], which contains tutorials, articles, and links on all aspects of .NET development.