

Masaryk University
Faculty of Informatics

Data-Centric Web Application Framework

Jan Pazdziora

Dissertation

Brno, September 2003



Data-Centric Web Application Framework

Jan Pazdziora

Dissertation

Advisor: Prof. Dr Jiří Zlatuška, CSc.

Brno, September 2003



Produced from DocBook XML source using The SAXON XSLT Processor from Michael Kay and XEP – an XSL Engine for PDF developed by RenderX, Inc. Printed using TrueType fonts Georgia by Microsoft Corporation and Luxi Mono by Bigelow & Holmes, Inc. and URW++.

S díky

Těm, co mne nepřestávali povzbuzovat. Netřeba jmenovat, vy to o sobě víte.



Table of Contents

Abstract	vii
Introduction	viii
I. The Motivation: Area and Problem Statement	1
1. Growing depth of Web technology deployment	2
2. Text-based interfaces promote quick solutions	5
3. Multiplicity of syntax and markup	11
3.1. Server side includes	11
3.2. CGI scripts and programs	13
3.3. Templating techniques	15
3.4. Access to database backend	16
4. Lack of clear internal interfaces	19
4.1. Interfaces in current Web applications	19
4.2. Communication among team members	23
5. The goal of this work	25
II. The Background: Characteristics of Web-Based Applications	26
6. Network-based applications	27
6.1. Applications and data on client	27
6.2. Applications on client and data on server	28
6.3. Both applications and data on server	29
6.4. Conclusion of classification	30
7. Request and response nature of HTTP	32
8. Centralized server part	38
8.1. Central and professional maintenance	38
8.2. Fast development cycles	39
8.3. Instant upgrades	39
8.4. Scalability using external resources	40
8.5. Performance	41
9. Client side highly distributed	43
9.1. Uncontrolled client side	43
9.2. Increased feedback potential	44
9.3. Cost efficiency	44
9.4. Web in all information-handling environments	45
10. Use of extreme programming	46
10.1. Specification of the methodology	46
10.2. Cost of change	48
10.3. Extreme programming and Web projects	49
III. The Solution: Intermediary Data Layer	52
11. Application data in native data structures	53
12. Data matched against formal structure	58

12.1. Data serialization	58
12.2. Overview of the data structure description format	60
12.3. Examples of data structure description document	61
12.4. Operation of data-centric applications	71
12.5. Design notes	71
13. Data structure description reference	74
13.1. Processing steps	74
13.2. Element names	74
13.3. Input parameters	74
13.4. Including fragments of data structure description	75
13.5. Data placeholders	76
13.6. Regular elements	77
14. Output postprocessing	79
14.1. The presentation postprocessing	79
14.2. Alternate HTTP response	81
14.3. Operation in CGI mode	82
15. Implementation	86
15.1. RayApp reference implementation	86
15.2. Performance of the implementation	88
16. Structure description as a communication tool	90
16.1. Analysis	90
16.2. Development and tests of application code	91
16.3. Development and tests of presentation layer	92
16.4. Structure of the Web system	94
16.5. Handling changes	95
16.6. Audit and logging	95
17. Multilingual Web systems	96
17.1. Localized database data sources	97
17.2. Application-transparent setting	101
17.3. Localization of other database objects	102
17.4. Localization of time and numeric values	103
17.5. Preprocessed stylesheets for static texts	103
17.6. Multiple content data sources	106
18. Migration and implementation rules	108
18.1. Separation of tasks	108
18.2. Migration of existing applications	109
19. Conclusion and further work	112
References	114
Curriculum Vitæ	119

List of Figures

2.1. Server validates documents but sends even those not well-defined.	10
4.1. Structure of Web application based on templates.	19
4.2. Interfaces in Web systems.	20
7.1. Requests and responses in user's work with Web-based system.	35
11.1. Core of a Web application.	54
11.2. Web application with native data output.	57
12.1. DSD-based Web application.	71
17.1. Database table courses.	98
17.2. Localized database views.	99



List of Tables

3.1. Tags and escaping in HTML.	12
3.2. String quoting and escaping in Perl.	13
6.1. Categories of network-based applications.	31
8.1. Numbers of applications to Masaryk University.	41
13.1. DSD: Reserved element names.	74
13.2. DSD: Attributes of parameter specification.	74
13.3. DSD: Parameter types.	75
13.4. DSD: URIs allowed in typeref.	75
13.5. DSD: Attributes of data placeholders.	76
13.6. DSD: Built-in data types.	77
13.7. DSD: Values of attribute multiple.	77
13.8. DSD: Attributes for application localization.	77
13.9. DSD: Conditional attributes for regular elements.	78
13.10. DSD: Attributes for attribute management.	78
15.1. RayApp performance.	89
15.2. Apache HTTP server performance.	89




List of Examples

2.1. A snippet of HTTP communication, using telnet as client.	5
2.2. A simple CGI script.	6
2.3. A simple HTML page.	6
2.4. Simple page in valid XHTML.	7
3.1. Server side includes.	11
3.2. Part of email message.	12
3.3. Perl code printing HTML markup.	13
3.4. Using function calls to generate markup.	14
3.5. PHP script producing HTML.	15
3.6. PHP script printing data without escaping.	16
3.7. Mixture of syntax in PHP script.	16
3.8. PHP script running SQL queries without escaping.	17
3.9. Perl script using bind parameters.	18
3.10. SQL queries can still be created dynamically.	18
4.1. Representing list of students in HTML.	21
4.2. Script doing updates and producing HTML.	22
4.3. Script calling module.	22
7.1. HTTP request with GET method.	33
7.2. Simple mod_perl handler.	34
12.1. Data structure description with a single scalar placeholder.	61
12.2. Output XML for various return data structures.	62
12.3. Data structure description with only root element.	63
12.4. Output XML for single root placeholder.	63
12.5. Data structure description for a search application.	64
12.6. Returned list of people for the search application.	65
12.7. Output XML for list of people.	66
12.8. Associative array is equivalent to regular array.	67
12.9. Single person result for the search application.	68
12.10. Output XML for a single-person result.	68
12.11. Associative array naming with values.	69
12.12. Serializing associative array.	70
14.1. Perl module for data-centric processing in external process.	83
14.2. Perl module for server-side postprocessing of CGI application.	85
15.1. RayApp Synopsis.	86
17.1. Code handling multiple languages with multiple branches.	96

Abstract


Existing styles of the development of Web-based systems use procedural languages or template-based techniques where the output is being made up from textual pieces of the target markup. These approaches not only lead to non-valid HTML pages easily, moreover, output markup in the code of individual applications prevents reusability of the code and makes maintenance and further development more complicated.



Therefore, in large systems, true separation of the content and the presentation part is needed. Data structure description format is proposed as a core tool for serialization of application output in native data structures to intermediary data layer in XML, validation of input parameters and output data, development of test suites, and communication among the members of the development team. Data output becomes independent on the intended presentation and can be postprocessed into any target format, while applications no longer contain mixtures of syntaxes that distract developers and make the source code error-prone. Applications become multilingual, as no language-specific code needs to be present and localization of both data sources and presentation stylesheets is moved from run-time to off-line definition. With increasing numbers of user agents, the presentation postprocessing of the data can be shifted to the client side, saving processing power of the server and increasing throughput. Above all, long-term maintainability of Web applications is achieved by stripping them of volatile dependency on the presentation markups.

Keywords: Web system, application code, data structure description, intermediary data layer, server-side processing

Introduction



Suppose that Alice wants some piece of information from Bob. She walks into his office and asks him. Alternatively, if she is not right in the same building, she may pick up a phone and call. Bob recognizes Alice's face or voice and tells her whatever she was inquiring about because he knows Alice is entitled to know. Maybe the information is public and he does not even have to verify Alice's identity — he is ready to tell the answer to anybody who would care to ask. Examples may include an employee asking the personnel department how many days off she has left in the current year, or the head of department verifying her employees' bonuses. Typical queries about public information are students asking registrar office about a schedule of a course or exam at a university, or a customer interested in price of a certain product that Bob's company might be selling.


Chances are, Bob does not hold all the knowledge in his head. He may use a computer to store and manage the information. So whenever somebody walks in or calls him on the phone to ask, he uses his computer to find the pieces of information needed, and then reads the answer from the screen of his monitor. Alternatively, Bob might not know the answer at all, nevertheless he may redirect the question to someone else who holds the necessary information in his or her head, or computer. The questions that Bob answers may also come in paper letters or be faxed in, or may arrive into Bob's email folder. Then, Bob has to write the answers down.

In the described situation, Bob serves only as an *intermediary* in the process of shifting the information around for many questions and answers, often adding no extra value. However, what he may add are mistakes and typos when reading and reciting the answers or writing them down because people are notoriously known not to be very successful in doing routine jobs, like reading and copying numbers or long texts. Less successful than computer. Moreover, it is either necessary to catch Bob in his office at the time he is there, or wait until he finds time to write the answer and send it back. If there are many people who need information stored somewhere in Bob's head, office or computer, Bob becomes an *obstacle* rather than help in getting the answers. Everyone depends on a human with limited availability and throughput.

In order that Bob gives correct answers, he has to keep his knowledge up-to-date and keep records of any changes that happen in the real life. Suppose Bob has schedules of courses at the university in his notepad or in his computer. Whenever a change occurs because two teachers swapped lecture rooms or new course or presentation of visiting professor got scheduled, Bob has to be notified, so that next time he is asked, he provides correct information. Thus, people pour into Bob's office or send in papers, to let him know about changes that have happened.

Since the information that Bob provides to Alice and other people is probably stored in his computer in some way, Alice and others may want to be able to access the information directly,

to learn what they may want to know without bothering Bob. Or without being bothered with Bob. Bob in turn would have more time to provide specialized advice and consultations, or explain the data in nontypical cases.



In order to access information stored in remote machine, computer networks are used as core infrastructure that allows computers to communicate, effectively allowing users of those computers to drive them to fetch or change data on other machines. Preferred models of storing and retrieving information using computers have shifted during the course of time, as the technology has evolved and relative strengths of computing power and storage versus networking capabilities (all with relation to price) have changed. From centralized computer systems with no direct remote access, to terminal accessible computers, and towards highly crumbled computer systems in the form of desktop PCs (personal computers). And back, via local area networks (LAN) and their occasional interconnections towards their total integration [FitzGerald01]. Nowadays, a computer network is merely the synonym for **the Internet**, using Transmission Control Protocol/Internet Protocol (TCP/IP, [Postel81, Comer95]) for connection oriented data transfers, with a set of additional standards for controlling the network and for specialized tasks, for example Domain Name Service (DNS, [Mockapetris87]) for name resolution. From the technical point of view, any computer in the world can be connected to the network, and there is the only global network, the Internet. Computer network infrastructure to reach any computer and process information stored on it is in place.

Being able to transport pieces of information across computer networks using wires or waves is only the beginning of their reasonable utilization. It is more important whether the computers are able to conduct a communication which would empower a human (or software program) at one end with reasonable access to information stored and managed on the remote system. In order to achieve this task of reaching and utilizing the remote pieces of information, sets of communication protocols, standards and formats were defined and became de facto standards all around the globe. The most viable since 1990s is HyperText Transfer Protocol (HTTP, [HTTP]), which together with HyperText Markup Language (HTML, [HTML]) formed the World Wide Web (WWW, or simply the Web, [Berners96]) as a user-friendly layer on top of the raw network.

Unified global protocols of the Web enable to anybody who has a computer connected to some sort of network providing access to the Internet to fetch the news from CNN's Web, or more specifically use software that utilizes open standards to retrieve and display document addressed by `http://www.cnn.com/` URL (Uniform Resource Locator, [Fielding94]). This is not to say that all computers are connected or that all people have access to the Internet. No, the world is not that far yet [Global02]. However, the point is that *if* somebody has a computer or is just getting one, it is reasonable to assume that such a device will be able to provide its users with access to information sources on the Internet, and notably on the Web. That includes hardware

support for network communication, the ability to resolve IP address of requested target and to make connection to correct port on the remote system, and a user agent which can conduct on-line HTTP communication necessary to retrieve the headlines from the news agency Web server and present them to the user. Rendering hypertext data together with images that may be attached to the document or reading context of the HTML page, should the aural output be preferred, are well supported by browser software on the vast majority of computing platforms. Physical implementation of the last mile may differ, various proxy and cache engines may come into play, but the Web content gets delivered and consumed seamlessly and in transparent fashion.

Having common, well recognized and supported protocols and standards for some types of computer interactions represents a technical achievement. For end users around the world, computers connected to the Internet bring the ability to access and work with the exact *same primary information sources* as millions of others, as everyone else who is on-line. This dramatically changes the way and speed at which information and knowledge is distributed. It redefines the role of distribution channels and brings potential for change in all human activities that include or are based on exchanging information. Individuals, businesses, organizations and states may or have to change expectations about their clients', customers', partners', employees' and citizens' ability to conduct communication, run business operations or take roles in all processes. And they have to act accordingly.

The next step in utilizing computer networks is a move from information distribution to data manipulation in remote systems. Not only can people find information via network, but also do actions in a *self-service* manner. Those actions are done in real time, directly, without any human intermediaries in between. On the platform of WWW, Web-based dynamic applications are being developed to cover processes and actions traditionally done by dedicated office staff. Lately, even an *internal operation* of many organizations is being moved to the Web, and the *number of applications* and types of tasks they need to handle *grows* — they have to provide both the user-friendly front end and stable and secure backend, for actions done daily by thousands of users and for rare specialized operations, for all actions that used to be done on paper by hand. For example, it is increasingly important to have not only a simple interface for customers that want to do a simple select-order-pay round, but for all the backend tasks that the enterprise has to do to actually create and deliver the service or goods, together with running the enterprise itself. E-business only starts with the end customer interaction, however, changes in customers' expectations and demands drives the computerized, Web-based systems deep into the structure of the business. Even for small firms, not-for-profit institutions and public organizations, Web architecture brings several advantages in central maintenance and data handling. Such *internal, administrative systems* that should provide support for all processes

of the organization may comprise hundreds of applications and their maintainability becomes the corner-stone of its long-term viability.

Observations about Web-based information system development and deployment depicted in this work come mainly from the development of Information System of Masaryk University [ISMU02]. This system for study administration has been gradually built since December 1998, supporting Masaryk University's conversion towards credit-based study programs and providing administrative and communication tool for all of its over twenty thousand full-time and nine thousand part-time students, all teachers, administrative staff and management of the university, faculties and departments. Alternative views were also gathered when the system was deployed at Faculty of Humanity Studies of Charles University and at private College of Finance and Administration. These two institutions outsourced the system and amended the in-house experience of the development team [Brandejs01] with a provider-supplier case.

The IS MU project uses open and progressive technologies, both during the development and the production operation. However, as the number of applications in the system reached into hundreds, existing styles of Web application development proved to be suboptimal. The experience with the current frameworks and analysis of their shortcomings (Part I) come from direct daily use of common development approaches. The design of the current dynamic application on the Web is mostly user-interface-driven, even if some approaches try (and mostly fail) for content and presentation separation. Applications only use one output markup language, reducing reusability of the system when different format is called for. In addition, the output HTML markup is spread across the code of individual applications and in supporting modules and the way output pages are generated leads very easily to non-valid documents.

In the Web environment, changes usually come from outside and are caused by external development in the technology. Thus, they cannot be avoided. External change and uncontrolled user bases are among the most prominent aspects of Web-based systems and affect the way dynamic administrative systems can be developed and deployed (Part II). Often, a minor change would be needed in the output format, leading to modification of all applications in the system because the markup is stored in those applications.

Therefore, a move from the presentation, HTML-bound development towards a framework which will be presentation-independent and data-centric is called for (Part III). Data structure description is designed to define the interface between business logic and presentation, bringing formalization to the application design, development, and testing, both in the content and in the presentation part. Using native data structures, the output markup is completely removed from the code of individual applications. Clean interfaces leave enough freedom for programmers and Web designers to perform their art, while bringing structure and validation to otherwise textual Web system. On top of the data-centric model, multiple output formats and multilingual

applications can be created, leaving the application code free of any format- or language-specific branches.

The framework is believed to be production ready, should the development team of the IS MU or of any other project wish to utilize it. This work includes both business-oriented analysis to help project managers discover the new methodology and decide whether it ought to be deployed in the development of their Web systems, and a lower-level technical view of the solution aimed at application programmers, which should make them comfortable with the radically different approach and provide them with implementation roadmap.



Part I

The Motivation: Area and Problem Statement



Chapter 1. Growing depth of Web technology deployment

While the original design goal of the World Wide Web (the Web) technology addressed the need of sharing and linking information to facilitate communication among individuals [Berners96], it has grown to a strong commercial channel and a way through which millions of business operations are conducted every day. Firms have realized the potential of the Web to bring (or take away) customers, other organizations can see its potential as a tool for cost reduction, free speech, and easy information distribution. Thus, even if the sharing capability of the Web is still in place, it also holds a strong position as a technical means of core operation of organizations and enterprises of any size [Tapscott96]. Such usage is more internally-focused and often completely replaces the original internal infrastructure of the firm or institution. Due to the technology achievements, new virtual enterprises appear and the notions of firm and business get redefined [Zwass98]. The whole new e-commerce area is based on *new communication paths* being available, which enables new groups of people and entities to conduct communication and information exchange.

The speed and level to which firms and organizations adopt the Web technology depends on benefits that they see coming from this media. Internet oriented businesses that based their core competence on the new communication channels and customer and user behavior, like eBay, E-Trade, or Amazon.com [CENISSoo], certainly had a higher commitment in the Web from the very beginning of their operation than classical enterprises that only slowly accepted Internet into their plans and carefully weighted each move, often without a clear long-term strategy [Raghunathan97].

The depth of Web applications deployment can be explored from various points of view, focusing on the technical or procedural integration. The *technical aspect* leads to generations [Cope-land97] based on features that can be seen by the external user:

1. Information publication, usually static pages.
2. Queries against database providing on-line read-only access.
3. Direct updates of the data base (like orders or payments) possible.

The primary criterion in this classification is the direction of actions that end users can do in with the data in the system.

From the *operational point of view*, it is more relevant to focus on levels to which the Web interface is linked to the core operation of the organization [Felix97]:

1. Information in static pages, maintained either by hand or generated off-line from internal primary systems.

2. A small number of dynamic services on top of this static data, for example searches, completely separated from the core data sources.
3. Actions and changes can be conducted in the Web system, but the data base is only occasionally synchronized with the primary systems.
4. The Web applications access the same data sources as other systems in the organization, on-line. They support the same processes, often taking over their functions.

Change from the level three to the level four seems to be the most fundamental because only here the Web interface is considered equal to other, more traditional means of data handling and information processing.

The *processes* that are conducted in the organization and their relation to records in computer systems are yet another type of possible classification [Pazdziora01b]:

1. Actions are done on paper or in other material form and passed over for later digitization, which is often done by separate specialized departments.
2. The person who did the actions in real life records them into the information system themselves, and is responsible for this record.
3. Actions are only done in the computer system and they are only considered finished when they are recorded in the database. Any material form (transcript, confirmation) which is produced is based on the primary electronic records.


Again, the shift to the last stage is the essential one because it modifies the notion of processes and actions in the organization. At the same time, it removes inconsistencies between what is the state of affairs in reality and what is recorded in the computer system.

Note that the classifications above are fairly independent — the information system may be fully on-line, but the data is entered based on paper records. And vice versa, all employees may be updating data in the Web-based system and be responsible for it, but the interconnection with the core information systems is only done off-line, in batch jobs. Nevertheless, often a shift to higher levels in one classification is accompanied by reevaluation of the role of the Web-based system as a whole and followed by a shift in other categories as well.

While many large enterprises have already moved their operation to those higher levels of previous classifications [Chang00], small and medium enterprises [NOIE01], educational institutions or state, municipal and public bodies [EC01, ESIS01] often find themselves at the very beginning. Web is still seen as an unnecessary burden by some institutions, so when the external need finally prevails, only *the simplest and fastest solution* is used to cover the immediate pressure. This is quite understandable, big commitments cannot be expected when then technology and its outcomes are new for the organization.

Later, when the institution gets comfortable with the simple Web system, new needs are being realized. They come from two sources:

- As new technology becomes available, more dynamic and colorful features and attractive looks are desired to in order keep the Web presence competitive, or simply modern looking.
- Institution may decide to integrate the Web system more tightly to its core technology or processes. Alternatively, new processes or business operations are created, using the technology.



New features are being added to the live system, but often as an add-on on top of existing framework, without more in-depth restructuralization of the whole system. Technologies in the Web space arrive very quickly and suddenly new feature may seen as necessary, feature that was not available or known a few months ago when the Web application went on-line. While Web systems make gradual extensions possible and easy, new face of the Web site and new functions are usually required fast. In addition, the Web system has to keep running to provide on-line services. As a result and under this external pressures, extensions are added in the easiest way possible, without any formalism that would link the old system with the new one. The Web system grows and responds to the external changes well and quickly.

However, with rapid additions, does the internal architecture of the Web applications stay clean and solid, maintainable in the long run? The Web system may soon consist of hundreds of applications, written in different times by different development teams, supporting many processes throughout the institution. The architectures of current Web-based applications are very simplistic and handle growth by allowing ad-hoc changes using textual interfaces. As a result, Web sites can start small, simple and cheap, with maybe only static presentation, and can be bridged into more complex setup. Nonetheless, the growth is not accompanied by better order or structure.

In the next chapter, text-based interfaces and their impact of the internal of Web-based applications will be explored.

Chapter 2. Text-based interfaces promote quick solutions

The beginning of the design of the World Wide Web dates back to 1989. The design was produced with flexibility and interoperability as the main concept. For example, the Universal Resource Identifier [URI] as the addressing mechanism on the Web was built around the following three design criteria [Berners94]:

- Extensibility;
- Completeness;
- Printability.

While extensibility and completeness is a usual requirement, printability gears towards special goal. Printability means that the naming of resources and documents in the Web is text-based and humanly readable. Thus, the address of a document can and is expected to be written down or spelled out by human, without any machine needed.

Likewise, the HyperText Transfer Protocol (HTTP) [HTTP] and HyperText Markup Language (HTML) [HTML] are textual formats, or at least text-based. In HTTP [Fielding99], the request line and request headers, the status line of the response, as well as the response headers are defined in terms of text lines. Even the separation of the HTTP message headers from the message body is denoted as empty line in the protocol specification. The notion of lines suggests that the information is supposed and planned to be manipulated by text editors and in text-oriented programming paradigms, as opposed to protocols that are defined in terms of fields with binary types and binary content.

Example 2.1. A snippet of HTTP communication, using telnet as client.

```
$ telnet www 80
Trying 147.251.48.1...
Connected to www.
Escape character is '^]'.
GET /~adelton/ HTTP/1.0
Host: www
Accept-Language: cs,en

HTTP/1.1 200 OK
Date: Thu, 10 Jul 2003 13:55:33 GMT
Server: Apache/1.3.27 (Unix) mod_ssl/2.8.12 OpenSSL/0.9.7a
Content-Location: index.html.cs
Vary: negotiate,accept-language,accept-charset
Last-Modified: Mon, 30 Jun 2003 10:27:35 GMT
Accept-Ranges: bytes
Content-Length: 2857
Connection: close
Content-Type: text/html; charset=iso-8859-2
... and many additional lines of HTTP response headers and body follow.
```

Text- and line-oriented nature of the core Web protocol makes it possible to watch and debug the Web operation using generic utilities like telnet, as demonstrated by a transcript of real communication in Example 2.1. Similarly, the simplest Web application in the form of Common Gateway Interface (CGI) [NCSA95] only takes a few lines in typical scripting environments. Example 2.2 shows a Perl script that outputs HTTP header and body, separated by an empty line, denoted by two new-line characters. In order to make the program useful from a Web browser and to achieve some dynamic actions, processing of input parameters would likely need to be added. For the same reason, the content type might be changed to HTML, making use of HTML forms and hyperlinks. Nonetheless, both the input data and the output would still be in textual form.

Example 2.2. A simple CGI script.

```
#!/usr/bin/perl
print "Content-Type: text/plain\n\n";
print "Hello world.\n";
```

The HTML standard is based on SGML (Standard Generalized Markup Language) [Goldfarb91] and again, it is a humanly readable format which can be manipulated by text tools. Certainly more sophisticated editors and approaches can be found, but the simple HTML page like the one in Example 2.3 can be authored by anybody in plain text editor.

Example 2.3. A simple HTML page.

```
<BODY>
<H1>Title</h1>
Select what you like most:
<LI><A HREF="search.cgi?all=1&price=40">All for forty</A>
</BODY>
```

Text-based formats tend to be open, non-proprietary, easy to understand and handle. Data in text formats suggest they can be created and manipulated by humans, not only by computers. Indeed, that was one of the design goals behind HTML. However, humans are well known for not being very good in routine work and composing HTML pages in text editor while trying to keep the syntax correct is certainly routine, calling for typos and omissions. Furthermore, when dealing with textual data, humans work quite a lot with external knowledge and derived context. People can figure out missing pieces to understand the whole document or data set, they can work with partial information. Computers are only expected to do that when specifically told so, not in a case of for example payroll handling.

Unfortunately, the Web browsers have greatly contributed to the fact that incorrect HTML code is used all around the Internet because they are too forgiving to broken syntax, only to help the

end users to get some information through. They allow HTML to be abused for quick and unclean solutions.

Consider the hypothetical HTML page at Example 2.3. We can find the following problems with that HTML source:

- The DOCTYPE is missing from the document.
- Even if DOCTYPE

```
<!DOCTYPE html PUBLIC "-//IETF//DTD HTML 2.0//EN">
```

for the basic version of HTML was assumed, validation would fail with:

- <BODY> not being allowed root element; there should be <HTML> there as the root;
 - element <HEAD> with at least the element <TITLE> missing from <HTML>;
 - element not allowed outside of element marking lists, like ;
 - the ampersand in the hyperlink has to be written as & ; or it will be taken as denoting an entity &price, which is however missing and should be ended by a semicolon.
- For the current XHTML [XHTML] strict type with a DOCTYPE specification

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

additional changes are needed:

- all element names in XHTML are case sensitive and have to be typed in lowercase;
- element that was inserted around the list item to satisfy HTML 2.0 DOCTYPE is not allowed inside of paragraph, and the start tag <p> has to be matched with an end tag anyway;
- the list element requires an end tag as well.


The page, when rewritten in XHTML syntax, gets changed to the source in Example 2.4.

Example 2.4. Simple page in valid XHTML.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head><title></title></head>
<body>
<h1>Title</h1>
<p>Select what you like most:</p>
<ul><li><a href="search.cgi?all=1&amp;price=40">All for forty</a></li></ul>
</body>
</html>
```

However, the change is purely syntactic. For example, even if elements <head> and <title> were added, the title is left empty. Hence, moving to valid XHTML document does not bring any better or new information. Why is it important to produce valid Web pages then?

The reasons are *compatibility* and *interoperability*. Traditionally, Web browsers went to a great length to support Web pages with invalid syntax and workarounded for errors and omissions in HTML pages in order to present the information to the end user. Even modern browsers support so called “quirks” modes that they use to render the document when they find out that the document is syntactically invalid. However, each browser implements its own workarounds and “quirks”, as well as its own extensions. That is why the results for invalid sources are not compatible among various browser platforms and versions.



In the Web environment, source code is not checked for validity before being served to the client over the network. Consequently, no test is performed as far as formal correctness of the document is concerned, something which compilers and preprocessors do in technologies where source is processed before being consumed by end users. Analysis of the source in the Web environment and its processing from the source code is performed on millions of clients, and the result is highly dependent on corrective actions and guesses that those many versions of browser software can make. With certain classes of end devices, like mobile phones and handheld computers with memory and computing power restrictions, only strictly correct documents may be accepted and no corrections are even tried.

Lack of validation and varying level of corrective actions that Web browsers perform has lead many Web developers to create Web systems *based on visual output* provided by their version of Web browser. Often without any deeper knowledge of the technology, copying textual constructs they have seen in HTML sources of other authors, these developers took advantage of low barrier to enter to the Web production. Far too many developers create Web applications by mimicking code and markup they saw elsewhere — with typos, mistakes, often without knowing the real purpose of the code they write. There are thousands and thousands of people who consider themselves experts in HTML, CGI, or ASP (Active Server Pages) coding, yet who never read the standards and never tried to understood the underlying technology.

As the audience on the Web becomes global, expecting everyone to have the same version of a browser from the same vendor becomes less and less viable assumption. That in turn increases efforts and costs that have to be put into building a Web site that displays and operates correctly on everyone's client software. There are two paths that the Web developers take to build Web-based system that will be operating even after month and years:

- Master all details, extensions, and errors of main browsers used today and in the future to produce perfect results for all users.
- Follow open standards and try to produce correct output.

Building Web systems according to the specifications and not according to the visual output of single version of Web client is often seen as less attractive. New features can only be added

for non-essential tasks improving user-friendliness of the Web application, but not for core functionality. In other words, time for addressing specialties and extensions of particular platforms and classes of browsers only comes once the basic Web project is operating and operating correctly. It is also worth noting that validity of the (X)HTML page does not guarantee that every browser will render it the same way. It rather ensures that each browser will start rendering the same source code, without any corrections to the syntax being necessary.

Validity of static HTML documents can be enforced by mandating that each Web page stored in the document area of the Web server has to be validated against appropriate DTD (Document Type Definition) first. For static pages, there is only one document source that has to be verified and this rule could be enforced. However, with non-static Web applications that produce output pages dynamically based on user's input and that often include data retrieved from the backend database, the resulting output is potentially different for each request. Two approaches to validating output of Web application are possible:

1. Validate each output produced by the Web application and only send it to the client if the output is syntactically correct.
2. Make such an arrangement and use such a framework that invalid output is not produced in the first place, or at least the chance of it occurring is vastly minimized.

Validating each output HTML stream and only sending those that are well-formed and well-defined ensures that the user only gets valid documents. However, when no additional support for producing valid documents only is utilized, the cases when applications produce incorrect result, for any reason, will happen and will need to be attended for. If the Web server refuses to serve the incorrect result to the client, the system ends up to be less usable from user perspective than a system without any checks. Users want to finish the work they started and if the transaction like airline ticket reservation is aborted because of some (perhaps small and unimportant) error and there is no way to continue, potential customer will leave and never use the system again. Aborting in case of syntactic errors in pages generated by the Web system is not a way to please users, especially when Web browsers will in many cases ignore minor problems and work around them, so that users do not even notice that internally not everything was correct. Since developers and not the end user caused the validation failure, developers should bear the burden of dealing with the problem.

Moreover, the errors still need to be fixed as soon as possible. Therefore, less intrusive and more viable approach is preferred. Each output page will be validated and instances when the output was not a hundred-percent valid reported to the developers, while still sending the resulting page to the user. The diagram in Figure 2.1 summarizes the flow control in such setup. With this approach, the user is still getting the service and developers can analyze the problems that were revealed by the production operation.

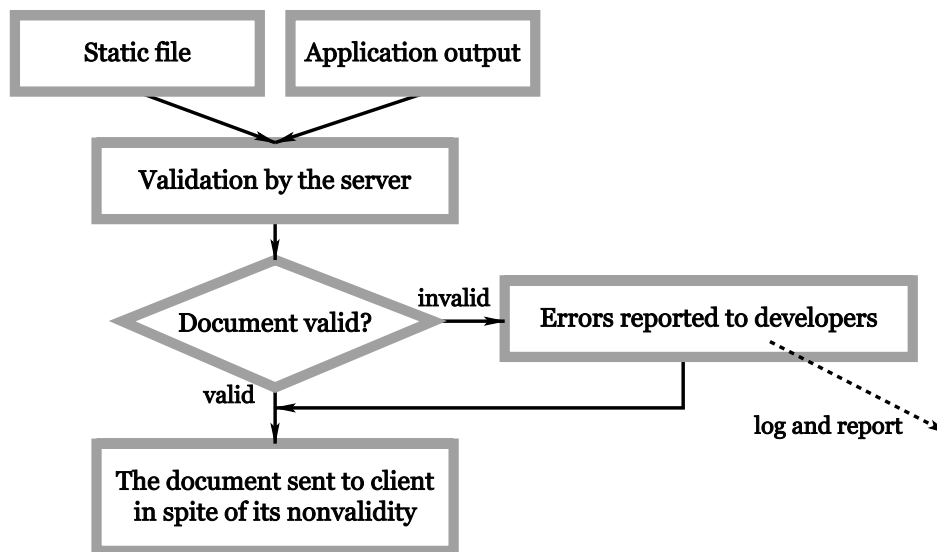


Figure 2.1. Server validates documents but sends even those not well-defined.

While checking every output and reporting errors may work well for small Web-based systems, for medium and large project it is not sufficient. It depends on fixing errors found only by randomly running the applications, not preventing them. In addition, it requires each page to be validated which is a computing-intensive task and may cause performance problems and overall slowdown of the system.

Thus, the second approach to valid documents in dynamic applications is also needed: to ensure that the mistakes do not happen and only valid documents are produced. Ideally, the framework should allow quick and unclean solutions, while supporting and encouraging experienced developers to use more systematic approaches for projects that are to last. To find out whether it is possible with existing frameworks, analysis of styles of programming Web applications currently in use is needed.

Chapter 3. Multiplicity of syntax and markup

Web-based applications that create dynamic content fall into three basic categories:

1. Server side includes (SSI);
2. Scripts and programs producing data together with surrounding markup;
3. Templating techniques.

3.1. Server side includes

Soon after the Web using static pages became more widely used, a need for dynamic content occurred. Server side includes were the first to define special syntax in HTML markup code which is parsed by the Web server.

Utilizing syntax for comments, these pre-set tags are recognized by the Web serving software and are replaced by the dynamic output. The Web browser then receives the result of such substitution. Example 3.1 shows a HTML source with included SSI commands.

Example 3.1. Server side includes.

```
<html>
<body>
<h1>Last mail delivered</h1>
    <p>Current time is <!--#echo var="DATE_LOCAL" -->.</p>
<!--#include virtual="head.html" -->
<pre>
<!--#exec cmd="cat last_mail" -->
</pre>
</body>
</html>
```

The echo directive puts the content of the variable to the resulting page, in this example variable DATE_LOCAL denotes current time. The include directive makes another (internal) HTTP request and places the response body to the output, including all markup in it. The directive exec runs the command specified with the cmd parameter, in this case a command cat that prints content of file last_mail to the standard output, and puts its output to the resulting HTML page. However, the output of the external command is pasted into the HTML page as-is. Consequently, if the output contains any characters or strings that denote markup, they may cause problems, if not intended.

Consider an email message shown in Example 3.2 being stored in the last_mail file. Email addresses enclosed in the less-than and greater-than characters in headers of the message will be copied verbatim to the output and render the resulting HTML code invalid because they

denote start and end of HTML tags. Moreover, depending on the browser version some of them will be considered HTML tags and will not be displayed at all while some will.

Example 3.2. Part of email message.

```
From dbi-announce-return@perl.org Thu Aug  2 20:12 MET 2001
Received: from onion.perl.org (onion.valueclick.com [209.85.157.220])
  by aragorn.ics.muni.cz (8.8.5/8.8.5) with SMTP id UAA00633
  for <adelton@fi.muni.cz>; Thu, 2 Aug 2001 20:12:57 +0200 (MEST)
List-Unsubscribe: <mailto:dbi-announce-unsubscribe@perl.org>
From: Tim Bunce <Tim.Bunce@pobox.com>
To: dbi-announce@perl.org
Subject: NOTE: Incompatible changes in next DBI release
Message-ID: <20010802171443.A14709@rad.ig.co.uk>
Content-Type: text/plain; charset=us-ascii
```

I may not get round to release this version for a few days but I wanted to give some advance notice ASAP.

The original message is much longer and also contains many more headers.

Server side includes work with text strings only, no matter whether they mean data content or markup, in current context. Programmer must be very careful to correctly postprocess any external data and sanitize it. Examples of HTML markup and escaping are listed in Table 3.1.

Table 3.1. Tags and escaping in HTML.

name	desired output	HTML source
less than sign	<	<
greater than sign	>	>
ampersand	&	&
apostrophe	'	'
boldface text	boldface text	boldface text

There are cases when markup from external sources is wanted in the document, like the inclusion of the head.html file using the include virtual SSI directive in Example 3.1. Here the header of the page is supposed to contain everything from the included source, including any HTML markup. However, SSI provides no means of ensuring that the markup brought in by the included file is allowed in the result as mandated by the Document Type Definition of HTML. Thus, included piece of HTML code may break the validity of the result even if both the including master and the included data are well-formed and matching HTML DTD.

Server side includes specification did not get standardized, leading to an incompatibility among Web servers. Application utilizing SSI would need to be checked and ported, when transferred to Web server by another vendor.

3.2. CGI scripts and programs

As a more powerful and portable solution was needed, fully dynamic programs built using Common Gateway Interface [NCSA95] were introduced. Rather than having special markup with dynamic commands inside of static HTML source, they are written in traditional or scripting programming languages like C, Perl, Basic or Java. For communication with the Web server they use environment variables and standard input and output filehandles. To generate HTML pages, output commands (`printf`, `print`, `out.println`) are used.

The output is printed as a collection of static HTML pieces in form of literal text strings and dynamic data coming from variables and results of function or method calls. They however pose the same problem as demonstrated above: text strings that have special markup meaning will appear in output unless treated specially, causing the output to be invalid, with severe consequences when rendered in Web browser. In addition, literal strings have to be treated carefully when they include the same characters as those denoting start or end of these literals, as demonstrated by a snippet of Perl code in Example 3.3. The code prints element with attribute and has to either escape the quotation mark in the literal string with backslash character or use other delimiters to start and end the literal string [Walloo]. As a result, less readable code needs to be maintained.

Example 3.3. Perl code printing HTML markup.

```
print "<A HREF=\"search.cgi?id=$id\">Search by id</A>\n";
print qq!<A HREF="search.cgi?name=$name">Search by name</A>\n!;
print '<A HREF="search.cgi?'
      . hrefparams($q, %params, restart => 1)
      . "\">Restart search</A>\n";
```

The rules for quoting of literal strings in Perl are summarized in the Table 3.2.

Table 3.2. String quoting and escaping in Perl.

string	character that must be escaped with backslash	variables interpolated
q!string!	!	no
'string'	'	no
"string"	"	yes
qq'string'	'	no
qq#string#	#	yes

In addition to that, the variables (`$id`, `$name`) in the Example 3.3 can contain ampersand or other character with special meaning in HTML, hence they should be escaped before using for interpolation in the string. On the other hand, it may be programmer's intent to produce markup via variable interpolation or function call, therefore each case has to be considered individually, which increases complexity of the programming task. Overall, the validity of the resulting document can only be verified once the application code was run, on its output. As the output is dynamic and can change with each invocation, validation cannot be done on the source code level.

Using function or method calls instead of literal strings to produce the output markup can help in keeping the output balanced, as presented by a Perl code in Example 3.4.

Example 3.4. Using function calls to generate markup.

```
#!/usr/bin/perl -w
use CGI ':standard';
print html(
    body(
        h1('Utilities'),
        p('First paragraph with ',
          a({ href => '/search' }, 'a link to search utility.'))
    )
);
```

However, applications are often more complex than this linear example and exceptions to this seemingly clean approach will likely be needed, to accommodate for various flow alternatives. As before, escaping of characters that have special meaning in the output markup is left on the programmer. In addition, function calls can be dozens of times slower than using literal strings which can decrease the performance of the system significantly.

CGI scripts do not necessarily need to be run as separate processes. Lately, interpreters embedded in the Web server (`mod_perl`) or handled by external application server (FastCGI) make it possible to write handlers that are run within the process of the Web server. No extra process has to be started for each new HTTP request, providing performance boost. The handler may even be able to directly use the internal application programming interface (API) of the Web server, to achieve specialized tasks. However, the way in which output markup is generated in handlers and scripts run within persistent environments remains the same as in classical CGI scripts.

3.3. Templating techniques

Templating systems extend the SSI technology by allowing full-featured programming languages like PHP, Perl (Embperl, Mason), Basic (for Active Server Pages (ASP)), or Java (Java Server Pages (JSP)) in HTML templates within special markup, replacing simple commands of SSI, or by providing specialized syntax for binding dynamic data back to HTML code (Template Toolkit, ColdFusion) [Harkins03]. When introduced, templating techniques came with promises to separate the content from presentation, thus making it possible for the application programmer to work on the business logic of the Web system and for the Web designer to focus on the HTML presentation. In reality, templating techniques tie the HTML markup to the procedural code the same way as CGI scripts do. In addition, the mix of wrapper HTML markup and in-line procedural code that produces HTML markup by printing it (either intentionally or by accident) gets even more complex.

An Example 3.5 comes from the PHP documentation and tutorial [PHP] and shows how HTML markup is printed by a PHP code, which is in turn embedded within HTML markup.

Example 3.5. PHP script producing HTML.

```
<html>
  <head>
    <title>PHP Test</title>
  </head>
  <body>
    <?php echo "<p>Hello World</p>"; ?>
  </body>
</html>
```

While in this case the paragraph markup around the string Hello World could have been also included in the HTML part of the page, printing HTML tags gets necessary in cases when dynamic number of elements, like table rows or columns, is to be produced.

In contrast with Server side includes that place the directives to HTML comments and have restricted set of commands and their syntax, sources of applications written in templating way are no longer valid HTML documents. The end tag `</p>` for element `p`, which is not open, will be reported by validator for both HTML 2.0 and HTML 4.01 Transitional DTD's for line 6 in PHP code presented in Example 3.5. Even if no HTML markup is generated from the PHP part, invalid HTML on output is easily achieved by using conditional templating commands that include only parts of the source in the resulting document. Thus, as with the previous approaches, the application can only be validated on the output end, once it is run.

Templating techniques gained great popularity among beginner Web developers because they allow them to start more easily and faster. If the embedded code is incorrect and does not work,

they are at least left with the skeleton of the HTML page. However, this dependence on incorrect or partially correct solution without any stress being put on the validity of the output is not a way to long term maintainability, and may even lead to compromised security.

Another snippet of code from the PHP documentation, displayed in Example 3.6, shows how input data is used, and by accident introduces HTML code injection (also called Cross Site Scripting) vulnerability. The data is injected into the output HTML page without any checking, making it possible to insert arbitrary markup to the output page and with client side scripting (for example using JavaScript) to compromise users' private data.

Example 3.6. PHP script printing data without escaping.

```
Hi <?php echo $_POST["name"]; ?>.
You are <?php echo $_POST["age"]; ?> years old.
```

The danger of this kind of vulnerability is not unique to templating environments and can be easily found in CGI scripts as well. They stem from the text-based nature of Web technology that makes building the applications seemingly easy. But with the ability of managing all details of the output using text manipulations, there comes a responsibility of addressing all potential cases and problem sources. Current techniques of building HTML output from individual textual pieces create too many places where proper handling can be omitted by accident. Moreover, readability of the source code is lacking when many syntaxes are mixed in one source file, as shown by real-life Example 3.7.

Example 3.7. Mixture of syntax in PHP script.

```
echo "<form action=\"./users/data_change.php?\".session_name().\"=\".session_id().
  \"\" method=\"POST\" target=\"_self\" name=\"user_form\"
  onSubmit=\"check_user_info(document.user_form);\">";
This example comes from posting to cz.comp.lang.php newsgroup. The Message-Id of the post is
<pan.2003.08.14.12.55.01.511800@pds.plz.cd.cz>.
```

3.4. Access to database backend

Previous sections focused on exploring how syntaxes interact and what issues arise when HTML markup is mixed in one source with special directives of SSI or full-featured procedural languages in CGI and templating cases. Most dynamic Web application utilize a database backend as a reliable storage, and Structured Query Language (SQL) [Kline00] is generally used as a standard communication language for updating records and running queries against database management systems. SQL is thus another environment that usually needs to be handled in Web-based systems. It comes with its own escaping rules and when used in truly textual manner without

any caution paid to passing parameters properly, it becomes a target of code injection attack similar to one shown in Example 3.6 with improperly treated user inputs.

Example 3.8. PHP script running SQL queries without escaping.

```
$link = mysql_connect("host", "u12", "qwerty")
        or die("Could not connect: " . mysql_error());
$result = mysql_query("update users set password = '$pass' where id = $id");
```

When the `$pass` variable in Example 3.8 that presumably comes from an HTML form where the user entered new password for his or her account in Web system is passed into the query directly without any checking, it can be used to manipulate the whole query to one with completely different meaning. Consider a value of

```
xxx', is_root = '1
```

which leads to command

```
update users set password = 'xxx', is_root = '1' where id = 752
```

This sets the password to `xxx` but also sets another field `is_root` in the database table to a new value. Using a comment syntax, the whole rest of the SQL command can be masked out and changed to unwanted semantic. Input value

```
' where name = 'admin' --
```

instead of legitimate password creates update command

```
update users set password = '' where name = 'admin' -- ' where id = 752
```

that resets password of completely different user, in spite of the fact that the value of variable `$id` might have been sanitized to contain correct integer value of user's identifier (752, in this example). The generic solution is to escape the unsafe characters in all values interpolated to the SQL query using an `addslashes` function. However, this function call has to be put to every place where values are pasted into SQL, which increases a potential for oversight and untreated vulnerability. In addition, care must be taken when the application is converted to different database backend, as various database system have different syntax rules for quoting and escaping literal strings.

Some database systems and programming languages remove the need for textual interpolation of values in SQL command by using placeholders in SQL queries and bind parameters as values externally binded, without using the values directly in the query. The Example 3.9 shows the use of bind parameters in scripts using Perl Database Interface (DBI, [Descartes00]).

Example 3.9. Perl script using bind parameters.

```
my $dbh = DBI->connect('dbi:mysql:localhost',
    'mm12', $mm12_password, { RaiseError => 1 });
my $sth = $dbh->prepare('update users set password = ? where id = ?');
$sth->execute($pass, $id);
```

Bind parameters were primarily designed to increase the performance of the database subsystems because prepared statement or query can be executed multiple times with different parameters. As they contribute to clean and safe code by defining an interface for passing data from one environment (the script or program) to another (the database part), many database drivers emulate them even if the underlying database system does not make use of them natively.

Placeholders and bind parameters are an efficient way of separating syntaxes of procedural application code from the SQL calls. Data is passed in as parameters, using normal function calls. Text manipulations can still be used to achieve certain advanced results, like fine-tuning the list of columns to be selected by a query or the ordering of the result set, demonstrated by Example 3.10. In these cases however, developers are expected to know the consequences of such dynamic manipulations, and never use tainted inputs directly.

Example 3.10. SQL queries can still be created dynamically.

```
my $dbh = DBI->connect('dbi:Oracle:prod9i',
    'oen', $db_password, { RaiseError => 1 });
my $name_style = ( $q->param('lastname_first')
    ? "last || ' ' || first"
    : "first || ' ' || last" );
my $order = ( ( $q->param('order') eq 'borndate' )
    ? 'date_of_birth, last, first'
    : 'last, first, dept' );
my $sth = $dbh->prepare("select $name_style, id from people where dept = ? order
by $order");
$sth->execute($department_id);
```

Web applications are built around textual manipulations, through which they generate the desired output. This results in syntaxes of many environments mixed in the sources. In the next chapter, we shall focus on interfaces that Web applications utilize.

Chapter 4. Lack of clear internal interfaces

4.1. Interfaces in current Web applications

The CGI and templating techniques are those prevailing in the Web development nowadays. Nonetheless, as shown in the previous chapter, they cause many languages mixed in one source file, with various syntaxes, quoting, escaping and binding rules. As a result, what would be a clear code turns into far less readable source, which increases chances for errors and makes debugging harder and longer. Externally, Web applications use open and standard protocols and interfaces [Fielding98]:

- Open Database Connectivity (ODBC, [ODBC]), Java Database Connectivity (JDBC, [JDBC]) and Perl Database Interface (DBI) for interaction with database systems are documented technologies. Backend-specific drivers are needed for most database systems, but the function calls in the client code are standard and reusable.
- SQL for database queries is textual and documented, even if each relation database management system (RDBMS) features its own extensions.
- HTTP for client–server communication is open and standardized.
- HTML for user interface and content markup is open and standardized.
- The same holds for Cascading Style Sheets [Bos98], used for styling of HTML documents.

Internally, a typical template-based Web application based is structured as shown in Figure 4.1. Each templating application thus includes the necessary top level HTML structure, the individual core application code, plus a code to form and markup the output data. Should the result be a valid HTML page, this code also has to include all the necessary escaping, like substituting `&` for each ampersand that may appear in the data and keeping produced document structure in synchronization with target Document Type Definition.

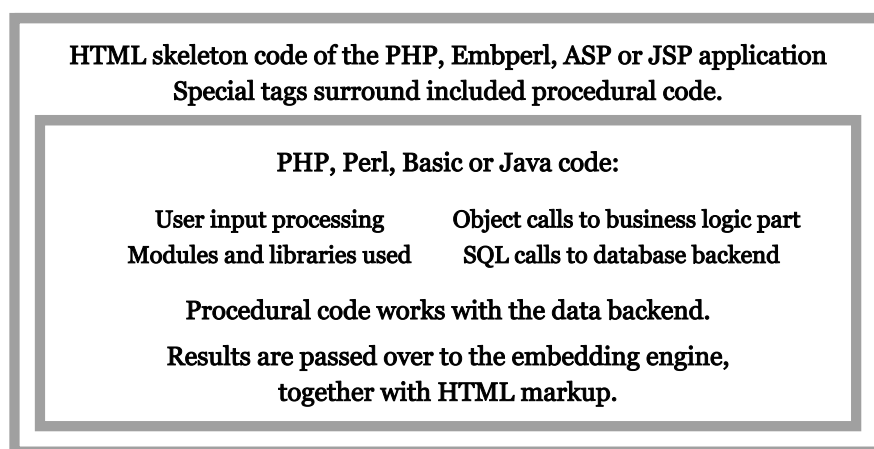


Figure 4.1. Structure of Web application based on templates.

Having the markup and the output processing code in each application program poses some complications:

- The applications are tied to a single output format, in most cases HTML. Should another format be desirable, such as Wireless Markup Language (WML) [WML] for mobile devices, comma-separated-values (CSV) format for machine processing, or any of numerous protocols targeted at computer-to-computer interaction, the application would need to be duplicated with another markup skeleton or procedural code, or postprocessed from the primary markup to the target one. Alternatively, external systems are forced to parse HTML.
- The flow control of the application is dependent primarily upon the presentation, not on the natural processing order. If the title of a page must come to output first and information about transaction processing later, the application must first retrieve the data for the title and only after that process the transaction and get its result, even if the reverse order might be more natural. Alternatively, the actions can be run in arbitrary order and the textual pieces of output markup with results stored in variables for later reuse. However, such approach diminishes the advantages that the templating style might bring.
- If a change in the existing markup is needed, like upgrade from HTML 4.01 to XHTML document type or modification of the design, as many as hundreds of applications would need to be checked and fixed for large systems. Such changes are often caused by changes in external conditions and cannot be avoided.

As current applications that are built using templates generally follow schema described in Figure 4.1, they have only two types of interfaces (Figure 4.2):

- External via CGI, processing input parameters from HTTP request and producing output document for HTTP response.
- Internal for access to data base backends and to other subsystems.

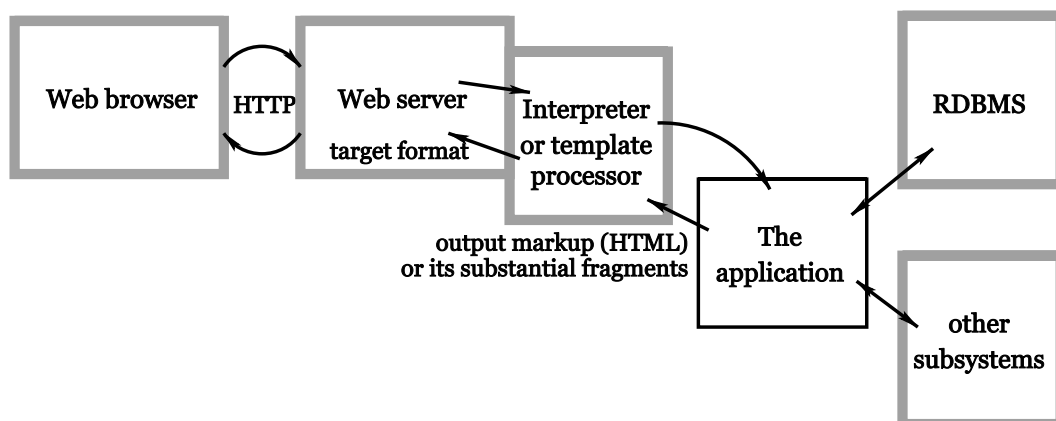


Figure 4.2. Interfaces in Web systems.

These are also the only places that can be used to setup regression tests and syntax validation. Having tests tied to HTML outputs would however mean that whenever any change is made to the design of the Web site, all tests would need to be updated to reflect the new output produced by the applications. In other words, such a test suite is too high level, hard to maintain and to keep up-to-date in the long run.

It is important to note that the design of the Web page does not mean the graphic part only. Most of current Web project use Cascading Style Sheets to fine-tune the colors, fonts, logos or positions of the elements on the HTML page. The stylesheet is typically stored in external file only referenced from the HTML pages. Still there is usually more than one element in the target markup that can be used to represent information in the output document. Consider a list of students, retrieved from the database. Should it be printed out as a list or will a table be better, as it makes it easier to show additional information (Example 4.1)? A need for reorganizing existing HTML structure may come from outside — a new CSS specification may allow better results if different HTML element is used, or the mainstream Web browser may be significantly faster with one way versus another.

Example 4.1. Representing list of students in HTML.

List:

```
<ul>
  <li>Albright, Sally (3)</li>
  <li>Gordon, Sheldon (5)</li>
  <li>Reese, Amanda (3)</li>
</ul>
```

Table:

```
<table>
  <tr><th>surname</th><th>name</th><th>term</th></tr>
  <tr><td>Albright</td><td>Sally</td><td>3</td></tr>
  <tr><td>Gordon</td><td>Sheldon</td><td>5</td></tr>
  <tr><td>Reese</td><td>Amanda</td><td>3</td></tr>
</table>
```

Web applications that do not utilize templating techniques share nearly all problems of the template-based ones. As they produce HTML pages, they are linked to this single format, which spreads the output markup across all the applications in the system, and prevent efficient maintenance. In theory, output markup could be hidden and maintained in predefined modules. Consider how a snippet of Perl code in Example 4.2 mixes processing of user input data (test for `really_delete` parameter that presumably comes from a submit button in HTML form), operation with the database record, and production of feedback.

Example 4.2. Script doing updates and producing HTML.

```

if (defined $q->param('really_delete')) {
    if ($dbh->do('delete from product where id = ?', {}, $productid)) {
        print "<p>Product $product_name was removed from the database.\n";
    } else {
        print "<p>Removal of $product_name failed:\n<pre>",
              $dbh->errstr, "</pre>\n";
    }
}

```

This code which combines three syntaxes (Perl, SQL and HTML) could certainly be hidden in a module which would leave the main application body in pure procedural language as in Example 4.3, with a function or method call to achieve the record removal.

Example 4.3. Script calling module.

```

if (defined $q->param('really_delete')) {
    print delete_product($dbh, $productid, $product_name);
}

```

The function `delete_product` would properly generate HTML output depending on the result of the database operation. In practice this change would only move the three-syntax code into another source file and would not solve any of the obstacles created by systems that lack internal interfaces and are tied to output markup language. Such a function cannot be used in application producing WML formatted output, unless the type of markup becomes a parameter — another complexity carried over to the lower levels of the setup.


Modular or object oriented approaches bring benefits when they encapsulate commonly used code or features that need not be visible to application programmers, while creating clean and well documented interfaces [Jacobson92]. Using them to hide the operations while passing all the ties down the call stack does not make the system more maintainable, as it does not separate individual actions — in case of Example 4.3, the module that provides the function `delete_product` would still need to be HTML-aware.

Scalable and maintainable solution should isolate application specific parts from those that are system-wide, whose presence in application specific files is only caused by shortcomings of the technology currently in use. Should many applications in information system generate various lists of students as shown in Example 4.1, the presentation of such a list in HTML form should not be coming from the application code. The application should rather communicate with the calling environment and pass back *the data* that it fetched from the database backend.

If the data transfer is done using well-defined and open interfaces, they can be used as places for tests and validations.

4.2. Communication among team members

When the Web systems grow larger, they are no longer developed by a single person but the development team cooperates. Many skills are needed to build and run enterprise Web system. They include but are not limited to:

- 
- Business analysts who transform customers' wishes into technical assignments;
 - Database analysts and programmers who create the database schemas and objects;
 - Application programmers who code the core application logic, talking to the database backend and handling user inputs;
 - Web designers who are experts in the technology of HTML pages and forms;
 - Graphic specialists who provide both the outline of the visual view of the Web site and the details on individual pages;
 - System administrators who manage the system background once the system went into production.

Each of these is a specialist in a separate area and except for cases when one person covers more than one field, their work fields are rather distinct. They usually work at different stages of the project development and do not meet regularly. Once the Web system has been deployed into production, it often evolves continually and so cooperation of these professions is needed at various and unpredictable points in time, when a change in functionality or design is called for.

Web applications built around single source file solutions, be it template or CGI script, create a single point in which all members of the team meet when a change needs to be done. However, application developers want to work with application code but do not like to deal with the design of the site, since it is rather remote from their qualification. Web designers on the other hand find their specialization area in HTML markup but not in the code of the application. The same holds for most other combinations of professions cooperating on a Web-based project.

Yet, current Web techniques either combine HTML templates with specialized tags that hold application code, or vice versa, print HTML markup from procedural applications, in one source file. Consequently, any time HTML details need to be changed, the Web designer has to edit a source code that also includes the business logic in Perl, PHP or Java, while the application developer has the procedural code mixed with external textual markup. This combination is clearly unfortunate because errors can easily be introduced to the area in which one is not an expert as a side-effect of a change in the common source. In addition, delays in committing

changes occur because Web designers might be afraid to touch a source file that also includes cryptic (for them) application code in Perl.

Shifting the markup generation to internals of individual business objects only solves the problem in theory because the mixture of external (HTML) markup with application logic is still present, only hidden at different level, as shown in discussion about Example 4.3.

In addition to one source file serving for multiple purposes, current frameworks do not promote formal communication of any changes that are done in the system over the course of time. When an additional piece of information needs to be added to the listing of student from Example 4.1, the change is most often driven by the presentation side — the requirement that new data should be placed to the resulting document. Both the code part and the markup part get modified, but the change is nowhere formally described. That makes it difficult to track what new attributes and information is processed by the page and how regression test suites should be extended to cover the new functionality.


In medium and large systems, it is desirable that competences of individual team members are distinguished and that they meet in a more formal way where changes and requirements can be tracked and communicated. What is the best way to allow all of them to work at high speed in their own area, and to conduct interactions with other parts of the system through well-defined interfaces?

Note

Existing styles of programming Web applications are well suited for small systems, where the size of a system is to be measured by the number of individual application and diversity of processes that it supports, rather than just by the number of hits the Web site receives in a day. Many Web portals with millions of visitors are very simple application-wise — a catalogue, a search, and a few applications for its content management. The catalogue may generate thousands of different pages, but they all are driven by one application. Internal information systems such as ledgers, CRM (customer relation management) or education management are much more complex, even if the number of their users is smaller.

Chapter 5. The goal of this work

In the previous chapters, we have focused on problems that current approaches to development and architecture of Web-based applications pose for development and deployment of scalable, maintainable and extensible systems. Based on that analysis, we shall focus on creating a framework that will minimize those obstacles.



The lack of internal interfaces must be addressed, to create needed entry/exit points that can be used for alternate output formats and markups, as well as for validation and testing. These interfaces will be primarily data oriented rather than markup oriented, as a change in user-visible markup should not cause change in interfaces. In the end, markup shall completely disappear from the application code and the business logic part. That will remove the multiplicity of syntax and drive the size of individual application programs down, to core parameter processing and data handling.

Clear separation of the application part and the markup postprocessing phase by a data interface shall finally provide true content and presentation separation, a promise that was given by templating techniques but that never materialized, for reasons stated above. As multiple output markups can be set, true reusability of the code and efforts will be possible. In addition, each of the specialists in the development team will be able to focus on its own part in its own source document or document tree, without interfering with others and being bothered by foreign syntax of other coding environments.

Prior to designing the solution, in the next part we shall gather background characteristics of Web-based applications.

Part II
**The Background: Characteristics of Web-Based
Applications**



Chapter 6. Network-based applications

The purpose of network-based computer applications is to enable users to work with data stored in remote computer systems. The users have their local computer or other end device — the *client* — at their disposal and the computer communicates via the computer network with other machines connected to the network — the *servers*. The categories of network applications can be viewed from many angles, such as architectural styles [Fielding00] that focus on the technical conditions and constraints of building the software. From the operational point of view, the main distinction of various approaches lies in different ways of how individual parts of this setup are distributed on the client or server machines and the level of communication over the computer network it takes to achieve a certain task in the course of work. From the business point of view, it is important to state borders of individual parts of the system — what pieces are under control of the Web system developers and what is common infrastructure, already beyond their control.

Possible models of the operational spread include:

1. Both application code and data reside on the client, synchronization with central server is done off-line, or not done at all.
2. Application code resides on the client and data (or the majority of it) on the server, accessed on-line.
3. Both application code and data are stored on the server, client serves mostly as a display device for handling user interaction.

6.1. Applications and data on client

The first model was the primary one until the beginning of the 1990s, with desktop personal computers (PCs) without any network capabilities, or connected to local area networks (LAN). The advantage of this setup is that the network does not represent a critical point in the application functionality, reducing probability that a network failure affects the core business operations. The disadvantage can be seen in many complex computers systems scattered across the organization, each holding a small portion of the overall data base. As data is stored locally on many computers operated by end users, a risk of its loss or corruption grows very high, since users' habits often overcome the machines and their ability to survive. The application software has to be maintained and upgraded on many systems, increasing the complexity of the task.

We can also see that this was the situation in which Alice from the Introduction of this work had to pay a visit to Bob's office or call him on the phone whenever she wanted any piece of information stored on his computer because there wasn't any other way of accessing the information. In this model, computers served merely as *information* technology, allowing Bob to

search through data on his PC's hard disk faster than he would go through papers in his filing cabinet. *Communication* needs had to be supported by more traditional means, like carrying the data on paper or on floppy disks.

6.2. Applications on client and data on server

The second approach, with remote data store on server and local applications using the server data base on-line utilizing computer network, is the typical example of a *client-server* operation. The data, which is more valuable in any business or institution than the applications themselves, is stored on a central server or servers, with the increased advantage of centralized data storage, maintenance, and security. Naturally, single central data store may possibly also create a single point of failure. With a reasonable maintenance and security policy, including backup and restore scenarios, with the possibility of data replication and off-site standby backups, the central data location provides a more stable solution than individual local computers in the long run. A small number of central data servers can more easily be managed by professionals in non-stop shifts, not leaving data in hands and habits of desktop computer users.

Nonetheless, with the client-server architecture, even if data is secure and well-attended on the server, one half of the operating environment is still on clients — applications. With increased stress put on wider on-line accessibility of information and increased number of users in and around organizations, maintenance of this load of client applications on local workstations creates a bottleneck in scalability of the client-server approach. Any change on the server side that changes external conditions and requires upgrade of client software, be it change in database schema, in communication protocol or simply a new feature, puts stress on maintainers of client machines because upgrade of both sides often has to be synchronized.

On top of this, we have to take into account problems arising from different operating system bases and coexistence with other applications. Unless all users of the system can be forced to use the same versions of all software (or the budget of the developed system allows for providing them with such uniform environment), compatibility issues will put additional breaks on the wider spread of the use of information system built using the client-server model. Any additional system that the user may need to access will require a new client application being installed and maintained on his or her local workstation. This requirement may be fulfilled inside an organization but is unrealistic with external users — customers, partners, and general public. They will not be willing to do any changes on their local computers installation in order to use an application that was planned for them. In most environments, end users will not even be technically able to install and run any additional software, beyond the standard suite provided by their local computer administrators.

6.3. Both applications and data on server

The third option in setting up the network applications places both data *and* application code onto server, while the network and client machines are used to display the output of the running application and handling user input. Here, the client only serves a presentation function. Two types of technology fall into this category: terminal-based applications, and the World Wide Web.

In both cases, there is a uniform communication and presentation protocol, which enables using *one client* software to access *many server applications*. If they use a standard way to transmit the presentation data to the client and use the same methods of handling input that goes from the user to the server, the specifics of the application being run on the server do not matter. It will always be possible to use any client machine with only minimal requirements: the computer network which provides means for on-line data exchange between parts of the system (communication) and the presentation layer which gives the user the same or functionally similar visual experience.

Minimal client requirements mean that *client side does not need to be considered an integral part of the information system*, which is an important feature of this option from the business point of view. Anybody who has a client software that supports standard interfaces (communication and presentation), plus the network connection, has fulfilled the technological preconditions for using such a network application. Which version of client side software or by what vendor or author is being used is of no interest to developers and maintainers of the server applications. These differences also do not affect functionality of the system and users' work with the applications stored and run on server.

For terminal applications, the core interface here is terminal emulation. In recent days when the private traffic goes across insecure public networks, security requirements are usually added, providing encrypted connection at the base level and the terminal emulation on top of it. Examples include secure shell (OpenSSH) implementations with terminal emulation either coupled into one application (PuTTY, Ttssh), or provided by separate programs (xterm, rxvt).

On the Web, the communication protocol is the HyperText Transfer Protocol (HTTP), often used over the Secure Socket Layer (SSL) [Freier96]. The presentation layer may come in a wide variety of World Wide Web Consortium (W3C) formats and languages, with the HyperText Markup Language (HTML) as a base standard and also the least common denominator which each Web-based application supports. There is a range of client programs — *Web browsers* — available for various computing platforms, many of them provided free of charge: Mozilla, Microsoft Internet Explorer (MSIE), Netscape Navigator (NN), Konqueror, Opera, to name the most common ones. Availability of client programs for all client operating platforms has made

the Web the technology of choice for applications where a large audience is anticipated or required.

The Web technology has brought many changes and improvements over the terminal applications. The user interface is graphical, following the latest trends in work ergonomics research. From the practical point of view, it provides application developers with more flexible presentation framework than the 80 column wide, 24 rows high terminal screens. Text nature of Web protocols make it easy to build new applications on top of them, and widespread availability of Web browsers make it the preferred choice.

6.4. Conclusion of classification

Note that for the purpose of the previous classification, all aspects of internal setup of the system, like multi-tier applications or load balancing and distribution across the number of computers were omitted. For example, the internal software architecture of Web system may be client-server because it consists of separate application and database machine. However, the main focus of this classification is on the distribution of data and application logic between the client part and the server part, where the client part of the information system is defined as that piece of the system which needs to be added to allow an additional employee, customer or citizen to use the system. Setup with thin clients in multi-tier or client-server applications only fall into the category of both data and applications on server if the thin client is not application specific, meaning that it does not need to be installed and maintained for each additional system or function that the user would like to use. If there is anything specific that needs to be installed on the client machine (or other hardware equipment) prior the user being able to work with the system, a burden of application maintenance is placed on the client side and the system falls into the category with applications on the client side.

We have also avoided any discussion about mobile code approaches, such as virtual machines like Java Virtual Machine [Lindholm99]. For the purpose of our classification, they fall into the category with the code on client, even if they use an on-demand distribution method. Still, there is a huge gap between a number of computer systems that provide Web browser software as commodity, standard part, and those that come with virtual machines. Web browser can be assumed to be present on any computer, compatible virtual machines cannot.

We can briefly summarize the benefits and problems of individual approaches in an overview Table 6.1.

Table 6.1. Categories of network-based applications.

	applications on client		applications on server	
data on client	pros	not dependent on network failures	not a reasonable combination	
	cons	no central maintenance and backup of data, maintenance and upgrades of software on many clients		
	example: desktop PCs with local dBase/Clipper applications			
data on server	pros	data in well maintained central location	pros	data in well maintained central location, client infrastructure is not part of the system
	cons	network failure may affect system operation, maintenance and upgrades of software on many clients	cons	network failure may affect system operation, client infrastructure is not part of the system
	example: desktop PCs with applications connecting to database server via networked ODBC connection		example: Web applications on server, used via Web browsers	

Note that for the category with both data and application on server, the fact that client infrastructure is not a part of the system is included both in pros and in cons of the approach. It represents a great saving in money and time that would be needed to purchase, develop or deploy and to maintain the client part because it comes as a standard part of any modern computing device. On the other hand, the client part can no longer be controlled. If the computer and operating platform vendors decide to cease support for Web browsers for any reason, the information system has to respond to this external change and adopt new communication and presentation means.

Chapter 7. Request and response nature of HTTP

Web-based applications are built on top of HyperText Transfer Protocol [Fielding99]. Typical round of communication is conducted along the following steps:

1. Communication is initiated by the client. It opens TCP connection to predefined port on remote computer and the connection is handled by HTTP server.
2. The client sends in HTTP request, based on previous input from the user or from parameters it was passed.
3. The HTTP request is received and processed by the server.
4. The HTTP server generates HTTP response, either by fetching static document from a filesystem or from database, or by executing application code that produces dynamic output. The response is sent back to the client.
5. Client receives the response and depending on the metainformation, it decides on next action — HTML page or graphics can be rendered on screen, compressed archive file stored to disk. The HTTP response can also hold redirection or error information.
6. The TCP connection is closed. Alternatively, client can keep it open in anticipation of another HTTP request being sent in short timespan (keepalive).
7. User may initiate another request, using hypertext links or submit buttons in rendered HTML page. Another request may also come as result of redirection information from the previous HTTP response. The client can utilize the same TCP connection that was kept open utilizing keepalive (step 2), or open a new one (step 1).

Besides keepalive, HTTP version 1.1 also allows request pipelining as a means of increasing throughput and reducing network latency. In pipelined mode, more HTTP requests are sent out by the client without waiting for immediate response. In all cases, however, it is the client that starts the communication with an HTTP request.

The HTTP request, sent by the client to the server, consists of three parts:

1. Request line, specifying the request method (GET, POST, HEAD, or others), request URI, and protocol version.
2. Request header that holds fields with metainformation about the request content and expected response, like content type, accepted types, character sets or transfer encoding, or information concerning advanced features of the connection (keepalive, caching).
3. The request body. Its payload is dependent on the request method and request headers. It can also be completely missing.

See the Example 7.1 for a header of real-live HTTP request sent by WWW browser Galeon to HTTP server running on port 8080 on server www. The URL of this request, as specified by the

user, is `http://www:8080/cgi-bin/search.cgi?host=all`. The Accept head field value would come in as a single line, it was wrapped here for presentation purposes.

Example 7.1. HTTP request with GET method.

```
GET /cgi-bin/search.cgi?host=all HTTP/1.1
Host: www:8080
User-Agent: Mozilla/5.0 Galeon/1.2.10 (X11; Linux i686; U;) Gecko/20030314
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,
text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: cs,en;q=0.5
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-2,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

Note that the request body is missing from the GET request. It would be present with POST methods and in other less common request methods, carrying either form data, long content or multipart payload. The HTTP server finds the resource that the request was addressing, retrieves it, and forms an HTTP response. The response consists of a header with meta-information and a body with the actual data content.

Usually, the URI of the HTTP request is mapped to a resource in server's filesystem, however, the mapping can be arbitrary. In this case, `/cgi-bin/search.cgi` suggest that the request will be handled by a CGI script, but it can also be a static document, database record or a complex `mod_perl` or JSP handler embedded in the server, depending on server configuration. Not only the resource mapping, the resulting content as well is completely under control of the server and its configuration. In the Accept header, the client specified as the last fall-back value `*/*;q=0.1`, meaning that it can handle any content type, with 10 per cent preference. So even if the request might have come from user following a link on HTML page, the response is not guaranteed to contain HTML document.

Should the request be passed to a Web application by the HTTP server, it will be provided with the content of the request via public API (Application Programming Interface).

- For standalone CGI applications that run as separate processes forked by the HTTP server, the request details are passed in using environment variables and standard input. The output is then expected on standard output, in the form of HTTP response, including HTTP header holding information about the response content type.
- For applications run in embedded mode withing HTTP server's space, direct interfaces using library calls are usually used both for passing the request content to the application and returning the response. See Example 7.2 for demonstration of the Apache/`mod_perl` API.

Example 7.2. Simple mod_perl handler.

```
sub header {
    my $r = shift;
    $r->content_type("text/plain");
    $r->send_http_header;
    $r->print("header text\n");
}
```

Typical application's purpose in life is to work with the backend database records and to search through the database to retrieve information that the user requested with the HTTP request. The response content produced by the application execution will then reflect the changes done and include the data retrieved from the database system. The most common styles of producing output markup in were analyzed in Chapter 3. The response is passed back to the HTTP server which can do further manipulation on it, like adding header fields, or even run additional processing of Server Side Includes tags.

After the application produces the desired output, it will finish its operation — the process can be ended, memory occupied by the embedded application can be freed. Especially with embedded systems, the setup tries to keep the application code loaded and parsed, with maybe even database connection kept open in persistent manner. However, the data of the request that the application just handled is no longer needed. When the next HTTP request comes and is processed, it might be from different user from different place in the world. The task of the application is always limited to processing one HTTP request.

When the HTML page which came as a response to previous HTTP request is rendered by user's Web browser, the user can follow with his or her work. When writing into textfields or clicking on popup menus, checkboxes and other elements of GUI (Graphic User Interface) provided by the page, only the status and content of the form on the client changes. While the user is working with the presented information in the local browser, there is no connection between the client and server (short of possibly keeping keepalive TCP connection open, a low level, technical extension). Only once the user initiates another request to the server, typically by clicking on some submit button, new connection is opened (or keepalive connection reused) and HTTP request send to the server.

Figure 7.1 shows that the network connection is needed only during the request/response phase, not while the user is reading the page and working with forms on it. In reality, the ratio of the time which the user spends dealing with the page to the time that it takes to server to process the network connection and run the application will get much bigger. On a typical server, request processing will take milliseconds or a few seconds, while users move through pages at slower pace, dozens of seconds or minutes per page.

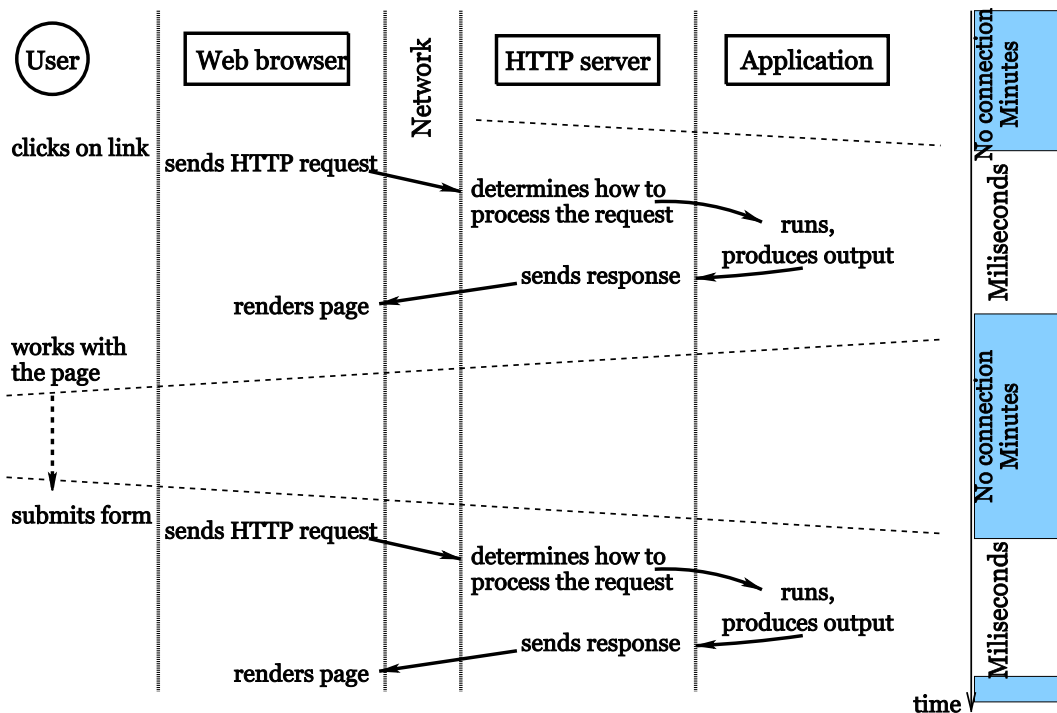


Figure 7.1. Requests and responses in user's work with Web-based system.

Among requests, no connection is needed, therefore no resources are utilized on the server during those time intervals. This approach is different from the connection and session handling in terminal-based applications. With those, there is usually a one-to-one relation between a running application process on the server and a connection that links this application to client's screen. The user logs in and for the whole duration of their work with the remote system (which may span days or even weeks), the server maintains a network connection and running application, holding the state of user's session. These applications consume resources of the system (network connections, open filehandles, processes, memory).

In the Web environment based on HTTP, the application only runs very short span of time, to process incoming request and produce response. Another incarnation of that application will come only when the next request is submitted, which means that the processing power of one Web server can be *multiplexed* among hundreds or thousands of users who work with the system concurrently. Due to the fact that users spend long amounts of time dealing only with the user interface on their clients, HTTP server can shift its resources to other users, without visible impact on users' experience. This processing model contributes substantial part to scalability of Web-based applications. Not only does not the client part need to be taken care of during deployment and production maintenance, but also many users can share one or small number of Web servers.

The statelessness of HyperText Transfer Protocol also creates new problems and obstacles, especially when used for traditionally session-based, transaction-driven tasks. As the World Wide Web was originally designed for highly distributed hyperlinked net of independent information sources [Berners96], it lacked native support for environments where it is used for continuous daily work with one information system. Reopening of TCP connections each time new HTTP request is sent to the server by active user or by page with many inline graphical elements can cause slowdown. That is why, connection keepalive and request pipelining were added to later versions of the protocol to address these potential bottleneck.

The most important however in session oriented environment is the fact that no information survives on server among individual requests of one user, unless specifically handled. In fact, the notion of *session* is rather blurred on the Web because the stream of actions by one user is not linear and does not have strict ending. Users can easily fork their work by opening two browser windows and continue in parallel operations, return back using either direct URL of one of their previous steps or by using navigational elements of their browsers, or simply abandon their work at any time. As there is no connection between the client and the server when users are working with the local information produced by the HTTP response, they can easily go to another Web address or just close the browser. HTTP server has no way of contacting the client back to figure out its state, once the HTTP response was sent out. Therefore, any information that is needed across multiple invocations of the application for given user has to be explicitly passed to it.

There are two basic possibilities¹⁾ how to keep data between invocations of the application code:

1. Passing values using hidden values in HTML forms or query parameters of URLs.
2. Storing current data in data store on server and only pass one identifier through the HTML form or parameter.

In both cases, the data has to be reinstated by each new application run, either from the content of the HTTP request, or indirectly, from the backend database. Of course, any data that the user had chance to change in the form need to be taken afresh, from the form data in HTTP request.

By keeping track of user's actions via passing the data back and forth through HTML forms, certain features of classical sessions can be maintained. Most prominently, keeping track about the identity of the user and his or her single stream of communication (thou HTTP authentication

¹⁾ Using HTTP cookies for this task is a questionable practice because it prevents users from working in parallel in two windows with one system, poses a security thread, and is not guaranteed to work for all users.


is the the preferred method for authentication in internal systems). Nonetheless, this session emulation cannot be used for database transaction handling, even if the database management system used as the backend storage supports transactions. Collision detection is a preferred means of handling concurrent updates of the same data element in stateless Web environment [Pazdziora01a].

The fact that any data that the application needs has to be passed in and that there is no implicit persistence marks the main difference from other user-oriented interactive environments. In most current GUI-based applications, the interface is first created with callbacks registered to individual widgets or actions, and after that the whole process is event-driven. With Web applications, the handling of user inputs is done on client, potentially with a help of client-side scripting (JavaScript/ECMAScript), but to the server the data only comes in isolated HTTP requests. The invocation method is thus *function call*. Each HTTP request carries in all necessary data as its parameters. It may be either actual values or data identifiers, in case of session handling using server-side storage. Any additional information that the application needs to correctly process the request and generate the output has to be fetched from the backend data base.

Chapter 8. Centralized server part

Web-based applications and information systems inherit the generic advantages of applications that are stored and maintained on the server side together with the data storage. In this setup, the local client computer only provides the presentation functionality and handles user input and output. In this chapter, the most important features of the server part will be briefly.

8.1. Central and professional maintenance



Instead of providing maintenance of data storage and application environments on many and all client computers, these are conducted at one place, on the server. Here, financial and personal resources can be more efficiently directed to a non-stop support of secure and stable environment, rather than hunting through thousands of computers to achieve backups of data, log surveillance, and installation of security bug fixes and upgrades. Internally, the server side may be further distributed to achieve maximum performance and security, for example using load-balancing techniques or off-site stand-by database. Still, the whole server part is under control of the central IT team.

Since applications run on the server and communicate using open and well-defined HyperText Transfer Protocol, it is possible to log and audit the communication to detect any irregularities in the system, either on-line or when a check is mandated.

IS MU Fact

In the Information System of Masaryk University (IS MU) project, internal watchdogs and reporting often allowed to fix a problem before it was even reported by the user because the malfunction was detected on the server side and reported to the development team via email.

On the client side, only the standard operating environment is needed. As Web browsers come as a standard part of any operating system distribution nowadays, its installation and maintenance is a part of the base client machine equipment, not of the information system that will be used via the Web browser in question.

IS MU Fact

During over four years of the IS MU operation, developers did not need to go out to solve problems with software on client machines. They were all handled by the local administrators or by the ISPs because these client issues with operating systems, browsers or printers were generic problems, not related to IS MU applications.

8.2. Fast development cycles

Web-based applications tend to be simple. Since they are built on open, text-based protocols and formats, simple applications based on CGI can be written in a matter of minutes — for the simplest case see Example 2.2, real applications would add forms and data processing. The use of scripting languages [Ousterhout98] greatly reduces the length of the development cycle, interfaces for HTTP communication and for utilizing HTML forms are open and well described. The processing model of the Web-application is request/response leading to isolated function calls to appropriate application code, as opposed to complex relations in event-driven environments.

The simplicity of basic Web-application means that a Web solution does not need to take weeks or months to be build. In addition to shorter development times, the distribution times are minimized as well. Distribution of new Web project is limited to uploading the software to the Web server, which is under the control of the developers, not in hands of hundreds of individual users or their local administrators. As with the initial release of the Web application, new features and bug fixes can be implemented into running system with similar ease, since no synchronization with the external world is needed.

On the Web, any project can start small and expand with the growth potential of the business, campaign, or with the change in organizations' administration. The cost is under direct control in the central place and Web applications can evolve to handle changing needs of their owners, to focus on the target. Being able to release new feature, announcement or product within minutes with no extra cost is a big rationalization step.

8.3. Instant upgrades

Web-based applications redefine the notion of software versions and distributions. Since the application logic and code is stored on the server, the end user never has to install the application software, never sees it. This decreases the importance of producing application software in discrete packages which are necessary if one software base gets distributed and installed in thousands of copies. If a faulty software gets packaged and released to users, bug fixes or new features will only be released and distributed after months, with extra installation efforts needed.

On the Web, any change made on the server is seen by all users the next time they access the server. Without any action on their side, those thousands of users start benefiting from the new feature, improvement or fix instantly. For this reason, changes can come in very small chunks, adding little touches towards system's final perfection. The ability to gradually modify and fine-tune the system makes Web-based applications well fitting into environment where requirements placed on the information system change all the time, be the changes small or substantial.

Indeed, this may sometimes bring unexpected surprises to the users because the look of the application, the forms or even logic has changed without them initiating or noticing the change. The users will not be able to go back and use the previous version because they liked it better — it is no longer installed on the server, no longer supported, and no longer available. Caution has to be taken to communicate substantial, visible changes to the core users (people who use the system daily, as their primary tool) and to make only incremental changes to existing applications, so that the changes do not shatter the users' overall experience.



IS MU Fact

The IS MU project has never released externally visible *versions* of the software. From users' point of view, all that matters is the current information system available on the Web and its current functionality. That leads users into playing a more active and cooperative role because they learn that problem report or feature request will be addressed and that there is no reason to wait with reporting them. Moreover, when a bug report comes two weeks after the problem occurred (often by a vocal note in a hallway: "by the way, a couple of weeks ago I spotted a problem in the system"), users are asked to retry their actions in the current system — chances are, the fix has already been applied.

Since changes can be deployed into the production system immediately after they are verified and tested, the difference between the development and production branch of the system never grows too large. Thus, team supporting the production operation can get better response from the development team because code bases are kept as close as possible and it does not take much effort and time to backport a bug or feature request report. Alternatively, one team can support both the development and the production operation, should the system be developed in-house.

8.4. Scalability using external resources

As Web-based system can be accessed using a generic tool, network-connected and Web-enabled computer, no special installation and care needs to be taken for client machines. There are no software or hardware parts specific to given Web-based application that would reside on the client. That way, system is accessible and fully usable from anywhere, growing its potential user base beyond limits of single organization. There are two groups for which this scalability can be utilized and provides advantages unseen in systems based on other technologies:

- customers can access the system on-line, increasing the base of potential customers;
- staff of large institutions and people affiliated with them, like students of universities, can access the system from outside of the internal computer network, decreasing costs.

Especially for universities and public institutions, it is important that students, teachers, partners, and public may become users of the system without needing any special software, training or support. In addition, the system can be accessed from any computer in the world, diminishing requirements put on the number of in-school computer seats. Thus, Web-based systems are superior to systems dependent on computer labs and specialized client-server software.

IS MU Fact

In 2001, Masaryk University started a project of *e*-applications. The application for study was moved into the information system and onto the Web. The statistics of two-year experience is summarized in the Table 8.1.

Table 8.1. Numbers of applications to Masaryk University.

academic year	total	via Web (<i>e</i>)	percentage of <i>e</i>
2002/2003	37716	10635	28
2003/2004	44030	19396	44

Since the application process is in large part pure information manipulation, digitization of the data from the very beginning was an important rationalization step. It also established more flexible way of communication between faculties and applicants, switching from on-line phone or costly paper mail to off-line email. Applicants worked with the system mainly in the evening and late evening hours (67 per cent of applications were entered between 19:00 and 01:00), so did not place any additional throughput burden on it. Part of the work that was previously done by the administrative staff was shifted to thousands of anonymous applicants, from school's computers to home computers of those users, and from the peak office hours to times when the system is mostly idle.

8.5. Performance

Web-based systems are built around the HTTP processing model, with generally stateless interaction and request / response batches that only tie server resources for minimal time intervals (see Figure 7.1). Client requests are multiplexed in time and they share the processing power of the server. It is common to have one average computer with a Web application to serve hundreds of users who would otherwise (as in client-server model) have to have their own copy of the same application written for their local operating system and also run on their local computer. Since the presentation part of the Web browsing does not require the same processing power as the full applications, less powerful (and less expensive) computers are sufficient when the system is switched to Web-based; with a chance that upgrades for performance reasons will not be needed at the previous pace.

The major potential limits of a Web-based application are in the network bandwidth and in the performance of the application servers on the server side. With reasonable sizing of the data chunks (HTML pages), good user-perceived performance can be maintained even for home users behind dial-up lines.

Since all the processing power is concentrated in one place, resources can be efficiently targeted on performance of the server. Both the database and application part of the server side of Web-based application can become a bottleneck, and solutions like clustering [Casalicchio2] or caching can be more easily planned on the central site, without clients being involved in any change.

IS MU Fact

The IS MU struggles with peak loads especially when actions involving thousands of students, like registration for courses, seminar groups or exams, start at one time. Students then try to get through and be the first ones who click and get the best deal. As a result, the load of the server system grows to a couple dozen times the average. This raises the question of planned throughput of the system that should be supported. There will always be the danger, especially on the Web system with users coming from thousands of computers around the globe, that overload will bring the system into stale. Each organization then faces the decision of trade off between the price of the solution (either more and faster hardware, or software fine-tuning) and the danger of the system not being sufficiently interactively responsive.

Actions that cause the overload can be routed to separate part of the system so that they do not slow down the rest of the users. But still, some of the user requests at a given moment will not be processed or will not be processed within specified response time. It is more reasonable to tackle this problem by changing the processes and/or study rules so that the "timestamp competition" does not take place at all. Assigning limited resources (like above-mentioned courses, seminars or exam terms), using other criteria than the second in which the student managed to register, is the viable solution.

Chapter 9. Client side highly distributed

9.1. Uncontrolled client side

Web-based information systems, especially those aimed at user base where the client computers and software cannot and are not expected to be under the control of the institution running the system, have to deal with unexpected changes in external conditions. These changes may affect users' experience or even the ability to work with the system. New versions of Web browsers bring minor incompatibilities — either breaking the rules set by standards and protocols or vice versa, when they start to be compliant with a standard that they were breaking. As a result, Web pages that use unofficial extensions or simply have bugs in them are no longer displayed and operating correctly.

Developers of Web-based applications have to be able to update output produced by the applications to match these external changes — to fight or to resist them does not make sense because thousands of users will not downgrade or change their software in order that it becomes compatible with one information system on the Web. The ability to *follow the trends* and to evaluate the changes reported, even if all the combinations of systems, browsers, additional libraries and facilities cannot be tested on-site, is an important skill set of a professional Web developer team. Hand in hand, this leads to conservatism in adopting new technologies and forces into minimizing requirements that are placed on the client side. Developing a Web system in such a way that it is usable with six-years old Netscape Navigator 3 or with text-based browsers like Lynx is a great way of preventing fatal incompatibilities with wildly changing sets of Web browsers in use today.

For developers used to traditional system architectures, Web-based environment may pose too much uncertainty. Indeed, developers of large production Web systems can seldom afford to be technology leaders. Instead, they have to learn to follow trends, so that

- features, even those based on published standards, are introduced only when they are supported by substantial percentage of browsers in use;
- new problems and incompatibilities of new versions of browsers are addressed in a timely manner.

While client side is not controlled, it must be monitored constantly and the eventual changes must be addressed in the system, so that compatibility is maintained both with mainstream browser versions and with minority ones. This fact leads to conclusion that Web-based applications can *never be considered finished*, even once put into production. They operate in environment which evolves, and so must the system, in order to maintain unaffected usability, performance, and security levels. It is a common misunderstanding that no costs will be involved

once the system goes live. Certainly, maintenance only consumes a fraction of resources needed to develop the application and only occasional checks will be necessary for small systems. For large, internal administrative Web-based systems, the maintenance tasks call for full-time, albeit small, team.

9.2. Increased feedback potential

With more users of the system who are not controlled by any central authority, feedback, suggestions and complaints will be arriving from them, concerning all aspects of the system — technical issues, organizational and factual ones concerning correctness of the information provided and presented by the system. Communication channels have to be established to route these messages to the personnel responsible for them based on the structure and processes in the organization. Often, users will not be able to distinguish wrong data from a bug in the application that presents the data. The team maintaining the Web system has to communicate any additional questions that may be necessary to resolve the issue, and take users' input seriously.

Feedback that comes from production environment is more complex than responses appearing during earlier stages (analysis, development or testing) because only in production a large number of users will be able to relate operation of the information system, ideas, and processes that it promotes to their local environment, to their daily work. Pre-production feedback is based on abstract problems, situations, and speculations. In production, the system together with its end users is stress-tested by real life tasks and deadlines. The production feedback from users will have a higher quality than the feedback from people only loosely connected with individual parts and actions of the system. Direct communication with all users and the ability to deliver a bug fix or new extension is a great potential of systems based on the Web technology. Development and maintenance teams become partners with users in bringing the expected utility. This potential can only be fulfilled if pieces of the production system get released to users as early as possible.

9.3. Cost efficiency

The fact that client side of Web-based system uses a stock hardware and software that even comes standard with current network-enabled computers, personal digital assistants (PDAs), and other end user gadgets, has a direct positive effect on the budget requirements of the system. Costs of the client system maintenance is distributed among the end users rather than being part of the system deployment. Responsible departments or home users see investment into core IT tools as their natural task. In fact, the marginal cost for adding a new end-user computer into the system is virtually zero not only for the system, but for the end user and his or her budget as well — nothing new needs to be bought or maintained.

IS MU Fact

In peak days of the year 2002, over 68 per cent of users accessed the IS MU from external networks, not from the network owned or maintained by the Masaryk University. The year average of accesses from outside networks is around 53 per cent. In the year 2003, the numbers of external accesses are even about two points higher. These figures include only *authenticated* accesses by students or members of the University, or people affiliated with it, who use IS MU to achieve their routine administrative or communication needs. With the random public traffic that is anonymous, the number of external hits gets even higher.

9.4. Web in all information-handling environments

The huge boom of e-business operations that benefited from increased numbers of potential customers, who can be served without location restrictions, across borders and during their local day, without necessary 24-hour full customer service staff, exploited the Web for classical business-to-customer sales operation.

Yet the sale of a product or service is only the last step, even if here the benefits are directly multiplied by the increase in numbers of users and consumers. Internal operations of firms and organizations are slower to switch to the Web and to direct access to core computer system. However, for employees and partners the increase in efficiency can be similarly strong. The effect will be seen even more significantly when Web-based systems take over the operation in organizations and institutions that do not have the classical advertise-order-pay-deliver scheme, where the product is not clearly distinguished and the service is a long time process, with quality and satisfaction being the target: universities (and educational institutions in general) and public and state departments. Experience described in this work come from university operations but analogies can be found in many other places — after all, information is the core of education, administration, and business.

Chapter 10. Use of extreme programming

The standard methodologies of software engineering all share one common idea — that change done in later stages of the project or errors and omissions that slipped through from analysis and design to implementation or even operation stages are very expensive to attend for. Moreover, the price of the fix or change increases with time, as the project is getting nearer into finish. Strict attention is paid not only to the quality and correctness of the project specification and analysis, but also to its completeness, in order to avoid as many changes, later down the implementation chain, as possible. If change is deemed bad, it must be prevented by analyzing all and everything prior to any other work. That consequently extends the time before any production results can be expected from the newly implemented system.

With classical implementation and distribution channels and typical client-server or even PC based architecture of the system in question, this aspect is understandable — if the software has already been shipped and installed on thousands of computers, cost and complexity of any change done afterwards gets multiplied by the number of installations and end points involved. However, as explained in previous chapters, with Web-based applications no application specific code or data reside on the client. Everything that forms the application is stored on the server side, no matter how many users and client computers use the production system. On the client machines, only standard, stock hardware and software is needed — computing device with network capabilities and a Web browser.

That consequently means that any change that needs to be made after the system has already been put on-line and is used by the customer (or by the public), is only made at one place, on the server. Users will activate the code with new features or with problems fixed next time they request an action from the system, fully transparently on their side.

As far as the simplest cases of sincere bugs and omissions are concerned, when for example a bad select command has slipped through the testing phase and the application gives wrong results, a fix, testing and integration into the production tree on the server may take as little as five minutes. Even adding new feature does not take longer because of the technical aspects of deploying the new code. Does it mean that the standard methodologies do not hold for Web-based projects?

10.1. Specification of the methodology

A new view on problems of software engineering, focused on the speed and flexibility of the process, occurred in the late 1990's. This methodology is called *extreme programming* (often abbreviated as *XP*) and it consists of many requirements, conditions, and suggestions that drive

the software development into efficient, flexible, low-risk, and controllable way of building the product. The overall introduction can be found in [Beck99].


The basic core rule is *develop with change in mind*. It reflects a frequently encountered situation when external or even internal conditions of the project change and the classical methodologies lead the project into costly changes that often mark failure of otherwise well-planned, well-analyzed and designed project.

The methods that extreme programming uses to prevent these failures or at least decrease risks of them happening, include:

- Project starts with very small set of features which can be delivered fast. That gives the customer greater control over the project, as it does not take years before anything comes of production stage. And if they decide to cancel the project, this decision is made earlier, so such failure is less expensive.
- The first release of the software product should include the core functionality and bring most to the business of the customer. That way, the value of the software is maximized.
- In each subsequent iteration, small and most important tasks and features are worked out. Anything that can be postponed into later is postponed. Thus, every day and week spent on the project is utilized as efficiently as possible.
- The release cycles are extremely short, and so are iterations inside these cycles — as short as minutes. That way the team always works on the most important parts of the assignment. Consequently, schedules do not slip because the developers are not overwhelmed by numerous individual tasks. They only move to the next task when the previous iteration is closed, with no outstanding errors.
- Short release cycles also mean that there is less change in requirements and in external conditions during one release development. Changes are not a problem if they do not step on toes of programmers currently working on a feature, if the change required by the customer concerns different parts of the system. Since the development is done in well-separated discrete iterations, it is possible for the customer to insert requirement for a change into the queue with proper priority and it will be picked up when previous iterations are finished and developers get to focus on the next item in the list.
- Extreme programming stresses the importance of comprehensive tests, both low-level (functions and methods) and high-level (programs). Tests are run regularly after each change during integration, which does not let bugs to go unattended.
- There is a strong link among features, tests and actual application code — features get formulated among others in the test suite, and even the customers are welcome to add their own tests. These tests then form targets which the software development tries to match and also ensure that subsequent changes in the code do not to break any existing functionality.

Test suites are written prior to the start of the application coding, and extended during the development cycles.

- The customer is required to be part of the development team in order to be constantly aware of the progress and to be in charge of the priority list.
- Programmers receive short, well-defined tasks during iterations, so backlog does not develop on shoulders of one person. Also, pair programming when two programmers work with one computer and help each other to keep drive and quality, as well as communication inside of the team are promoted.



Implementing only the highest-priority tasks and features in each task and deferring the less important ones is a strong strategy towards gradual success of the project. If the customer has a high-priority need that will be addressed first, they will put more effort into getting the requirements sorted out well, and the new feature will be available as fast as possible, without waiting for other less important parts. Those lower priority tasks would certainly take development time, but lower priority put on the option also means that the business can run without it for a while or that the exact requirement is less clear at the present time than those parts that the business is desperate to have in place.

By identifying the important features and distinguishing them from merely colorful add-ons, customers become more aware of their own needs. When the customer is integral part of the development team, as required by extreme programming, this closer view to the requirements and their relative weight lets both the customer and the developers focus on those parts that will bring most utility. This prevents wasting time, money, and other resources on unimportant options, which have been bundled into the initial requirements by somebody without good comprehension of the needs and costs, just for a good measure.

10.2. Cost of change

Even if change in requirements is considered part of the software project development and deployment, developers still face the traditional wisdom that the cost of fixing a problem (or generally change in the software) rises exponentially with time. Fortunately, this observation only holds with traditional approaches, where the change is considered inferior and all effort is put into restricting chance of it happening. This is a chicken and egg problem because presumed cost of late change is taken as input premise and further reduces flexibility.

Extreme programming proposes and supports changes by simplicity of design with clear interfaces, preferably with the help of object-oriented technology. Any interface that is included in the system has to have immediate use. That decreases the number of situations when complex structures and relations are designed, based on the initial requirements and analysis, but the requirements change before the project makes any real use of them. Extreme programming

bets on people not knowing everything and learning as the project evolves, and promotes the possibility of finding out simpler ways or new uses for existing solutions at any time, not only during analysis and design.

An additional aspect of extreme programming, which contributes towards change not being a disaster for software project, is the use of a comprehensive sets of automated test, that makes it easy to verify whether a change breaks the overall functionality. If the functionality at all levels of complexity, both internal interfaces among functions and methods and external on the application level, is covered by tests, the tests have become part of the system specification and if they pass, the system functionality was not corrupted.

Last but not least, the final factor is the exact reverse of the traditional assumption about exponential rise of cost of change over time. The experience with changes and the recognition that change is allowed and cheap makes changes easier because psychology does not play against the programmers and does not tie them to the existing state of the system and to previous decisions. On the contrary, it brings the freedom of making adjustments whenever they contribute to the overall quality and well-being of the project.

10.3. Extreme programming and Web projects

In the previous chapters, some important aspects of the Web-based applications and their development have been described; in this chapter, software development methodology called extreme programming was summarized. From these overviews, clear matching of features of applications based on the Web technology (and their development) with the extreme programming ideas can be deduced.

Instant upgrades of Web-based applications, where there is only a single server side with application logic, correlate with short development engineering tasks and iterations and with shortened release cycles. When the distribution channels only lead from developers to single or relatively small number of servers, it makes sense to ask for the new features and bug fixes to be deployed on the production sites as soon as possible. If the developers of Web-based applications can deliver changes in progressive steps, they can go on-line in continuous *floating releases* that do not need to be explicitly packaged or distributed. Distribution then becomes part of the development process.

Extreme programming supports this gradual development of the whole project, from the most important features to the less important ones, with changes in priority queue not being an issue because they are not seen by the developers until they fetch next engineering task from the queue. A comprehensive and constantly extended test suite ensures that a new feature or change does not break previous code and does not cause a fatal failure when the code is integrated into the on-line system.

Writing test suites for interactive applications is definitely harder than testing batch programs or individual functions. With interactive environment, too many external conditions can occur on the input — users' behavior and ability to do a sequence of actions that was never expected is one example. Fortunately, the processing schema of Web-based application, the two-stage request and response model, makes it possible to view these applications as simple functions. For a given set of input parameters they produce expected changes in the internal state of the information system, and return output in HTML or other well-defined format. Presence of clear, text-based interfaces, enables testing of individual stages of the application's work as seen from outside. Internal functions and methods can reasonably be tested using traditional approaches.

Subsequent chapters will focus on content and presentation separation, which should further reduce the complexity of testing Web-based application, even in situations where the presentation requirements change considerably.

IS MU Fact

The first IS MU application, a publication database, was handed over to users three months after the decision about its development was made, and three weeks after the actual development and programming work started.

As the deployment of new information system in the Web environment concerns mainly (or only) the server side, it is possible to put the Web-based system into production as soon as the first feature required by the customer is in place — the time needed may be merely a couple of days or weeks, depending on the technical infrastructure that needs to be in place before the system will have some sense.

Installing and setting up the server hardware, operation system, network, fail-over and backup facilities and procedures, as well as the database system, Web server and necessary production run-time environment for the application code will certainly take some time. But in parallel with these preparations, developers can focus on the highest priority feature and bring it on-line as soon as it passes tests provided by the customer and by the team itself.

IS MU Fact

The first users of IS MU were teachers, filling up their publication records, followed by faculty management preparing the Catalogue of course offerings. Students only came to pre-register about three month later.

The following updates will continuously be added to the live system and new groups of target users will be able to start using them gradually, without any need for the Big Bang type of switch.

That eliminates the costs related to switching over everyone in the institution to the new system at one moment.

Dependence on external conditions, with new Web browsers coming to use without any control by the Web-based system developers, can effectively be tackled with the extreme programming's short engineering tasks. If an incompatibility arises with upgrade on client side, preventing large number of users to use the system or use it efficiently, the task can be put on top of the priority queue and will be handled in a timely fashion, without disturbing the development process or bringing in the change that would be hard to manage.

Having clear communication channels that allow all users to comment on the quality and overall utility of the system, as well as minor details that make them angry, is possible in the era of computer networks and email. Users can avoid long hierarchical structure for reporting, and send feedback directly to the developers. First hand description of the problem, together with exact parameters and actions that caused it, is an invaluable source of knowledge. The problem report can be used as test case which ensures that once fixed, it will not appear again. Or the issue reported is not a bug in the software — in may be a plea for new feature or extended variations in behavior of existing ones. With extreme programming approach, they can be picked up by the programmers based on the agreed-on priority and the change or bug fix may go on-line within minutes. That of course does not mean that all requests will be implemented within minutes or even within weeks — that would suggest that the capacity of the team became infinite. But the feedback is always considered and if the problem is verified or the need of a new thing or change is acknowledged, users can be certain that their requirements will be met in some way eventually.

From developers' point of view, it is important that they can start working on the task that lies at the top of priority queue at once, without waiting for a new semiannual meeting being held to resolve changes in requirements, analysis, and design. The rapid development of production code shortens the turnaround time in the communication, and users will be able to confirm that the bug is fixed or new feature added when they still have fresh memory of their original report.

Overall, the rapid, progressive development of the system using techniques of extreme programming matches well with the Web-based nature of centralized development and maintenance with minimized length of distribution channels. Even if the developers will not implement for various reasons all requirements of extreme programming and their work on the Web-based information system will not be extreme in this sense, individual parts of the methodology may still be utilized to develop and drive the system efficiently.

Part III

The Solution: Intermediary Data Layer



Chapter 11. Application data in native data structures

In the Part I of this work, an analysis of the existing approaches to the development of Web applications revealed problems which all stem from the fact that

currently, the individual Web applications are expected to return a functional HTML output.

With all the common approaches, the HTML markup is being included within the application code or the code of supporting libraries, leading to a complicated mix of syntaxes. Consequently, individual members of the development team are forced to work with source files which include languages and markups foreign to them. The HTML-based applications are hard to reuse for other target formats. Since the presentation is hard-coded part of the application, regression test suites cannot efficiently check the core functionality of the applications, as the first point where the output of applications can be grabbed and inspected is at the HTML level. Thus, any change in the presentation affects tests verifying business logic, and vice versa. In addition, generating HTML from procedural languages often leads to non-valid HTML documents because with a more complex nonlinear application code it is very easy to lose track with the intended format of the target markup.

In order to make Web applications reusable, maintainable, and easily testable, it is essential to weaken the HTML dependency. The first step in moving away from Web applications tied to HTML markup is to

remove the output markup and its handling from the code of individual applications.

With all types of dynamic Web applications, be it classical CGI programs that use print-style output, handlers that pass output directly to the surrounding environment using functional API, or both kinds of templating techniques, the generated output markup is a reflection of application's operation and data that it produces. The markup is determined by the following inputs:

- Processing of input parameters.
- Results of operations that the business logic code conducts on backend data.
- The data which was directly selected from the database backend.
- The code that further tunes the result based on the above conditions.

Abandoning the traditional requirement that the Web application has to provide the output in the target markup which could be directly used as an HTTP response, all parts of the application that deal with the production of HTML can be removed from the code of all applications in the

Web system. As it is evident from Figure 11.1, only the core operations remain after this reduction, compared with the original schema in Figure 4.1.

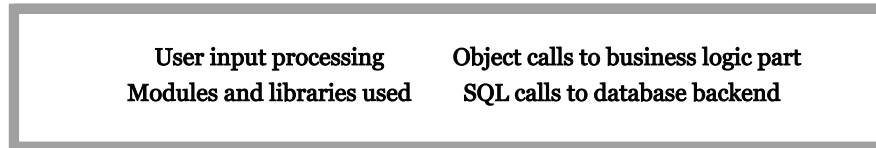


Figure 11.1. Core of a Web application.

However, the application output still needs to be gathered and translated to the HyperText Markup Language. It may not be necessary for all applications and all clients, but there still has to be an option to convert the output to this format. As shown in the previous chapters, the Web applications are highly dependent on external conditions on the client side and must follow trends in the browser technology and act accordingly. Since HTML (or its successor, XHTML in version 1.0) is still the most compatible format, the one that is most widely supported by majority of Web clients and for many of them the only hypertext format, the option of output in HTML must be preserved. Thus, even if the HTML handling parts will no longer be present in the source code of the individual Web applications, data of those applications must be properly handled and converted to HTML in post-processing stage in the external environment — application server or Web server.

While it is feasible to provide a direct converter of the application data to HTML, it would also mean that such a conversion code would need to be repeated for each new programming language and each new output format. For example, Perl converter would be able to do the HTML conversion of data returned by applications written in Perl, nevertheless, a Python-to-HTML converter would be necessary for Python data, and this number would again double once a new output format like WML was required. For m programming environments and n output formats, there would be $m \times n$ combinations and the same number of separate converters needed. Having maintainability and extensibility in mind,

an intermediary data format that would allow a seamless representation of the application data is called for.

Portable serialized data in intermediary format can be post-processed and converted to the desired target format using generic converters. By having a common format in the middle of the operation, the total number of serializers and converters drops²⁾ to $m + n$ for the above specified numbers of programming languages and output formats. In addition, as the interme-

²⁾ For only one language and a single output format, two tools will be needed. While not saving in terms of numbers, the internal interface will still prove to be an advantage.

diary format will hold information about data and not about presentation expectations, it will be possible to share the presentation transformations of the same or similar data by many applications.

In the Web environment, most formats are based on SGML, Standard Generalized Markup Language [Goldfarb91], and on its subset, Extensible Markup Language (XML) [Bray00]. Since most of the end formats to which the output of Web applications will be converted can be expected to be in the Web domain, using XML for intermediary data representation promises a trouble-free integration into the overall setup of a Web-based system. In addition, XML features important benefits:

- XML has an open definition, the language is non-proprietary and widely accepted.
- Its content model is not restricted, it is well suited for arbitrary data structures.
- Unlike HTML, XML aims at semantic markup of the data, not at presentation. Therefore, it fits better the serialization purpose where the data content is expected to be presented in various formats.
- There are many tools available supporting the conversion to other formats and markups through well-defined and open interfaces.
- Latest generations of Web browsers support XML directly. Hence, with increasing numbers of clients, no further postprocessing will be needed and the intermediary format will be the target one.

Here, the question arises whether the conversion to the standard intermediate format, XML, should be a part of the application code of individual applications?

Changing code of current Web applications to print its output data in XML instead of the HTML markup would certainly bring only a marginal advantage. If Perl code producing HTML fragments using textual output

```
print "<ul>\n";
for my $row (@data) {
    print "<li>$row</li>\n";
}
print "</ul>\n";
```

is merely replaced by its XML-ized counterpart

```
print "<product_features>\n";
for my $row (@data) {
    print "<feature>$row</feature>\n";
}
print "</product_features>\n";
```

the result will inherently pose the same problems as the HTML-bound applications.

The escaping and markup problem of multiple syntaxes analyzed in Part I of this work is still present in such an approach, as the data can contain characters or strings that have escaping and markup meaning in XML, thus changing the intended semantics of the output. It would be necessary to sanitize each value which needs to be included in the output document, using textual manipulations or utility functions:

```
print "<product_features>\n";
for my $row (@data) {
    print '<feature>', sanitize_for_xml($row), "</feature>\n";
}
print "</product_features>\n";
```

This will in turn increase back the complexity of the application code and call for omissions and errors, since all occurrences of the data would need to be handled. Many existing projects do not HTML-sanitize its data, assuming that it will never contain characters < and &. When the assumption is eventually proved wrong, debugging becomes a time-consuming work. Keeping the output XML document well-formed and in sync with its document type definition would be yet another tedious and error-prone task.

Therefore,

the Web applications should return data in their **native data structures**.

Under this assumption, the code of individual Web applications does not need to include any functionality that would provide the transformation of native (Perl, PHP, Java) data structures to their textual representation in XML, neither directly by code with literal tags of the elements, nor indirectly by calling utility functions or methods which would achieve the conversion. All the necessary manipulation can be done in the postprocessing stage on the data returned by applications. Thus, the conversion and serialization task is shifted from being present in each application in the Web system to the surrounding environment of the application server or the Web server, from which the applications are invoked, as shown in Figure 11.2.

Following-up with the above code that returns a list of data from Perl array @data, the application code will use an anonymous associative array (hash) and a regular array to create a nested data structure with named data elements, instead of producing text:

```
return {
    'product_features' => [ @data ],
};
```

This way, the application code is greatly simplified. Most common scripting languages provide some functionality for dynamically naming elements of data structures by string names and for creating nested structures. Should the index-by-string associative array type be unavailable in certain programming environments, it can be replaced by a simple array with paired values, denoting key and its value.

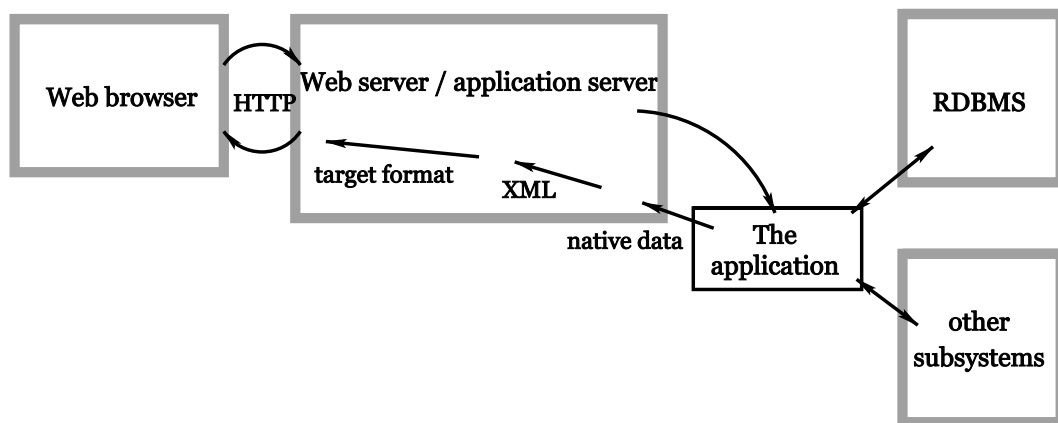


Figure 11.2. Web application with native data output.

The interface definition of the Web applications stays the same in the input part, as the application code receives data of the incoming HTTP request and additional parameters and possibly interface to the application or the Web server. On the output side, however, it is not an HTML page that is sent to output, it is the core data of the output document, returned in the native data structure of the programming language or environment in use. Such a change brings numerous advantages:

- The application programmer can focus on the business logic of the application, not on the HTML presentation markup.
- No escaping or markup collisions need to be handled because the output data of the application is in native format, not serialized to text.
- The flow control of the application can be more natural, it no longer depends on the order in which the output page is being built.
- The output can be transformed not only to HTML, but to any output format easily, making the application code reusable.
- The application code is smaller, increasing readability and maintainability.

Once the native data structure was passed to the calling layer, application server or Web server, it will be serialized by generic tool to XML format. With latest versions of Web browses, this format can be used directly on the output. For older browsers, the postprocessing to HTML can be done on the server; for mobile devices, either WML or XHTML can be chosen as the end target format. The intermediary data format also makes it possible to reuse the single application code to produce data for other purposes than interactive Web work, i.e. comma-separated values (CSV) lists for machine processing, paged typesetted output for high-quality printing, or even conversion to proprietary formats.

The next chapter focuses on the XML serialization of native data.

Chapter 12. Data matched against formal structure

12.1. Data serialization

Once the application returns its output data in native data structure, the following step is serializing it into intermediary data format, to which generic converters to target markups could be applied. Applications can be written in any language that the developers find most suitable for a particular task or even using a mixture of languages, as long as there is a converter to one common format — XML — for the programming language in question. The serializer will have to be written for each programming language used, since it deals with data structures specific to the language in question. On the other hand, the serializer is application-independent.

There are many ways data structures can be serialized to XML — only for Perl structures, at least modules `XML::Simple`, `XML::Writer`, and `XML::Generator::PerlData` can be found. However, they all are driven by the input data structures, by the data they are serializing. This is certainly beneficial when the serialized data is stored on disk and later reused by the same application, since the application itself determines its own data format. Nonetheless, in the Web system where the data will be further post-processed, such a flexibility and freedom is not desirable. Should the application programmer by mistake or in good faith rename one of the fields in the data which the Web application returns, the resulting XML would be different, which would likely lead to a failed or incomplete conversion to the target format in the end.

In addition, the data structure would only be known once the application is run, not beforehand, and with no guarantee that the format will not change with different input parameters being passed to the application for the next invocation. Such a volatile environment would make the development of the postprocessing stages unnecessarily harder, providing no fixed point that could be built upon. Therefore,

the structure of the data hold by the intermediary XML should not be dependent on the application code.

Quite the opposite, the application code has to produce native data structures that will be matching pre-defined data structure. As the application code doesn't provide any declarative information about the data, building it dynamically in run-time, an external description of the data structure is necessary. Such a data structure description can be used both for the application development because the application programmer will have a fixed target that the application has to match, and for the development of the postprocessing stage because it will be known that application's output will obey the given structure. The data structure description can serve as a specification of the allowed input parameters as well, holding the complete information about the interface of the application code.

Existing mechanisms used for describing the structure of XML document are targeted at validation of the content:

- Document Type Definition (DTD);
- XML Schema [Brown01];
- RELAX NG [Clark01].

They provide means of specifying the allowed element and attribute names, the order and nesting in which they can appear in the XML document. Validating XML parsers then can utilize this information to *validate* that the XML stream is well-defined, matching the required structure. Another possible use is for retrieving default values and substituting them for those not specified in the XML data.

Nonetheless, the validating nature of these tools assumes that the XML data stream or XML document already exists. Their features make them less suitable for communicating the information for the purpose of *creating* the XML:

- DTD is not an XML document.
- The description using DTD nor XML Schema is not a document with nested content that would be resembling the structure of the XML that it describes.
- As the DTD nor XML Schema document does *not look like* the document that it describes, it is hard to use it to communicate the information about the document to people who are not fluent in these validating mechanisms, i.e. application programmer or Web designer.
- RELAX NG is still rather verbose when it comes to expressing even the simplest data structure.³⁾
- XML Schema and RELAX NG are not compatible with DTD, as their validation scope is much broader.

In addition, as the validation only concerns one XML document, the description of the output data using these validation tools could not contain formalized specification of input parameters as well. Having allowed input parameters of applications specified in declarative form is an important feature for larger systems, since it not only allows programmers of individual applications to know the interface, but it also makes possible to verify the correctness of post-processing transformation. A hypertext link or form action leading to a particular Web application should carry only specified parameters, and it ought to be feasible to test for this earlier than by running the application and observing errors it returns.

³⁾ Originally this work was leading towards using RELAX NG. However, application programmers found its syntax to be unnecessarily verbose for the task at hand.

For the above reasons, using a generic solution did not fit the intended goal and a specific lightweight data structure description format was chosen instead in order to hold information about application interface. For the design of this format, the following characteristics requirements and constraints were considered as the primary ones:

- The data structure description must be an XML document. This requirement is important as not to broaden the syntax domain that is used in the development of the Web system.
- It must resemble the data it describes, the structure of this specification must look like the resulting document.
- It should allow both DTD and XML Schema to be generated, so that standard tools can be used to validate the output XML by third parties.
- The description has to hold information both about the input parameters of the application and about the output data, in one source.
- The format must be usable primarily for creating valid XML document from application data. It should also allow for checking that the data matches the requested structure.
- The format has to support most common data constructs of current scripting languages — scalar values, arrays indexed by integers, and associative arrays (hashes) indexed by strings, and their combination and nesting. This is important for cases when Web interface is being created for existing applications where the output data structures are already set.
- The format has to allow most XML constructs for cases where the expected XML document is specified as the primary requirement.
- The instances describing individual applications have to be concise.

Based on these design criteria, *data structure description* (DSD) format was defined.

12.2. Overview of the data structure description format

1. The data structure description defines permitted input parameters that the application accepts and the output XML document, to which application's output will be converted from native data structures. It is a well-formed XML document.
2. Elements in this description document define input parameters of the application, output elements that will appear in the output document unchanged and their structure, and placeholders that will be replaced with data returned by the application. Attributes of these elements may specify types that the input parameters or output data can hold, or state conditions under which elements will appear in output.
3. The `_param` elements denote input parameters. The name of the parameter is specified by the attribute name. Alternatively, a generic set of parameters with common prefix can be defined with an attribute `prefix`. Multiple occurrence of the same parameter can be allowed using attribute `multiple`. Parameter elements can appear in any place of the data structure (inside of root element) and are removed before output serialization is started.

4. As the next step, content of leaf elements with attribute `typeref` is set to local or remote fragment of data structure description, identified by unique `id` attribute; root element is used if URL with no fragment information is included. The fragment may be a data placeholder, but it can also be a regular part of the document. The next processing runs on XML document with removed parameter elements and expanded `typerefs`.
5. During serialization, the data structure description document is processed starting with the root element and following recursively with its descendants.
6. Four categories of elements serve as placeholders which will be replaced with data returned by the application code: elements that have no children (leaves), elements named `_data`, elements that have attribute `type` with one of the built-in types, and elements with attribute `multiple` denoting an aggregate structure. The name of the element will be used as a key to lookup the data from the associative array returned by the application, except for the element `_data` in which case value of its attribute `name` is used as the key. Children of such placeholder elements specify placeholders for nested data of the particular data value.
7. Other elements will be copied to output unchanged, unless conditional attributes (`if`, `ifdef`) that act according to returned top-level values specify that they should be skipped.
8. The result of this process is the XML document with intermediary data of the application serialized in portable format.

12.3. Examples of data structure description document

Example 12.1. Data structure description with a single scalar placeholder.

```
<?xml version="1.0"?>
<data>
  <fullname/>
</data>
```

A very simple data structure description is shown in Example 12.1. The root element `data` will appear in output XML without any change. Its child, element `fullname`, is a leaf element (does not have any children elements), therefore, it is a placeholder for application data. If an associative array

```
return { 'fullname' => 'Sally Albright', };
```

is returned by a Perl application, the output XML document after serialization will be

```
<?xml version="1.0"?>
<data>
  <fullname>Sally Albright</fullname>
</data>
```

The value of the key `fullname` gets binded as the text content of the placeholder element `fullname`. Note that the application did not need to specify in the returning associative array

that the element `fullname` is a child of element `data` — the placeholder corresponding to the value in the hash was located automatically. In fact, it would be an error for the application to return a value with key `data` which is not specified as a placeholder in the data structure description, as shown in the Example 12.2 which lists permitted and forbidden return values for the data structure description in the Example 12.1.

Example 12.2. Output XML for various return data structures.

return data	output XML
<pre>return { 'fullname' => 'Sally Albright', };</pre>	<pre><?xml version="1.0"?> <data> <fullname>Sally Albright</fullname> </data></pre>
<pre>return { 'fullname' => '', };</pre>	<pre><?xml version="1.0"?> <data> <fullname></fullname> </data></pre>
<pre>return {};</pre> # empty hash, no fullname	<pre><?xml version="1.0"?> <data/></pre>
<pre>return { 'fullname' => 24, };</pre>	<pre><?xml version="1.0"?> <data> <fullname>24</fullname> </data></pre>
<pre>return { 'name' => 'Sally Albright', };</pre>	Error, no placeholder named <code>name</code> was specified in the data structure description.
<pre>return { 'fullname' => 'Sally Albright', 'age' => 24, };</pre>	Error, as no placeholder <code>age</code> was specified.
<pre>return { 'data' => { 'fullname' => 'Sally Albright', }, };</pre>	Error, no placeholder named <code>data</code> was specified; the root element of the DSD is not a placeholder.
<pre>return { 'fullname' => { 'first' => 'Sally', 'last' => 'Albright', }, };</pre>	Error, the <code>fullname</code> value has to be a scalar, not a hash.

The listing also demonstrates that a missing value results in the placeholder element being removed from the output XML, while a value of empty string leads to an element with no content. Individual implementations are free to serialize the empty element either as `<fullname/>` or as `<fullname></fullname>`, in any case empty string ensures that it will be present.

In the DSD in Example 12.1, no input parameters were specified, therefore no input values will be passed to the application. The application may be returning constant value, or it may act according to other environment information — returned `fullname` may be the name of the salesperson of the month, or it may be the name of the user in an authenticated information system.

The simplest data structure description is presented in Example 12.3. The root element is a leaf, therefore it is a placeholder. In this case, an exception from the handling of missing values applies — the root element will always be present in the output to make it well-formed XML, even if the application did not provide its value. The Example 12.4 shows how various output values are processed.

Example 12.3. Data structure description with only root element.

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

Example 12.4. Output XML for single root placeholder.

return data	output XML
<pre>return { 'fullname' => 'Sally Albright', };</pre>	<pre><?xml version="1.0"?> <fullname>Sally Albright</fullname></pre>
<pre>return { 'fullname' => '', };</pre>	<pre><?xml version="1.0"?> <fullname/></pre>
<pre>return {}; # empty hash</pre>	<pre><?xml version="1.0"?> <fullname/></pre>
<pre>return { 'name' => 'Sally Albright', };</pre>	Error, no placeholder named name was specified in the data structure description.

A slightly more complex Example 12.5 shows the data structure definition for a complete application that will allow users to search in a database of people in an information system. The user can enter a name and either get a list of matching records or detailed information about

one person if the search yielded only one result. From the list, the user can get detailed information about each person, presumably by clicking on the hyperlinked name on a Web page. Thus, the application will accept a search parameter with a search pattern, or a numeric identification used to address the exact record from the list. The output will consist either of data structure about the list of people or about details of a single person.

Example 12.5. Data structure description for a search application.

```
<?xml version="1.0"?>
<findpeople>
  <people multiple="listelement">
    <_param name="q" />
    <_param name="submit" />
    <person>
      <id type="int" />
      <fullname />
      <affiliation multiple="list" id="affil">
        <id type="int" />
        <deptname />
        <function />
      </affiliation>
    </person>
  </people>
  <person ifdef="id">
    <_param name="id" type="int" />
    <id type="int" />
    <fullname type="struct" id="person_name">
      <first mandatory="yes"/>
      <middle/>
      <last mandatory="yes"/>
      <title/>
    </fullname>
    <affiliation typeref="#affil" />
    <studies multiple="list">
      <id type="int" />
      <deptid type="int" />
      <_data name="deptname" />
      <programid type="int" />
      <programname />
    </studies>
  </person>
</findpeople>
```

The root element is `findpeople` and it will be the root element in the output XML document.

Element `people` specifies a placeholder which will be filled with array data, denoted by attribute `multiple`. The value `listelement` of this element says that the element `people` will not be repeated in output; instead, its sole child `people` will be holding individual values of the array. Three children of the element `person` then serve as placeholders for individual values of members of `people` array. The `id` element mandates that returned value has to be of type integer. Element `fullname` does not place any restrictions on the data type and as it is a leaf element,

it accepts any scalar value. The `affiliation` specifies an array value with its `multiple` attribute, and each of the rows of the array are to be three additional scalar values.

Element `person`, the second child of the root element, will only be present in the output document if value with key `id` is defined by the application, due to the attribute `ifdef`. It has four placeholder elements: scalar integer `id`, a hash/structure `fullname`, `affiliation` with an array type included using the `affil` fragment, and another array, `studies`.

In the data structure description in Example 12.5, there are also three parameter elements: `q`, `id`, and `submit`. The first two are meant for the search string and for the numeric identification. The third one will likely be ignored by the application, since the application only provides single functionality, the search. However, as some value is likely to come from a "Search" button in the HTML form, it is necessary to include it in the DSD in order for the interface to be complete. The `_param` elements can be present in any place in the document, their location does not hold any significance. Here they are shown in those parts where they logically belong.

Example 12.6. Returned list of people for the search application.

```
return {
  'people' => [
    {
      'fullname' => 'Rebeca Milton',
      'id' => 25132,
    },
    {
      'fullname' => 'Amanda Reese',
      'id' => 63423,
      'affiliation' => [
        [ 1323, 'Department of Medieval History' ],
      ],
    },
    {
      'fullname' => "John O'Reilly",
      'id' => 1883,
      'affiliation' => [
        [ 2534, 'Department of Chemistry' ],
        [ 15, 'Microbiology Institute' ],
      ],
    },
  ],
};
```

The Example 12.6 shows a potential output data that the application might return when the user searches for all the people in the database with names starting with "Re". When serialized according to the data structure description, the output XML document in Example 12.7 will be created.

Example 12.7. Output XML for list of people.

```

<?xml version="1.0"?>
<findpeople>
  <people>
    <person>
      <id>25132</id>
      <fullname>Rebeca Milton</fullname>
    </person>
    <person>
      <id>63423</id>
      <fullname>Amanda Reese</fullname>
      <affiliation>
        <id>1323</id>
        <deptname>Department of Medieval History</deptname>
      </affiliation>
    </person>
    <person>
      <id>1883</id>
      <fullname>John O'Reilly</fullname>
      <affiliation>
        <id>2534</id>
        <deptname>Department of Chemistry</deptname>
      </affiliation>
      <affiliation>
        <id>15</id>
        <deptname>Microbiology Institute</deptname>
      </affiliation>
    </person>
  </people>
</findpeople>

```

The values of an array referenced by `people` are expanded to elements named `person` due to the `multiple="listelement"` specification. Each value of the array is an anonymous associative array and its elements are binded to children placeholder elements of `person` using their names as keys: `id`, `fullname`, and `affiliation`. As Rebeca Milton does not have any affiliation in the data, this element will not show in output in her case. The affiliations will be expanded in the same place where they were defined, because the value of its `multiple` attribute is `list`, as opposed to `listelement`. Mr. O'Reilly has two affiliation records and they will come out next to each other, at the same level as his `id` and `fullname` elements.

The values of the `affiliation` array are interesting: associative arrays with keys `id`, `deptname`, and `function` could be expected. However, in any place where a hash is expected, an array may be used as instead. Its values will be binded in the natural order to corresponding placeholder elements. Of course, an associative array is also legal value. The Example 12.8 demonstrates different data structure used for the record of Amanda Reese, with identical meaning. Using unnamed arrays may yield performance benefits, especially when used on large data sets. Using associative arrays ensures better compatibility because the keys have to match placeholder names and mismatch is more easily detected.

Example 12.8. Associative array is equivalent to regular array.

```

[
    63423,
    'Amanda Reese',
    [
        {
            'id' => 1323,
            'deptname' => 'Department of Medieval History',
        },
    ],
],

```

Also noteworthy is the fact that the function value is missing in all cases — the values of affiliation array are two-element anonymous arrays, with the third value missing. This is equivalent to a missing value from associative array. As the placeholder does not have a mandatory attribute which would enforce the value to be present in the returned data and make it an error for the value to be missing, the function element will simply not be present in the output XML. Of course, having more than three values in the anonymous array in the place where only three are expected would lead to an error, as would having bad hash key name.

No other elements will be present in the output XML document for data in Example 12.6 because the element `person` has an `ifdef` condition which mandates that this element and its children will be present in the output XML only when value with key `id` is defined by the application. This mechanism makes it easy for the application to handle more than one action and to include more than one tree structure in its data structure description, while not having the output document polluted with stubs of branches that were not activated for the particular execution.

Should the application return data for a single person as in Example 12.9, without specifying any `people` value, the `people` placeholder and its structure would not be present in the result, instead, the structure inside of element `person` would be populated in the generated XML (Example 12.10). The `fullname` placeholder expects a data structure as opposed to the scalar from the list of people. In the list of people, `fullname` is not a top-level placeholder, so there is no type conflict. A structure placeholder can receive either associative array or regular array as its bind value. In this example, an array is used to demonstrate how an undefined value can be used to skip the middle element, when the last value still has to be present. The `mandatory="yes"` attribute makes the last name value mandatory and it would be an error if it was not returned by the application. However, as the `fullname` is not marked mandatory as well, it could be missing completely and no breach of the interface would occur. Mr. O'Reilly is a professor and does not have any study records in the database, that is why the element `studies` is not expanded in the output.

Example 12.9. Single person result for the search application.

```
return {
  'fullname' => [ 'John', undef, "O'Reilly", 'Prof.' ],
  'id' => 1883,
  'affiliation' => [
    {
      'id' => 2534,
      'deptname' => 'Department of Chemistry',
      'function' => 'Head of department',
    },
    {
      'id' => 15,
      'deptname' => 'Microbiology Institute',
      'function' => 'Tutor',
    },
  ],
};
```

Example 12.10. Output XML for a single-person result.

```
<?xml version="1.0"?>
<findpeople>
  <person>
    <id>1883</id>
    <fullname>
      <first>John</first>
      <last>O'Reilly</last>
      <title>Prof.</title>
    </fullname>
    <affiliation>
      <id>2534</id>
      <deptname>Department of Chemistry</deptname>
      <function>Head of department</function>
    </affiliation>
    <affiliation>
      <id>15</id>
      <deptname>Microbiology Institute</deptname>
      <function>Tutor</function>
    </affiliation>
  </person>
</findpeople>
```

If neither people nor id value was returned by the application, the output would consist of empty root element only. If the returned hash included other keys than one of the five top-level placeholders (people, id, fullname, affiliation, studies), an error would be signaled because no other values are specified in the output interface. Whether such invalid return data leads to a warning or fatal error depends on the setting of the serializer. The output document can be generated in a best-effort way, or the serialization conflict will be reported and the process aborted. Which of these two settings are to be used depends on the nature of the Web system, the output format, and on the state (development or production) of the application.

The error handling can be set independently for the input parameters as well. For the above data structure description, only parameters `id`, `q`, and `submit` are allowed to reach this application. Examples of valid requests include HTTP GET requests with query strings

```
?id=42
```

```
?q=Re*&submit=+Hledat+
```

```
?q=&submit=
```

Examples of invalid requests are

```
?id=delete
```

where the type of the parameter `id` is not integer as mandated by its `type` attribute in the data structure description,

```
?name=Amanda&style=fast
```

where the parameter names do not match at all or

```
?id=1883;id=2900
```

since `id` parameter does not have attribute `multiple` set to `yes` which would allow it to receive multiple values.

Strict types of input parameters (`integer`, `number`) are only to be used for parameters that come from machine generated sources, for example hypertext links or values of checkboxes in HTML forms, never for values that are accessible and editable by the user. If the value of the parameter comes from a textfield that the user filled, the type has to be specified as the (default) generic string, even if the user was asked to enter number of their siblings. Having string value entered to a field where integer value was expected is not a breach of interface and has to be handled gracefully, by the application code. On the other hand, if the integer parameter value always comes from sources that were not directly editable by the end user, it should be declared with stricter type, as such declaration provides better means of testing the whole setup and provides guarantees to the application that the value will always be of the given type.

Example 12.11. Associative array naming with values.

```
return {
  'p' => {
    63423 => 'Amanda Reese',
    1883 => 'John O'Reilly',
    25132 => 'Rebeca Milton',
  },
};
```

So far the examples were showing how associative arrays are used to name elements, fields of structures. However, in scripting languages they are often used to address data with other values, as demonstrated in Example 12.11. For this kind of data structure, attribute `multiple` with value `hash` can be used. The values of the hash will be binded as the content of repeated element. The keys will be stored in an attribute named `id`, unless the placeholder specified different name using attribute `idattr`. Similar to `multiple`'s value `listelement`, `hashelement` with analogous semantic is also available. Some implementations of associative arrays have the keys ordered, in others the order is random. The data structure description can specify ordering using `hashorder` attribute. See Example 12.12 for ways of describing the data structure from Example 12.11 and results that this description generates after serialization.

Example 12.12. Serializing associative array.

data structure description	output XML
<pre><?xml version="1.0"?> <data> <p multiple="hash"/> </data></pre>	<pre><?xml version="1.0"?> <data> <p id="63423">Amanda Reese</p> <p id="25132">Rebeca Milton</p> <p id="1883">John O'Reilly</p> </data></pre>
<pre><?xml version="1.0"?> <data> <p multiple="hash" idattr="m"/> </data></pre>	<pre><?xml version="1.0"?> <data> <p m="63423">Amanda Reese</p> <p m="25132">Rebeca Milton</p> <p m="1883">John O'Reilly</p> </data></pre>
<pre><?xml version="1.0"?> <data> <p multiple="hashelement" idattr="m"> <q/> </p> </data></pre>	<pre><?xml version="1.0"?> <data> <p> <q m="63423">Amanda Reese</q> <q m="25132">Rebeca Milton</q> <q m="1883">John O'Reilly</q> </p> </data></pre>
<pre><?xml version="1.0"?> <data> <p multiple="hash" hashorder='num' /> </data></pre>	<pre><?xml version="1.0"?> <data> <p id="1883">John O'Reilly</p> <p id="25132">Rebeca Milton</p> <p id="63423">Amanda Reese</p> </data></pre>

12.4. Operation of data-centric applications

The Figure 12.1 shows the role of the data structure description in the operation of the data-centric application. The DSD is first consulted in the input stage when the parameters are checked against parameter specification. Then the application code is invoked and when it returns its data, the data structure description is used again, this time to validate the structure of the data and serialize it into an independent format, the XML. Thus, the data structure description is in the center of the Web application operation.

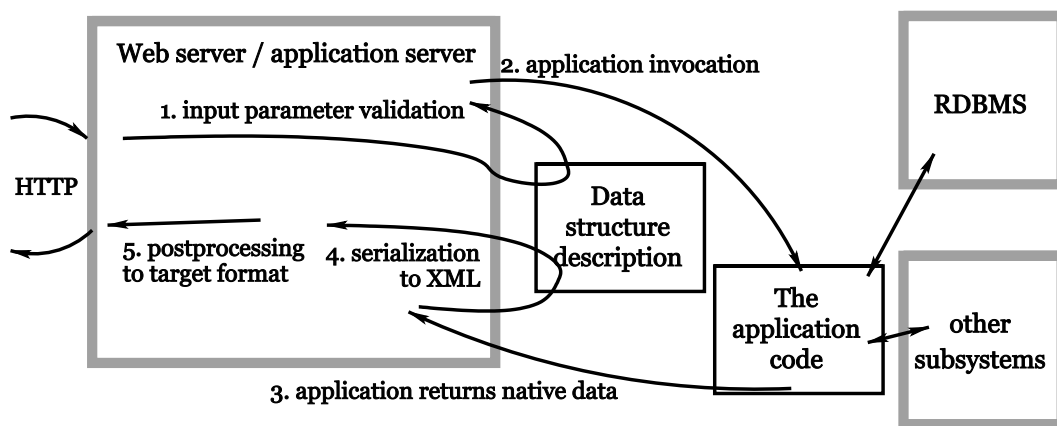


Figure 12.1. DSD-based Web application.

A reference of data structure description format with the complete list of functional elements and attributes can be found in the Chapter 13.

12.5. Design notes

The data structure description was designed to achieve a strong separation of the Web application code from its presentation. The application code is stored in one source, while information about its interface, input parameters and output data, is stored in a separate file. Yet another source is used to hold post-processing instructions. Thus, to write an application in this data-centric framework means to create at least three different source files. Many programmers are opposed to the idea that the whole application cannot be stored in one file. However, the target of this work is not a project consisting of only a few scripts or servlets. Those will be better served by traditional approaches analyzed in the first part of this work, applications written in a single source with a mixture of syntaxes that can be authored in a couple of minutes.

The target users of the data-centric Web application framework are developers of projects with hundreds of applications. Here, it is not the time spent building the first application which matters. The more important is the effort and time spent to maintain the first application after three years of operation, to update it with a new user interface or with a new feature without

breaking it or any other application that depends on it, or the effort and time spent adding and integrating application number 374. For these kinds of project, data structure description provides a concise, declarative description of application's interface which can be used by all parties in the development team.

The format is declarative with no attempt to bring in procedural or functional features, as those are readily provided by the application's programming environment. That is why conditional attributes only check the single value in the returned application data, not supporting any expressions or access to nested structures. If a more complex test is necessary, it is best handled in the application code which can compute the required scalar value and return it together with the rest of its data. In the serialization stage, only the simple test for presence of true or defined value is needed.


For the same reasons, the input parameter specification using `_param` elements only defines a list of parameter names and prefixes, with a minimalistic support for checking numeric values. Again, it is the task of the application to verify that all other complex internal conditions are fulfilled before it can act according to the input parameters. If a discrepancy is encountered, for example user submits a request for entering new record but does not supply value of a mandatory field, it is the application logic which must decide how the request should be handled and what error messages should be reported. The same reasoning holds for support of default values being omitted from the specification.

The design of the data structure description is strictly on the minimalistic, declarative side, supporting well the intended purpose of the format. Most of the existing templating methodologies started with only a handful of features and gradually added more and more functionality (test, loops, database access), often in unstructured manner. The data structure description is the interface layer for projects where programmers use programming languages and Web designers use presentation tools. These are the parts where the operations with data and text take place. Not in the declarative data specification.

No programming methodology is forced on the application programmers by the intermediary data layer in XML and data structure description that is used to drive the serialization. The application can be written using object-oriented paradigms or function calls; the business logic can be encapsulated in object and libraries or the database records can be manipulated with SQL commands directly from individual applications. Any programming style can be used, as long as the data is returned to the calling environment as a proper data structure, with values named according to the interface definition in data structure description.

The XML document holding the data structure description cannot be validated against any Document Type Definition or XML Schema, mainly because the specification allows arbitrary

element and attribute names in this metainformation format. The restrictions that are put on some combinations cannot be caught as the content model does not have fixed element names. However, it can serve as a source not only for the serialization of application data, but for the production of validation specifications as well. For example, a `search.dsd` data structure description file serving as an interface description for application `search.pl` can be used to produce `search.dtd` or `search.xsd`, on demand. The serializers may provide options for including DOCTYPE or namespace information into the output XML document.

 The data structure description is namespace-independent, meaning that it does not require nor prevent using namespaces. The serializer uses element names including prefix of data placeholders for locating the top-level placeholders. The namespace information is not taken into account during this lookup.

Chapter 13. Data structure description reference

13.1. Processing steps

1. Elements `_param` which denote parameter specifications are gathered into one list of allowed parameters and their types. They are removed from any further output processing.
2. All leaf elements with attribute `typeref` are replaced by content of referenced data structure description or its fragment. The target data structure description is normalized before inclusion – parameter elements removed and `typerefed` elements expanded.
3. The data structure description XML document with removed parameter and expanded `typerefed` elements is used to validate and serialize data returned by the application.

13.2. Element names

Element names that have special meaning in data structure description are listed in Table 13.1. Any other elements either serve as placeholders for data returned by the application or are copied to the output XML unchanged. Elements in the output XML document keep names of the elements and placeholders from the input data structure description (with the exception of `_data` placeholder).

Table 13.1. DSD: Reserved element names.

element	description
<code>_param</code>	Input parameter for the application.
<code>_data</code>	Generic way of specifying a placeholder for application data.

13.3. Input parameters

Parameters are defined with element `_param`, its supported attributes are listed in Table 13.2.

Table 13.2. DSD: Attributes of parameter specification.

attribute	description
<code>name</code>	Name of the parameter.
<code>prefix</code>	Prefix of the parameter. All parameters with this prefix will be allowed.
<code>type</code>	Restricts the data type that will be accepted for this parameter. The list of possible values is in Table 13.3. Optional. The default is to place no type restriction on the parameter value.
<code>multiple</code>	When set to yes, allows the parameter to hold multiple values. Optional. The default is no.

Exactly one of `name` and `prefix` attributes has to be specified. The `prefix` specification of parameter can be used for lists where it is necessary to distinguish values whose number and identification is generated dynamically, such as lists of records of the same type. As such parameters cannot be reasonably enumerated, only generic prefix is specified.

Table 13.3. DSD: Parameter types.

type	description
int, integer	Integer value.
num, number	Numeric value.
string	Unrestricted scalar value. The default.

13.4. Including fragments of data structure description

Before the output serialization processing is started, contents of all leaf elements with attribute `typeref` are replaced by data structure descriptions referenced by the URI in that attribute. The Table 13.4 summarizes types of values and their meaning. The `typeref` allows reuse of data structure descriptions by multiple applications. If the applications use objects or library calls to generate data structures (which will come out the same in all cases), it is most convenient to have the structure described in one library file and only reference it.

The serializer should use the `Accept` HTTP header field value `text/x-dsd` to signal the remote server that it requests the data structure description resource, not the result of running the application described by the remote DSD.

Table 13.4. DSD: URIs allowed in `typeref`.

URI	description
<code>#ident</code>	Local fragment. Element with attribute <code>id="ident"</code> will be used.
<code>lib.dsd#m12</code>	Relative URI with fragment. Data structure definition from <code>lib.dsd</code> will be normalized and element with <code>m12</code> identification used.
<code>lib.dsd</code>	Relative URI without fragment. The root element of the data structure definition from <code>lib.dsd</code> will be used.
<code>http://www/lib/people.dsd#ident</code>	Absolute URI with fragment.

13.5. Data placeholders

There are four kind of elements in the data structure description that are recognized as placeholders for application data:

1. Leaf nodes. They can hold scalar values.
2. Elements `_data`. Their attribute name specifies the name of the data value and the name of the element in the output XML document.
3. Elements which have an attribute `type`, specifying what kind of data can be binded to the placeholder. Supported built-in types are listed in Table 13.6.
4. Elements which have an attribute `multiple`, specifying placeholder for aggregate data structures.

The Table 13.5 brings list of permitted attributes for data placeholders.

Table 13.5. DSD: Attributes of data placeholders.

attribute	description
<code>type</code>	Type of the placeholder, one of the built-in types listed in Table 13.6. Optional. By default, any string value is allowed for leaf elements and structure/hash value is used for placeholders that have children elements.
<code>mandatory</code>	When set to yes, it will be an error if the data is not returned by the application. Optional. The default is no.
<code>multiple</code>	Used to specify array and hash values; list of possible values and their meaning is in Table 13.7. Optional. The default is no.
<code>idattr</code>	Specifies the name of attribute which will hold keys of individual values for <code>multiple</code> values hash and <code>hashelement</code> . Optional. The default is <code>id</code> .
<code>hashorder</code>	Order of elements for values binded using <code>multiple</code> values hash or <code>hashelement</code> . Possible values are <code>num</code> , <code>string</code> , and <code>natural</code> , which is the default.
<code>cdata</code>	When set to yes, the scalar content of this element will be output as a CDATA section or a list of CDATA sections. Optional. The default is no.
<code>id</code>	Unique fragment identification.
<code>name</code>	Name of the resulting element and of the data that will be binded to this placeholder. Mandatory for <code>_data</code> elements, not allowed for other placeholders, for those their name will be used.

The attribute type can be specified for elements that specify multiple attribute as well. In that case, the type denotes the type of individual values of the array or the associative array, not the type of the aggregate structure.

Table 13.6. DSD: Built-in data types.

type	description
int, integer	Integer value.
num, number	Numeric value.
hash, struct	Normal nested data structure, default for non-leaves.
string	The default for leaf elements, places no checks on the scalar type.

Table 13.7. DSD: Values of attribute multiple.

value	description
list	An array is expected as the value. The placeholder element will be repeated for all elements of the array.
listelement	An array is expected as the value. The child of the placeholder will be repeated for each of the array's element, not the placeholder itself.
hash	An associative array is expected. The placeholder element will be repeated for all values of the array. The key of individual values will be stored in an attribute <code>id</code> or in an attribute named with attribute <code>idattr</code> .
hashelement	The same as hash, except that the sole child of the placeholder will be repeated for each value, not the placeholder itself.

13.6. Regular elements

The application to which the data structure description corresponds is generally located by the application server or the Web server based on their file names, by changing extension of the DSD file. Root element's attribute `application` can be used to specify another location of the application code.

Table 13.8. DSD: Attributes for application localization.

attribute	description
application	URI that addresses application which should be used to process the request and return the data. Only allowed on the root element.

Any elements that are not used as placeholders will be copied to output unchanged, unless conditional attributes (listed in Table 13.9) specify otherwise. The value of these attributes is always the key of top-level data value in the associative array returned by application.

Table 13.9. DSD: Conditional attributes for regular elements.

attribute	description
if	The element is kept in the document if the value is defined and true. Otherwise, it is removed.
ifdef	The element is kept in the document if the value is defined. Otherwise, it is removed.
ifnot	If the value is true, the element is removed.
ifnotdef	If the value is defined, the element is removed.

By default, no attributes (except those reserved by the XML specification, starting with case insensitive xml) beyond the data structure description specification are allowed in the DSD input file, and all are stripped from the output XML document. By using attributes listed in Table 13.10, it is possible to generate the output XML document with user-defined attributes.

Table 13.10. DSD: Attributes for attribute management.

attribute	description
attrs	Space separated list of attributes that will be preserved in the output XML.
xattrs	Space separated list of space separated pairs of attribute names. The first name in the pair is the attribute name in the source data structure description and the second one is the name to which the attribute will be renamed. This mechanism makes it possible to produce XML documents with user-specified attributes which would otherwise clash with reserved attribute names.

Chapter 14. Output postprocessing

The data structure definition was developed as a tool for serialization of native data structures into the intermediary XML data stream, to keep code of individual Web application free from handling HTML or XML markup. However, as the primary format supported by current Web browsers is HTML, serialization into the target markup on the server is still needed.

Server-side XML transformations have a solid backing in open application programming interfaces of common Web servers, most prominently the open source HTTP server Apache. There are two main projects for the Apache code base, Cocoon based on Java environment and AxKit built on mod_perl, and a number of smaller ones. Nonetheless, they all focus mainly on transformations of existing static XML or XML-based sources. While they could be used directly as the server-side postprocessing backend, XML documents coming from the invocation of dynamic application pose certain complications especially with respect to transformation selection, management, and caching.

14.1. The presentation postprocessing

Once the application data was serialized to correct XML document, it can either be sent directly to XML-aware client, preferably together with processing instruction specifying what transformation to use to reach the target format, or the XML document can be transformed on the server side.

An obvious transformation mechanism for XML data are Extensible Stylesheet Language Transformations (XSLT) [Clark99], however, many other can be found. While specialized or proprietary transformation tools may provide better performance and wider range of functions, the advantage of XSLT lies in client-side support in many modern user agents. If XSLT is chosen as the transformation tool, the same set of stylesheets can be used both for server-side and client-side transformations, configured in run-time. The transformation is driven by templates stored in stylesheets. The stylesheets can `import` each other, providing for flexible setup of functionality. By pointing the transformer to a specialized stylesheet which overrides certain templates, fine-tuning of the output is easily achieved. The stylesheet will only provide specialized handling for elements and areas where change is needed, and `import` the default stylesheet to take over the main task.

When the transformation takes place on the client, the overall throughput of the whole system may increase significantly, since client-side processing does not tie resources of the server.

The individual end users may experience some decrease in interactivity because once the data is received, another HTTP request may be needed to retrieve stylesheets describing the transformation and presentation, followed by the transformation itself. On the other hand, the total

amount of data transferred may be lower, especially once the stylesheets were retrieved by the client and cached. This may have positive impact on clients behind slower network connections, especially if they dispose enough computing power and modern fast browsers. Therefore, it is necessary to judge the impact of the postprocessing distribution among Web server and Web client from a number of positions. Certainly the most significant fact to find out is whether the client supports any transformations of XML data at all. The decision may be affected by:

1. A user making an explicit one-time selection by following a link that enforces server-side or client-side transformation.
2. The preference that the user set and which is retrieved either from their record in authenticated system or derived from cookie data of the HTTP request.
3. The Accept HTTP header field sent by the user agent in the HTTP request. XML-compliant We browsers include value `text/xml`.
4. The User-Agent HTTP header providing certain indication of browser type and version.
5. Heuristic tests based on the previous HTTP requests and handling of responses that the system sent.

If the result of these tests leads to client-side processing, a processing instruction with a URI of proper stylesheet is included in the XML document and the document is sent back to the client, with content type of the HTTP response set to `text/xml`. When the server-side processing is requested, the XML document and the stylesheet is handed over to transformation (XSLT) processor on server, and the result is sent to the client.

In both cases, it is important to have a flexible way of setting *which stylesheet* should be chosen. The browser may prefer XHTML with the content type `application/xhtml+xml` and with attached CSS stylesheet over HTML with content type `text/html` and CSS stylesheet included in the HTML source, although the user may follow a link to a text or printer-friendly version, request output in typographical quality in Portable Document Format (PDF), or ask for output in proprietary format for specialized processing. Especially the choice of the printed output or other special output format is done in a one-time manner, by clicking a link which offers a "printer-friendly" document. Therefore, such a selection cannot be based on permanent settings of the Web browser, it must be driven by the URL of the resource, in order to allow for simple choice.

Processing and selection of the output processing is handled by the serialization and postprocessing environment, not by the individual Web applications. In fact, the application may even not have a way of knowing what output format the user requested. The postprocessing environment processes any well-defined query parameters or parts of the request's URI and configures the output postprocessing chain accordingly.

A single set of parameters driving presentation needs to be defined for each Web system. For example, query string `?style=print` will select print output and `?style=text` textual one. Another parameter select compatibility stylesheets for older browsers, or choose colors of the user interface. Each Web system will have unique requirements concerning the range of options it will provide to its users. Hand in hand with the choice of presentation options that the Web system will use, corresponding stylesheets must come that will provide user interface to reach the selection. Nonetheless, all this functionality is achieved completely without involvement of the application side.



14.2. Alternate HTTP response

A Web application which is run in data-centric mode built around the data structure description returns data structures that conform to the preexisting description. The serialization and postprocessing is handled by the external environment. In many cases, however, the application may want to avoid the postprocessing and return its own HTTP response. These situations include:

- Applications that serve data from external Web sources or from local document storage, where the content type of the document may be arbitrary and not suitable for any further processing. In case of document management systems, it is a strict requirement that the document may not be modified in any way.
- Applications which generate dynamic information that cannot be handled in textual or structured manner. Graphical information like graphs of network traffic, system statistics or business operation are the premier examples.
- Applications that serve as a frontend to other systems, both Web-based and legacy ones, which generate their own user interface.

Due to the internal setup of the system, it may be hard to isolate such applications to separate resource with distinct URI that could have different handling specified directly in the configuration of the Web server or the application server. It is often the application itself that decides on-line, based on many complex conditions, what data in what format ought to be produced. Even if the data is actually served by specialized resource, HTTP redirect may need to be created and served to the HTTP client.

The processing model of data-centric framework considers the response data to be the content of the output, not a metainformation *about* the output. Therefore, a mechanism of returning this metainformation needs to be added to the setup, in order to match real-life requirements. The metainformation could be added directly to the data structure, in a form of special element placeholder that could be filled in by the required data. However, it would place unnecessary

restrictions on the type of information that can be returned, while polluting the data model of the application with service information.

Therefore, an alternative approach is preferred: the application is free to create, form and send the HTTP response itself, while returning only numeric status information as its return data, instead of an associative array with the data fields. Thus, the lack of presence of the data structure signals to the upper layers that the Web application handled the HTTP request fully and no serialization and further processing of the data is desirable. The HTTP response produced by the application may contain a 200 OK status signaling correct result together with Content-Type header field and data in exotic format, or it may produce a 301 Permanent Redirect with no content and let the Web client retrieve the resource from different URI. Note that missing return data structure is different from missing individual values in that structure, which is checked using the mandatory attribute in the data structure definition.

This mechanism of alternate HTTP response is also useful in cases when the application itself decides to change the default presentation and the calling layer was not able to handle the change. Consider an application used for maintaining and creating new presentation stylesheets for output design. The user may upload a new stylesheet into the system and want it activated. As the stylesheet is new, it does not have any internal identification yet, maybe the stylesheet will not pass validation and it will be refused. Only once the application processed the input, stored the stylesheet in database, and assigned it an integer handle, the user's browser can be directed to use it. The application will issue a HTTP response redirecting the browser to a reload page which will acknowledge the upload and have the new stylesheet already activated for postprocessing.

14.3. Operation in CGI mode

The design of the data-centric framework assumes in most parts that the application code is executed in the same process as the upper layer, the application server or the Web server. This is a common processing model supported by widespread technologies like `mod_perl` or Java Server Pages (JSP). The application code is processed by an interpreter or runtime engine embedded in the process of the Web server, therefore it is free to receive input parameters, return data, and call Web server's internal methods directly. However, there are situations when this setup is not available, for security or political reasons. In that case, the Web applications have to fall back to the traditional Common Gateway Interface mode of external processes that are dispatched by the server and communicate with it using environment variables and standard input and output filehandles.

Nonetheless, even in this case, the data-centric model can be used, in two possible modes:

1. All necessary parts of the framework are run in the external process.

2. The input checking and output postprocessing stages (see Figure 12.1 for an overview) are still run in the Web server context, only the application code is executed by the external process. (Consult Figure 12.1 for the overview.)

The first option is most suitable for smaller systems, where it is not reasonable to invest extra effort to installation and administration of embedded interpreter (`mod_perl`) in the Web server. On the other hand, the performance of such solution will be lower, since all necessary software will be loaded and recompiled for each request. That also applies to external data sources like data structure description XML sources and postprocessing stylesheets that will be parsed in each invocation.

Even with the processing in the external process, the application sources can stay clean of any links and function calls to the framework. For example, the Perl interpreter processes environment variables upon startup, one of which is `PERL5OPT`. When set to `-MModuleName`, the named module will be used and can take over the execution, at the beginning with `BEGIN` block and at the end of the process life using `END`. Other programming environments provide similar functionality.

An example of code that ensures parameter validation and processing of data output can be found in Example 14.1.

Example 14.1. Perl module for data-centric processing in external process.

```
package Process_CGI_Data;
use DSD_Processor;
if ($0 =~ /\.(xml|dsd)$/) {
    # error handling omitted for space reasons
    my $dsd_processor = new DSD_Processor;
    my $dsd = $dsd_processor->parse_file($0);
    $dsd->process_env_input;
    my $appname = $dsd->app_name;
    if (not defined $appname) {
        ($appname = $0) =~ s/\.(xml|dsd)$/.pl/;
    }
    package main;
    do $appname;
    my $data = main::handler();
    if (ref $data) {
        $dsd->serialize_and_output($data);
    }
    exit;
}
1;
```

The module code is compiled and executed before the underlying "script" gets called. It then changes the name of the program name stored in special variable `$0` from the name of the data structure description file (which of course does not contain executable Perl code) to the file

name of the real application. The application code is compiled and a function with an agreed-on name handler run. The result is then processed against the data structure description and postprocessed, transforming it into required target markup language. The complete response data is sent back to the Web server and process ended, before the Perl interpreter gets to start processing of the input XML file.

It is important to note that the serialization and postprocessing is run only when the handler function returned a data structure. This approach matches the findings from the previous section — the application can generate the complete HTTP response itself, without any help from the data-centric framework. In that case, the serializer will not interfere.

Since this handling is possible with a single-line setting of the PERL5OPT environment variable in the Web server configuration

```
SetEnv PERL5OPT -MProcess_CGI_Data
```

it can be easily deployed even at conservative places that do not permit the interpreter and framework embedded into the Web server.


The second approach is most suitable for environments where it is feasible to configure the input checking and the postprocessing stage into the Web server, preferably using embedded interpreter of a scripting language like `mod_perl`, while keeping the Web applications in separated processes. The solution utilizing PERL5OPT is used once again, although this time the input parameters were already checked by the code in the Web server and so will be the result and postprocessing. The only outstanding issue is to retrieve the data returned by the application's handler function to the XML serializer running in different process.

Perl itself comes with its own serializer, module `Storable`, which is crafted to match the needs of Perl structures. Unlike the XML serializer designed in Chapter 12, `Storable` does not utilize any validation and is aimed at serializing data structures and their revivification to the same Perl structures.

A module which will achieve the task of passing data to the handler in the Web server process is simpler than the one handling the whole data processing in the external process. The module lets the script compile and execute as if it were regular CGI application, and only in the END block the handler function is invoked and data passed back to the Web server process. A custom content type header `application/x-perl-storable` is used to signal the receiving handler in the Web server the type of the payload.

It is up to the handler within the Web server to thaw the data back to the nested structure and continue with XML serialization and postprocessing as if the data came from code executed within internal Perl interpreter and no external process was involved. As before, the usage of

the module that handles the communication can be driven by the environment variable PERL5OPT. As a result, the code of the individual Web applications is not touched at all, they are written as if they were regular mod_perl handlers.

Example 14.2. Perl module for server-side postprocessing of CGI application.

```
package Process_CGI_Data_Storable;
use Storable ();
END {
    my $data = main::handler();
    if (ref $return) {
        print "Content-Type: application/x-perl-storable\n\n";
        Storable::store_fd $return, \*STDOUT;
    }
}
1;
```

Chapter 15. Implementation

15.1. RayApp reference implementation

A reference implementation of the data structure description validator and serializer was implemented to verify the correctness of the design and to receive performance estimates which could be used to judge the impact of the data-centric framework on real-life Web system projects. Named RayApp, this set of Perl modules provides the functionality described in the previous chapters: parsing of the XML document with the data structure description, validation of input parameters sent by the HTTP request, execution of the application code, serialization of the result data, and server-side postprocessing using XSLT stylesheet. The RayApp distribution is available from CPAN, the Comprehensive Perl Archive Network, from the directory authors/id/JANPAZ, and includes full documentation of its API.

The top-level package is named RayApp and it serves as a dispatcher for all the actions of the serializer. The Example 15.1 demonstrates its interface and usage.

Example 15.1. RayApp Synopsis.

```
use Apache::Request ();
use XML::LibXSLT;
use RayApp;

my $rayapp = new RayApp;
my $dsd = $rayapp->load_dsd($filename);
my $app = $dsd->application_name;
if (not defined $app) {
    ($app = $filename) =~ s/\.[^.]+\./.pl/;
}
my $data = $rayapp->execute_application_handler($app);
if (ref $data) {
    my $outxml = $dsd->serialize_data_dom($data);
    my $xslt = new XML::LibXSLT;
    my $style = $xslt->parse_stylesheet_file($stylesheet);
    $result = $style->transform($outxml);
    # or a utility method serialize_style
    # $result = $dsd->serialize_style($data, $stylesheet);
}
```

The constructor new of the RayApp class returns an object which can be used to load data structure description XML documents, using a set of load_dsd* methods. By loading the data structure description, an object of class RayApp::DSD is created, holding the parsed data structure. The name of the corresponding application is either specified by the application attribute of the root element in the DSD, or is deduced by some external rule. The application is then run via one of the execute* methods which supports both the invocation of the applic-

ation code as a `mod_perl` handler in the context of the Apache Web server, and as a CGI process which serializes the data using the `Storable` module, approach described in the Chapter 14. The execution methods return the data in native Perl associative array, as received from the application code.

Using one of the `serialize*` methods of the `RayApp::DSD` object, the data is validated against the data structure description and serialized to the intermediary XML data document. As in many cases the intermediary XML data will be further postprocessed on server, `RayApp::DSD` also provides utility functions `serialize_style` and `serialize_style_dom` that immediately apply one or more XSLT stylesheets to the result. The intermediary result is stored in Document Object Model and accessed using DOM interface which makes it possible to skip serializing to string and parsing it back between individual stages of the processing.

Since the `RayApp::DSD` objects are linked together via their `RayApp` parent object, their methods can efficiently reuse resources already parsed and loaded by the previous requests. Consequently, XML and XSLT parsers, parsed documents in DOM, and meta-information about the resources can only be loaded once during the whole lifespan of the process.

The `RayApp` serializer can be directly included into the `mod_perl` environment in Apache HTTP server, version row 1.3. The module `RayApp::mod_perl` defines a default content handler `RayApp::mod_perl::handler` which can be configured to handle the HTTP request using the `PerlHandler` directive in the HTTP server configuration, for various configuration sections (Files, Location, Directory). A request for a data structure description file will then execute the underlying application, unless the HTTP header field `Accept` value `text/x-dsd` specifies that the data structure description file is requested instead.

One of the most important issues is the name of the XSLT stylesheet which should be used for the presentation transformation. As the selection of the correct stylesheet may depend on numerous factors, including user's preference stored in the database in an authenticated system or selection via query parameters in the URI, `RayApp` handles the selection of the stylesheet using a user-defined callback function. The system administrator must register a function using `PerlSetVar RayAppStyleSelector module::function` in the configuration of the Web server. This function is called with the `Apache->request` object and should return a list of XSLT stylesheets that will be applied to the XML data document or sent in processing instruction to the client for client-side transformation.

In production systems, a custom version of the `RayApp::mod_perl` handler is likely to be used, adding required special functionality not present in the generic version.

The `RayApp` project uses perl modules `XML::LibXML` and `XML::LibXSLT`, which in turn rely on `libxml2` and `libxslt` libraries for Gnome.

15.2. Performance of the implementation

The absolute performance of the data-centric Web application framework was never the key target. Certainly, a system which produces validated and syntactically correct output will take more processing power than a solution which generates string output which does not follow any formal structure. The following benchmarks were gathered to receive an insight into relative times consumed by individual parts of the setup.

The test machine was a personal computer with Intel Celeron 800 MHz, running Linux kernel 2.4.21, glibc 2.3.2, libxml2 2.5.10, libxslt 1.0.32, and Perl 5.8.1. The machine has 128 MB of memory and during the benchmarks in never used more than 30 per cent of the total amount; the machine was otherwise idle during the tests. The numbers shown are median values of nine runs and were measured using Perl module Benchmark.

Table 15.1. RayApp performance.

test	# per second	ms per call
a simple function call	362605	.003
a simple method call	176978	.006
parsing a file to DOM	696	1.436
parsing a string to DOM	1792	.558
parsing of a XSLT stylesheet	450	2.222
parsing and validating DSD from file	65	15.385
parsing and validation DSD from a string	157	6.369
compilation and execution of a handler	204	4.902
execution of a handler, without a recompile	458	2.183
serialization of data to DOM	232	4.310
serialization of data to a string	215	4.651
execution plus serialization of data	67	14.925
XSLT transformation	1292	.774
execution, serialization, and transformation	61	16.393

The serialization performance is rather poor as compared for example with the parsing and transformation speed of libxml2 and libxslt. The change of context between the Perl and C environment during the processing of the Document Object Model is a probable cause for the slowdown. Better results could be expected in the DSD verification and data serialization was rewritten in C, or if pure-Perl XML parser was used.

In a typical `mod_perl` environment, the data structure description will be parsed once and reused for the whole life of the Apache HTTP server child. The same holds for the code of the application and the transformation stylesheet. The typical setup will thus achieve the 61 calls per second throughput. To get a comparison with a real-life throughput provided by the Apache HTTP server, another set of test were conducted. The version of the Apache server was 1.3.28, the ApacheBench was used as the client. With all tests, the number of concurrent requests was first gradually increased up to a value where the server reached the maximum throughput, before it started to be overloaded.

Table 15.2. Apache HTTP server performance.

test	# per second	ms per call
requests for a static page	104	9.615
requests handled by a <code>mod_perl</code> handler	243	4.115
requests handled by <code>RayApp::mod_perl</code>	27	37.037
requests handled by <code>RayApp::mod_perl</code> in CGI mode	20	50.000
requests by a simple perl script	32	31.250
requests by a perl script which loads 5 modules	23	43.478

The performance of the server-side DSD solution is still higher than an external CGI script with a moderate set of used modules.

Chapter 16. Structure description as a communication tool

In the Part III, we have derived that the output interface of Web applications should have data in structures native to the programming language used for the particular application. To provide stable and verifiable means of building intermediary XML data stream, the data structure description format was proposed and designed. As a result, application programmers can focus on business logic of the application and serialization of the output is done outside the scope of the application code.

Nonetheless, having a formal description of the application interface serves not only the application programmers, in taking away some of their previous work. The data structure description can also be used as a central piece of information about the Web application and serve other members of the development team, both during the development and the maintenance stages. The areas that may benefit from having a data structure description of the interface include:

- Analysis;
- Development of the application code;
- Test cases and tests for the application logic;
- Development of presentation transformations;
- Test cases and tests for the presentation postprocessing;
- Deployment of the application and maintenance of the whole structure of the Web system.
- Management and handling of changes.

16.1. Analysis

In the analysis stage, the data interface can be recorded in the final format in which it will be used during development by both the programmers and Web designers. Both parties can cooperate in fine-tuning the data structure description to suit their needs:

1. The presentation layer may need data that is not displayed verbatim in output HTML pages but which is used to drive the presentation, giving hints how to show the information.
2. The application may need to process and output data that is never displayed but which serves for keeping session and status information across subsequent requests of the same user.

Since the application programmer will no longer create the application markup in the application code, it is essential that those members of the team, who will be responsible for the presentation of the application data, learn the data-oriented mindset, to be able to express their requirements in a way in which the requirements can be fulfilled using the intermediary data layer. No longer can a request be formulated by specifying the markup that the application should produce. For example, "make all rows in the list which have no children in table watch red" is impossible to

satisfy, since the application itself does not produce any HTML markup, therefore it will not produce red markup either. Instead, the information about the data (row does not have any children in the named table) has to be passed out as a data field — either as a boolean value that specifies if there are any children records, or as an integer value which holds the number of children. Here, data structure description for the application serves as a formal requirement that the application programmer should obey.

On the other hand, application programmer may need a certain value to persist across multiple invocations of the application or numerous applications. Data produced or entered by the user at the beginning of a session may need to get transferred across a couple of HTTP requests, in order to be used at the later stage of the transaction. As the data needs to be passed through the response / request cycle via HTML pages, proper HTML code with hidden form fields or hyperlinks that include all necessary data must be generated for the data. In this case, it is up to the presentation postprocessing to produce the required output markup which will ensure that the next HTTP request will arrive with the necessary data.

Data structure description serves as a mutual contract between the application and the presentation part about the interface that the Web application will use. While the semantics of the data fields is not included in the formal specification, it describes the syntax of the interface which can be used to keep the two parts of the system communicating correctly.

16.2. Development and tests of application code

Once the Web application does not produce HTML markup, it is seemingly more difficult to see and judge that the applications work correctly, according to the initial requirements. With the traditional display-driven development and especially with the templating techniques, the HTML page was the most important part of the system. If the application produced HTML output that rendered correctly in a browser, its core was considered bug-free. However, with data-centric Web applications, there is no longer any output page of the application code which could be examined visually. Instead, the application returns complex nested data, in structures that may come directly from the underlying database and that might not be humanly readable per se.

This aspect dramatically increases the importance of automated testing. Since the postprocessing presentation may not yet be finished and available at the time the application code is written, having automated data-driven tests is the only feasible option to verify applications' functionality. Test-first programming [Heatto2] is no longer a methodology option — it becomes the root of the development. The data structure description provides a solid backing for any output that needs to be verified.

1. The type, mandatory, and multiple attributes on data placeholders provide the first level of validation of the output data, specifying types of data for individual data fields, whether the data is optional or mandatory, and if it comes in an aggregate structure. While the type system is not particularly detailed, it was tailored to support the types of data commonly used by the Web applications.
2. More complex interdependencies among the output data fields, as well as the operations that the application execution is expected to produce in the database backend, have to be specified in individual test cases. It was a design decision of data structure description format not to have any incarnation of full expression language for the conditions placed on the output data. Therefore, a test suite of test cases must be developed for each application or unit in the system. Nonetheless, any test case that will be created can build on the fact that the syntax of the output of the Web application is already described.

As a matter of fact, once the data structure description is ready, tests for the application can be written in parallel with the application itself, and often even *before* the actual application development started. Since the interface of the application is declared in the description, the application and the tests will converge in the common target, defined beforehand using the DSD.

16.3. Development and tests of presentation layer

As with the application code, the presentation layer is rooted in the data structure description as well. For the presentation, the intermediary layer in XML is the input side. Since the application format will be serialized according to the data structure description, the input format of the presentation is known even if the actual application might not be ready yet, not producing any actual output. Examples of the application output and its transformations can be written and verified in parallel with the development of the application code. The data structure description provides formal borders to which the XML input entering the presentation is guaranteed to fit in.

The separation of the business logic and its presentation proves beneficial not only for the development itself but for the testing as well. The application code generally needs tests for completely different sets of issues than the presentation layer. For example, the tests of the application code may spend many test cases verifying that the update operations and queries run against the database produce correct lists of data, for given input parameters. For the search application with data structure description shown in the Example 12.5, the test suite of the business logic part will be most concerned with the actual values of records returned for a given set of input parameters `q`, `submit`, and `id`. Validating whether parameter `q`'s wildcard value `Re%` finds the same records as wildcard `Re*`, people with names starting with string 'Re', may be the most explored and the most tested case.

On the other hand, the presentation layer is generally not concerned with the literal string values at all. What may matter more are the numbers and structure of the values and the presentation of corner cases. For presentation layer of the search application, it is irrelevant whether the data it receives is three-record list of people with names starting with 'Re'

```
<?xml version="1.0"?>
<findpeople>
  <people>
    <person>
      <id>25132</id><fullname>Rebeca Milton</fullname>
    </person>
    <person>
      <id>63423</id><fullname>Amanda Reese</fullname>
    </person>
    <person>
      <id>1883</id><fullname>John O'Reilly</fullname>
    </person>
  </people>
</findpeople>
```

or with names starting with 'Mi'

```
<?xml version="1.0"?>
<findpeople>
  <people>
    <person>
      <id>25132</id><fullname>Rebeca Milton</fullname>
    </person>
    <person>
      <id>25132</id><fullname>Michael Stas</fullname>
    </person>
    <person>
      <id>72243</id><fullname>Michael Wayne</fullname>
    </person>
  </people>
</findpeople>
```

Unless the presentation also considers text content of the XML elements, the second test case does not bring any additional information. It might be more worth to have tests which would verify how the information is presented when there is no output, one record, and more than twenty records, versus the existing three-record case.

Yet, in traditional systems where the intermediary data layer is not available, business logic and presentation can only be tested together. In such cases, not only any change in presentation of the output may break tests that the application programmers are using to validate core applications' behavior, but the structurally identical tests add unnecessary noise to the testing of the presentation side as well.

Separation using data layer solves both problems, creating a new point in the application operation where tests can be conducted.

16.4. Structure of the Web system

Data structure description is primarily concerned with the interface of a single application. Nonetheless, since it includes specification of input parameters that are valid for that particular application, it can be also used to check dependencies among applications in the whole Web system.

Relations among Web applications are created using two mechanisms: hyperlinks and Web forms. It is the task of presentation layer to create the output HTML or XHTML page in such a way that following a hyperlink or submitting a form will trigger HTTP request with correct parameters for the target application. A test suite for the presentation layer can be used to achieve consistency with the input interface of referenced resources. Any hypertext link and any HTML form that the tests produce can be processed and verified against the data structure description of the target URI. The parameters and the content of the request have to match `_param` parameter names, prefixes, and types. Checking whether the application operates correctly not only as a standalone information source but also as a part of larger system is an important part of integration [Beck99]. This feature is not available with traditional systems because the information about allowed input parameters is nowhere available in declarative form that could be processed and used for validation, short of running all the target applications.

Certainly, passing the formal test of valid parameters does not provide a 100 per cent guarantee that the link is not broken in a functional way. If by mistake the presentation layer shows Amanda Reese's data

```
<person>
  <id>63423</id><fullname>Amanda Reese</fullname>
</person>
```

as a hyperlink pointing to a different person

```
<li><a href="search?id=25132">Amanda Reese</a></li>
```

clicking on the link will not give semantically correct output. Nevertheless, each layer of validation decreases the space in which outstanding errors may be hiding. When the presentation transformations are written in a sensible way

```
<xsl:template match="person">
  <li>
    <a>
      <xsl:attribute name="href">
        <xsl:text>search?id=</xsl:text>
        <xsl:value-of select="id"/>
      </xsl:attribute>
      <xsl:apply-templates select="fullname"/>
    </a>
  </li>
</xsl:template>
```

it is far harder to make a mistake that will replace data value with a completely different one than to mistype literal string denoting the name of the parameter.

16.5. Handling changes

Validating cross-references becomes even more important in the maintenance stage, when the system evolves and the applications change. Usually, it will not be the new application which would come with broken presentation with a failed link — that omission will probably be caught by the quality assurance team during user testing. More mistakes can be expected when an application is refactored or when new features are added. Change in an application often enforces changes to the external interface as well. In such a situation, having input parameters defined in the data structure description resource serves two purposes:

1. It reminds the developers what the original interface was.
2. If the interface was changed, it makes it possible to run integration tests across all applications in the system to validate that none use the old parameters.

Thus, the formal description is not only mutual *contract of interface* between application programmers and Web designers of one application, but also an external one, across application boundaries.

16.6. Audit and logging

The intermediary data layer which is in the middle of data-centric operation of Web-based system is helpful not only in development and maintenance, but in troubleshooting as well. Since the output data is serialized to the XML format, it can be logged and stored for later analysis. Should a user report anomaly in system operation, logging of their communication with the system can be switched on.

The data can be checked in their pure form, making it easier to find out if a problem is in the business logic (wrong operation was done or wrong data was generated) or in the presentation part (the correct data was not postprocessed correctly). The output data can be viewed either through the output presentation transformation used by the user in question, giving the exact output page, or using a debugging, raw presentation to visualize the data result. With traditional systems without central data layer, system administrators are dependent on the output-end HTML-only format.

Chapter 17. Multilingual Web systems

Web applications, like most other types of computer systems that have human oriented interface, combine two types of data in their output pages and screens: primary dynamic data which is a result of application invocation and often comes from database backend, and additional textual information which helps to navigate through the application and provides necessary structure or helpful comments. Besides multiple output formats, an often-required functionality of Web systems is support for multiple languages and localized user interfaces, providing parallel contents in multiple languages. The audience of Web systems is fast becoming global and many firms and organizations find themselves needing more than the primary language on their Web site, to better communicate with their remote visitors.

Since the core data is often managed in the database in many localized versions already, the task is to retrieve correct language version of the data and provide the dynamic content with correct presentation wrapper. Similar to the case of multiple output formats, the traditional approach to multilingual Web applications as separate code for each output language, as the need for localization is gradually realized. Soon, the application code is filled with constructs similar to the Example 17.1. With such a code, additional `elsif` branches are needed when a requirement for yet another language occurs. The business logic gets duplicated which complicates the maintenance of the application, as modifications done in one language branch will have to be manually synchronized into others, calling for mistakes and omissions. To prevent maintenance becoming a bottleneck of the long-term viability of the system, a better way of handling internationalization and localization of Web application is needed.

Example 17.1. Code handling multiple languages with multiple branches.

```
if ($lang eq 'en') {
    ($title) = $dbh->selectrow_array(q!
        select name_en from courses where id = :p1
        !, {}, $course_id);
} else {
    ## the default
    ($title) = $dbh->selectrow_array(q!
        select name_cs from courses where id = :p1
        !, {}, $course_id);
}
```

When building internationalized applications, the following areas have to be handled:

1. Support for character sets and encodings which can hold localized texts for individual users.
2. Proper handling of user input.
3. Localized versions of data stored in databases, for example product and services names and their descriptions.

4. Translations of surrounding navigational and informational texts and of all parts of the user interface.
5. Correct presentation of date, time, monetary, and numeric values.

In the area of character set support, both client and server side of the Web technology advanced from the simple seven-bit US-ASCII character set over the Western ISO-8859-1 towards fully global infrastructure, using Unicode [Unicode]. Unicode Transformation Format-8 (UTF-8) is supported by all modern user agents and it is the default internal and output encoding of XML tools. Many conversion tools exist, providing transformation to other encodings for clients that do not accept UTF-8 natively. Modern Web clients support metainformation about Web data correctly, making content handling in localized character sets less tedious task than it used to be in the early years.

Handling user input correctly is mostly a client-side task, where keyboard settings or speech recognition take place. The Web system only receives the data once it arrives in the form of a HTTP request. While character set metainformation is not included with the payload of the request, most modern Web browsers send in the HTTP request in character set corresponding to the one of the previous HTTP response. As an additional guard against broken data due to unrecognized character set change, hidden field or query parameter can be included in each HTML form or hyperlink, with a predefined distinctive value. In the Czech environment, letter Š (S with caron) might be used as it takes different positions in all commonly used character sets. By checking the value when it is sent back to the server with the next HTTP request, irregular changes in the character set selection can be detected.

In the following sections, we shall focus on the remaining internationalization targets from the above list.

17.1. Localized database data sources

Most of dynamic Web applications use relational database management systems (RDBMS) as the backend data storage and management tool. The data is organized in relations of tuples and stored in database in tables with records with attributes / columns. Besides database tables which provide the core storage, many other types of database objects are commonly supported, i.e. database views for virtual relations, stored functions and procedures, triggers, and packages.

Common relational database systems often support further partitioning of the collection of database objects into separate namespaces. For example, Oracle RDBMS uses *schema* as the primary tool of distinguishing database tables and other database objects of individual users. Each table is identified by its name and its schema. There may be tables with the identical name in different schemas, without causing a conflict. Each database session runs with exactly one schema set as its current schema. The current schema makes it possible to address tables or

other database objects by only using the names, without specifying the fully qualified names. To reach an object in different schema, the full name including the schema part must be used.

Database systems which support schemas and current schema setting for database sessions (some systems may use different term for the same concept) can be used to provide Web applications with localized database data sources. The solutions builds on the fact that the same database command or query has different meaning depending on the current schema in which it is run.

Consider a Catalogue of course offerings in a university information system, which needs to be presented in two languages, Czech and English. The main goal is that the application should not be changed to handle those two languages. One code is to be used for both language-specific operations. Assume that the information about the courses is stored in a single database table, `courses`, with a structure outlined in Figure 17.1. The name of the course, which is the only textual information that needs to be translated, is stored in two table columns: the Czech name in the column `name_cs`, the English translation in the column `name_en`.

Table courses			
id	name_cs	name_en	other fields

Figure 17.1. Database table courses.

To avoid code like that shown in Example 17.1, two database views on top of this table will be created, mapping the columns `name_cs`, `name_en` to a single column named `name`. The views will be created in different language-specific schemas (`czech`, `english`), with the same name as the original table (`courses`). The table itself shall be stored in yet another schema, `base`. See Figure 17.2 for the overview of the whole setup.

Should the base table be located in a schema `base`, the two localized views will be created by two data definition language commands:

```
create view czech.courses
as
select id, name_cs name, credits, semester_id
from base.courses
```

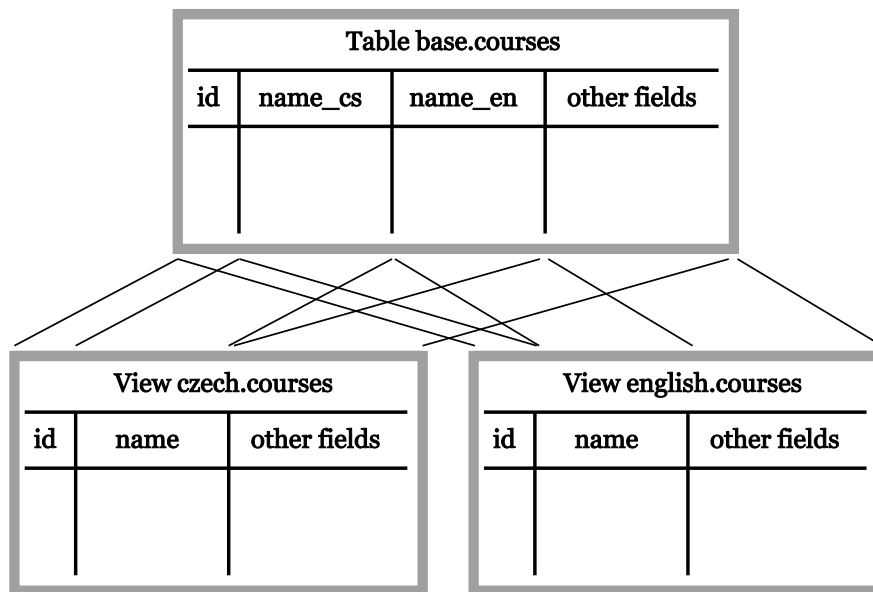



Figure 17.2. Localized database views.

```
create view english.courses
as
select id, name_en name, credits, semester_id
from base.courses
```

Note that a column alias provides mapping of the original two columns with localized versions of the course name to a single column with name name.

With such an database schema, SQL command

```
select name from courses where id = :p1
```

will now generate different results, depending on the current language-specific schema in which it is run. If the current schema of the active database session is czech, the query will be internally rewritten to

```
select name_cs from base.courses where id = :p1
```

In the english schema, the effective query executed by the database system will be

```
select name_en from base.courses where id = :p1
```

The single SQL command achieves different results depending on the setting of current schema in the database session which the client uses.

The mapping of the column names may be more dynamic than the plain renaming of columns to a common column name. Consider a situation when the primary language at the university is Czech and not all course records have the English name filled in the database. In such a case,

it is reasonable to fallback to the Czech name if the English translation is not available. Not perfect, but it is still better solution than giving out no name at all. A view definition

```
create view english.courses
as
select id, nvl(name_en, name_cs) name, credits, semester_id
from base.courses
```

will map the name column to an expression `nvl(name_en, name_cs)`. The `nvl` function returns its first argument if it is not null, otherwise it returns the second one. Therefore, select from view `english.courses` always returns at least the primary Czech value, if the people responsible were slow to update the database with full translations.

Consider that the third language version needs to be added to the Catalogue and should be presented using the same, language-independent SQL command. The third column (`name_de`) might be added alongside the first two language versions. However, such extensions could soon lead to polluted, unmaintainable database schema with dozens of columns in each database table. For minority languages, a more flexible approach can be taken. A generic translation table holding translations of strings from the primary language to multiple secondary languages will be created. The German courses view will consult the translation table first and fallback to the English or Czech course name if no translation was found:

```
create view german.courses
as
select id, nvl(nvl(
    select target_text from base.translations
    where orig_text = name_cs
      and lang_id = 'de'
  ), name_en), name_cs) name, credits, semester_id
from base.courses
```

As the number of requests for German translation is likely to account for only a small percentage of all selects, the subselect with the `base.translations` table involved in execution of this view will not have significant impact on the overall throughput of the system. Note that the translation table is a generic solution, holding not only course names, but texts that need to be translated, from any table in the system.

In the described setup with localization schemas that hold language-specific versions of database tables, the issue with correctly handling updates of values in tables may appear. Inserts, updates or deletes on database views are only supported by the database systems if those views do not contain constructs which would prevent the database system to correctly locate the records and their attributes in the underlying base tables. This obstacle can be overcome with *instead of* triggers that map data manipulation commands on views into user defined sequences of operations on base tables of the views. The localization views can then behave exactly like regular tables.

17.2. Application-transparent setting

Names of database tables are resolved according to the current schema of the database session. The current schema can be changed by reconnecting under different user name which corresponds to the required schema. Many database systems also offer some kind of vendor-specific SQL command for changing the default schema of running database session without a need for a disconnect and a new connection. For example in Oracle RDBMS, `alter session` command is supported:

```
alter session set current_schema = english
```

After this command, a query

```
select name from courses where id = :p1
```

will search for a table or view `courses` in the database schema `english`.

In the data-centric Web application framework, setting up correct database connection with correct default schema is part of the general environment from which the application code is invoked. As with the selection of presentation format discussed in the Chapter 14, the required language can be deduced from numerous inputs, including `Accept-Language` HTTP header field, preference stored in the database backend or in a cookie value, or an explicit parameter in the URI of the HTTP request. The calling environment in the Web server or the application server will process and strip this information and select a localized database schema best matching the request, before the application code is invoked. In cases when the application code is run by an embedded interpreter or virtual machine in the Web server or the application server process, the application code will be called with pre-opened database handle already set to the correct localized database source. When the application is run as separate process, the naming of the database source can be stored in an environment variable and handled by the database connection library.

By using localized database schemas with database object mappings, a single application code can be executed in multiple language environments, providing instant localization without any change needed in application code. The selection of the schema is done by the calling environment, not by the individual applications. Therefore, the application does not need to know at all to what database schema its database session is currently set.

Nonetheless, in the real systems, it would be uncomfortable for the users to switch to different localized user interface to for example edit the name of the course in a foreign language. The head of department responsible for their course records may want to insert all language variants of the course name from one HTML form, which will presumably be in one language. Thus, the application will have to be able to manipulate with the language-specific data not only using the localized views, but directly. Fortunately, the applications can always access the base tables

in the base schema and modify all language versions of the record with a single SQL command, without any changes in the current schema setting being necessary.

17.3. Localization of other database objects

Database tables provide the core data storage in relational database systems, but language specific information is present in other database objects as well. Stored functions, procedures, and package may process and produce texts which might be language-specific. These objects have to be identified and their correct localized versions made in individual language schemas.

Nevertheless, many of the database tables and other database objects contains no language-specific data at all: numeric, date, and time information and texts for which no translation is intended, for example email records or names of people, do not need to be handled using language mapping views. However, applications will be running in localized schemas as their current schema and it would be cumbersome to prefix access to each language-independent database object with the base . schema identifier. Many database systems provide ways for alternate naming of database objects: synonyms. As they can point across database schema borders, it is possible to have tables in the foreign schema named with a local name as well. For example table `base.grades` with students' number-only study results can be referenced by a synonym `czech.grades`, making it possible to run a query

```
select grade, credits, changed_by from grades where id = ?
```

in the context of the database schema `czech`.

For multilingual systems, synonyms pointing to the base database objects have to be created in all language-specific database schemas for all user-accessible objects to provide the same local background for queries and data manipulation commands run by applications in all language-specific contexts. This task can be automated with a script which will retrieve list of all database objects in the base schema for which no corresponding object of the same type (with the exception of views for tables) exists and create the mapping synonym in all the language-specific localized schemas. If later a need for localization of a particular object arises, the synonym will be replaced by an more complex object providing the language translation.

Another possible approach to the mapping of the language independent database objects is to create public synonyms for them. That way, the objects would be made visible instantly from all other schemas. Any schema would still be free to have its own version of the object, as during name resolution, local objects are considered first. Only if no object of the name in question is found in the current schema, list of public synonyms is consulted.

With all three ways of mapping objects across schemas, correct access right settings are also necessary. Synonyms and views only create name resolution links between database objects but the access has to be granted by the owner of the target database objects.

Simple mappings using synonyms or views that rename columns from the base tables imposes no speed penalty on the execution of queries, benchmarks on Oracle RDBMS (version 8.1.7 and 9.0.1) have shown. Views which use subselects and joins to retrieve translation data in a more complex manner will use different execution paths and have to be fine-tuned carefully to minimize the performance impact. In the code of the individual applications, no branching due to language selection is needed, leading to smaller and faster code of individual applications. Overall, language-specific database schemas with database object mapping provide efficient way of accessing localized versions of dynamic data instantly, for all applications in the system, without any change in the end application code needed. In the data-centric Web applications it is readily available, as the setup includes a central layer which handles the request before it is processed by the application, therefore making a central language setting and administration possible.

17.4. Localization of time and numeric values

The presentation format of date, time, monetary or numeric values is highly language-specific as well. The names of weekdays and months, the 12-hour versus 24-hour format of time, the way decimal numbers are written, these all have different representations on output based on language preferences of the user. The same approach as with language-localization of database schemas can be used to transparently provide Web applications with format-localization as well.

When setting the current database schema to provide localized data source, the Web server or the application server will at the same time setup all other aspects of the language selection as well. Either through environment variables or using administrative SQL commands, the database system can be set to change formatting styles of values it receives and generates. As before, the setting is done outside of the code of the individual applications and is transparent to them.

17.5. Preprocessed stylesheets for static texts

The dynamic content which comes from the database backend is in the resulting output wrapped in more or less static texts that provide context and explanation in human language for the data retrieved from database. Static text can also serve navigational and help purposes. In the traditional systems, static output texts often appear in the applications as literal texts that get printed by the application code. When internationalization time comes, these static texts are replaced by calls to appropriate functions that provide lookup of correct translation, based on the active language settings in application's environment [Drepper96]. The search in translation

catalogue is done in runtime at each occurrence of such text. After the resulting text is merged with other data, its origin blends with other information printed out.

In the data-centric Web application framework, the static texts are only inserted to the output in the postprocessing stage, using stylesheets which are used to generate the target format of the application output. Therefore, localization calls can be expected to take place within the output stylesheets, as part of the rest of the presentation transformation. Indeed, this is exactly the path taken for example by the XSLT stylesheets for the DocBook documentation system [Walsho2].

However, the localization often requires more than just a text replacements. Assume that an application in a university information system produces an XML result of searching for students according to a certain criteria:

```
<active_students>
  <name>Jane Hunt</name>
  <name>Peter Pine</name>
  <name>Tom Wolf</name>
</active_students>
```

The list is to be formatted on the Web page as a bulleted list using the `ul` and `li` elements of the HTML markup. On top of the list, a summary line telling the total number of students matching the search criteria should be added:

The search found 3 students.

- Jane Hunt
- Peter Pine
- Tom Wolf


This output can be achieved by the following XSLT template:

```
<xsl:template match="active_students">
  The search found <xsl:value-of select="count(name)"/> students.
  <ul>
    <xsl:apply-templates />
  </ul>
</xsl:template>
```

The template matches the element `active_students` in the XML document and replaces it with the message which includes the `xsl:value-of` instruction element that computes the number of students from the data source. This value then becomes part of the message that appears on top of the list of students.

The language-specific parts of the stylesheet, in our example the sentence about the count of students, have to be identified and translated into other languages. The results will likely be

stored in a message catalogue, providing mapping from the primary language to other languages, and addressed by special template calls. The translated texts have to be stored as self-contained grammatical expressions because it would be wrong to break the sentence above to two parts, the first one before the number and the trailing one with the word "students". For example in German, the verb "gefunden" would appear at the end of the sentence and such a split of the message would not serve well. Thus, the localization catalogues have to accommodate for variable positioning of parameters in translated texts. The goal is generally reached by special escape sequences (%s, %s) which are replaced by the appropriate parameters.



Nonetheless, having a special localization template invoked each time the output needs to be language specific brings unnecessary overhead. The question arises, could the run-time impact of the localization task be minimized, benefiting from the data-centric Web application framework? With traditional applications that produce the output step-by-step, the lookup in the message catalogue has to be done for each occurrence of the internationalized data, as it will not be available at later stage. However, with Web systems built around intermediary data layer, the presentation transformation is only done in one place, processing the whole output of the application with one transformation series. In that case, it is possible to pre-process the stylesheets which will be used for the transformation and generate language-specific stylesheets. After all, all the message catalogue lookups of an application running in the German context will head to the German message catalogue.

Therefore, a set of localized stylesheets will be created off-line, with literal texts and constructs worth translating replaced in-place by their localized versions. When the stylesheets are used for transformation of the XML output of application data, no message catalogue lookups will be needed, as the whole stylesheet will already have been localized.

This approach builds on the same ideas as the the localization of database sources. The core of the localization was shifted from run-time to the definition stage. The localized database schemas provide translated data, localized stylesheets provide translated transformations. In the run-time, localization only consists of selecting proper XSLT stylesheet which should be applied, using similar decision criteria as with selection of output format or setup of language-specific database handle. For larger XML documents, the time it takes to transform it using template based system like XSLT is directly related to the number of templates that are invoked during the transformation. By reducing the number of templates that have to be processed, significant throughput improvements were observed.

Note that by liberating the localization of transformation from the strict borders of message catalogues and their calling conventions, better results can be used by hand-tuning the transformations to best match individual linguistic requirements. A normal Czech translation of the above XSLT template producing message with the number of students might read

```
<xsl:template match="active_students">
  Bylo nalezeno <xsl:value-of select="count(name)"/> studentů.
  <ul>
    <xsl:apply-templates select="*" />
  </ul>
</xsl:template>
```

However, such a solution which mimics the message catalogue approach by replacing a single sentence with its translation does not provide adequate results in Czech for all possible numbers of students. In the Czech language, the form of the noun "student" changes with the number, therefore a stylesheet with a set of options is preferable:

```
<xsl:template match="active_students">
  <xsl:choose>
    <xsl:when test="count(name) = 1">
      Byl nalezen 1 student.
    </xsl:when>
    <xsl:when test="count(name) &lt;= 4">
      Byli nalezeni <xsl:value-of select="count(name)"/> studenti.
    </xsl:when>
    <xsl:otherwise>
      Bylo nalezeno <xsl:value-of select="count(name)"/> studentů.
    </xsl:otherwise>
  </xsl:choose>
  <ul>
    <xsl:apply-templates select="*" />
  </ul>
</xsl:template>
```


Note that the change only took place in the stylesheet, not in the application code. No longer are ifs selecting proper grammar needed in the individual applications. Of course, for proper output, a case of no students could also be included. Any language-specific stylesheet may provide as complex transformation as needed for the language in question, without forcing other language versions to provide the same complex rule as well.

As the XSLT stylesheets are XML documents which can be edited using text tools, the maintenance of consistency between the primary language version and the translations can be done using regular text or XML-aware tools. A translation is then defined by a document change from the primary stylesheet. Any time new version of the primary stylesheet is created, the change set is reapplied to receive new translations of all language mutations.

17.6. Multiple content data sources

The idea of specialized data sources for multiple languages can be reused for non-language-oriented localization as well. Consider an HTTP request with Accept header field set to `text/vnd.wap.wml`, denoting WML (Wireless Markup Language). This markup is often used by mobile devices which have limited display dimensions.

To provide the client with information which will better fit its small screen, the Web server or the application server may decide to run the application code with a database handle set to special database schema. The schema will provide access to shorter product, course or bus stop names, in order to deliver reasonable content that would not overwhelm the mobile device. The application will then transparently use any environment set by the calling server. The selection of mobile device targeted database schema is orthogonal to any language-specific selection. The Web system may utilize any of these mechanisms, or both at a time.

 A true separation of content and presentation is reached once the application which generates the content has no need to know how the data will be presented, what language version it is serving, and for which user agent platform. The intermediary data layer allows such an isolation.

Chapter 18. Migration and implementation rules

18.1. Separation of tasks

When the data-centric framework is used for development of complex Web-based systems, parts of the system have to be identified carefully, in order to achieve well-balanced distribution of tasks between the application code and the presentation transformation. A list of students which the application returns might be serialized to an XML document

```
<students>
  <name><id>729232</id><first>Jane</first><last>Hunt</last></name>
  <name><id>728233</id><first>Peter</first><last>Pine</last></name>
  <name><id>709227</id><first>Tom</first><last>Wolf</last></name>
</students>
```

Should the output be sorted, what part of the setup is to provide the sorting functionality for the resulting data set. Should the application return the data already ordered according to specified criteria (by last name, internal identification, by students' percentile), or should the result be sorted by the presentation transformation?

In all cases, the intended usage of the system and the relative processing power of the client and server side have to be taken into account. Most often, the underlying database management system will be able to provide the data already sorted according to `order by` clauses, efficiently using its indexing and data-manipulation features. In addition, the user may specifically request different ordering than the default and the application might compute the ordering according to specific needs. Therefore, in most cases the application will provide the data ready for seamless consumption by the presentation side.

That also includes other manipulations that technically may be conducted in both parts. If a comma-separated list of teacher names is to be displayed next to each course name in a departmental schedule, the comma-separated list might be generated either by the application directly, or the application may return an array data structure and it will be up to the postprocessing transformation to drop unnecessary pieces and come with a list which was required.

```
<course>
  <name>Database Systems</name>
  <teachers>Wilton, Gork</teachers>
</course>
```

```
<course>
  <name>Database Systems</name>
  <teachers>
    <person><id>823</id><last>Wilton</last><first>John</first></person>
    <person><id>119</id><last>Gork</last><first>Peter</first></person>
  </teachers>
</course>
```

Careful consideration of the underlying database schema is needed, and the finding should be reflected in the data structure description. Chances are, the database system already holds names of the teachers precomputed in a special table column, to speed-up some other operations. In that case, using the data which exactly matches the required field is preferred. On the other hand, the request for comma-separated list might only be valid for a printed output typeset in a high quality. For on-line schedule, a hyperlinked teacher name could be preferred, with a link to each professor's home page. In that case, the more generic format with the structured information should be used, as it is easily converted to a string where needed using a simple XSLT transformation:

```
<xsl:template match="course/teachers">
  <xsl:for-each select="person">
    <xsl:value-of select="last"/>
    <xsl:if test="position()!<=last()">, </xsl:if>
  </xsl:for-each>
</xsl:template>
```

Output data is *well-structured* if it ideally matches the needs of the consumer. If a different ordering or structure of data is seen necessary by the postprocessing part, request for change ought to be raised and a change of interface initiated. The data structure description in data-centric framework serves formalizing and communication goal, nonetheless, it should not pose an obstacle in developing the interface further, as new output formats and user requirements keep coming. Quite the contrary, the data structure description helps all involved parties to better communicate the change, keeping the facts spelled out in a formal way.

18.2. Migration of existing applications

When existing Web applications are being migrated to the data-centric framework, analysis of the existing code and the existing output, as well as possible additional requirements, is necessary. There are three types of data in Web applications [Kohoutek01]:

1. Dynamic data;
2. Static data;
3. Data which holds state of the session;

The dynamic data usually comes directly from the database backend and the information is copied verbatim to the output, by the presentation transformation. Names of students or teachers produced by a query in a university information system or a list of airfares in a reservation system are examples of dynamic data. Static data include the additional informational texts, as well as names of the input parameters, and presentation of Web forms, for example a name of the textfield used to enter a search pattern and its maximum allowed length. This data will most likely be produced by the postprocessing presentation transformation.

Data which holds the state of the session include status and values of form elements in the HTML page, such as checkboxes that are checked or the default value selected by a popup menu. Since in the data-centric Web application framework the application programmer no longer controls the GUI elements and fields by creating HTML markup which would implement them, the information must be added into the data structure description in order to be correctly generated by the application and handled by the presentation part.

While the application code may become simpler and cleaner in the part which generates lists and tables of objects as the complex data structures fetched from the database are simply sent to the XML serialization, handling the data which holds state of the user session may prove to be the most tedious task. All the information which is needed to recycle a Web form with the same data that the user sent in, with active elements in the GUI set to correct default values, have to be explicitly named and generated.

The data-centric approach will shift substantial amount of work from the shoulders of application programmers to the authors and maintainers of the presentation transformations. The first applications in the system will likely generate more work and bigger source codes than if the same applications were written in a traditional, procedural or template-oriented way. Each new application which will be producing new data structures which will require new transformation templates. Nonetheless, once templates for core objects that are most commonly used in the information system are in place, the marginal work with each new application is diminishing, as no new transformations are required. Through reasonable choice of the data interface and names used in the data structure, later applications will fully reuse presentation code prepared for the earlier ones.

The work of [Kohoutek01] was based on a sample of real-life applications and output data taken from the Information System of Masaryk University, which is written using the traditional approach. In most cases, the XML intermediary document was 5 to 60 per cent smaller than the HTML code generated. That is exactly the savings of network bandwidth which can be expected when the presentation transformation is shifted to modern Web clients. In some cases, the XML document could be made even smaller by using less verbose element names. The size of the code of individual applications dropped by 45 to 80 per cent. As many of the applications in the administrative system deal primarily with presenting database records to the user, in many cases the application code was reduced to a series of `selectAll_arrayrefs`.

The XSLT stylesheets used to transform the XML data to HTML code equivalent to the original IS MU output grew fast initially, as the work focused on a wide range of applications. By doing a test migration of applications similar to the one already done (listing teachers' sent email folder and listing of newsboard, which were similar to regular mailbox email reader), many templates of the existing transformations were reused and the parts specific for these particular

applications consisted mainly of the text messages. By reusing existing XSLT templates, a more consistent user interface was achieved, as the presentations shared common elements.



Chapter 19. Conclusion and further work

The intermediary data layer which pipes data from application towards presentation postprocessing plays an important role in separation of content and presentation. The separation is further stressed by placing application code into separate source file away from the presentation stylesheet and by having the interface formalized using data structure description. That way the applications can only return pure data, in data structures native to the programming language in which they are written, and serialization to XML is done by the calling layer. The intermediary XML document can even be sent directly to the client together with processing instructions attached, as increasing number of Web browsers supports XSL transformations and will be able to generate the target format themselves.

The data structure layer defines what input parameters the application accepts and what output data structures it produces. Therefore, postprocessing transformations can be developed and tested in spite of the fact that the application itself may not be ready yet. And vice versa, the application development is not dependent on the postprocessing, leading to concurrent work by the two parties. Test suites are rooted in the data structure description as well, as the output of business logic and input of the presentation part are well-defined. The test cases can be written according to the interface definition.

Once the applications are freed of the HTML dependency, more flexible approach to their development is possible. The application can be run on top of localized database sources, providing multilingual support, or postprocessed with alternate stylesheets, leading to multiple output formats. As the application code only includes the business logic part, it is reusable, smaller, and readable, hopefully leading to less errors and easier maintenance. The presented framework will bring most prominent advantages to large systems where the number of individual applications has grown high. For small systems with only a couple of distinct applications, formalized internal structure may seem like an unnecessary burden.

The data structure description focuses primarily on the internal interface of a single Web application. While it can also be used for integration testing and reverse validation of the presentation results, individual applications are considered to be strictly separated. While this may prove useful during development, the application will live in the context of the Web system and other services. Therefore, interaction of the data-centric paradigm with the external Web system and its structure is natural topic for further work. As many Web system start to use Web services as their data sources, the one-to-one mapping between the URI resource and the application that serves the content may no longer be valid. Exploring how output and layout of such systems could be handled is another direction for further research.

The performance of the reference implementation of the data structure description serializer, RayApp, could certainly be improved by using either full C or pure-Perl implementation. The extensive suite of its tests makes even gradual development and fine-tuning possible, since the correctness of the code is anchored in hundreds of test cases. With XForms advancement to W3C Proposed Recommendation, XForms compliant user agents can be expected soon, extending the external interface of Web applications with complex XForms model. Therefore, data structure description should be extended to handle it natively.



References

- [Beck99] Kent Beck. *Extreme Programming Explained: embrace change*. Addison-Wesley Publishing Company, 1999.
- [Berners94] Tim Berners-Lee. Universal Resource Identifiers in WWW. RFC 1630. CERN, June 1994. Available at <ftp://ftp.rfc-editor.org/in-notes/rfc1630.txt>.
- [Berners96] Tim Berners-Lee. “The World Wide Web: Past, Present and Future”. *IEEE Computer*, October 1996, pp. 69–77. Draft available at <http://www.w3.org/People/Berners-Lee/1996/ppf.html>.
- [Brandejs01] Michal Brandejs, Iva Hollanová, Miroslava Misáková, Jan Pazdziora. In-house Developed UIS for Traditional University: Recommendations and Warnings. In *Proceedings of the 7th International Conference of European University Information Systems (EUNIS)*, Humboldt-University Berlin, March 2001, pp. 234–237. On-line at <http://is.muni.cz/clanky/eunis2001-casestudy.pl>.
- [Bray00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler (Eds.). Extensible Markup Language (XML) 1.0. Second Edition. World Wide Web Consortium (W3C), October 2000. W3C Recommendation. Available at <http://www.w3.org/TR/2000/REC-xml-20001006>.
- [Bos98] Bert Bos, Håkon Wium Lie, Chris Lilley, Ian Jacobs (Eds.). Cascading Style Sheets, level 2. World Wide Web Consortium (W3C), May 1998. Available at <http://www.w3.org/TR/1998/REC-CSS2-19980512>.
- [Brown01] Allen Brown, Matthew Fuchs, Jonathan Robie, Philip Wadler (Eds.). XML Schema: Formal Description. World Wide Web Consortium (W3C), September 2001. Available at <http://www.w3.org/TR/xmlschema-formal/>.
- [Casalicchio02] Emiliano Casalicchio. Cluster-based Web Systems: Paradigms and Dispatching Algorithms. Doctoral dissertation. University of Roma Tor Vergata, March 2002. Available at <http://www.ce.uniroma2.it/publications/PhDThesis-casa.ps>.
- [CENISSSo0] CEN/ISSS. Amazon.com Case Study. On-line at http://www.iss-awareness.cenorm.be/Case_Studies/Amazon_frame.htm.
- [Chang00] Liu Chang, Kirk P. Arnett. “Exploring the factors associated with Web site success in the context of electronic commerce”. *Information & Management*, Volume 38, Issue 1, October 2000, pp. 23–33.

- [Clark99] James Clark. (Ed.). XSL Transformations (XSLT). World Wide Web Consortium (W3C). November 1999. Available at <http://www.w3.org/TR/1999/REC-xslt-19991116>.
- [Clark01] James Clark, Murata Makoto (Eds.). RELAX NG Specification. Organization for the Advancement of Structured Information Standards, December 2001. Available at <http://www.relaxng.org/spec-20011203.html>.
- [Comer95] Douglas E. Comer. *Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture*. Prentice Hall, 1995.
- [Copeland97] Kenneth W. Copeland, C. Jinshong Hwang. Third-Generation Web Applications, Full-Service Intranets, EDI: The Fully Integrated Business Model for Electronic Commerce. In *Proceedings of the INET '97 Conference*, Kuala Lumpur, Malaysia, Internet Society, June 1997. On-line at http://www.isoc.org/inet97/proceedings/C5/C5_2.HTM.
- [Descartes00] Alligator Descartes, Tim Bunce. *Programming the Perl DBI*. O'Reilly & Associates, February 2000.
- [Drepper96] Ulrich Drepper. Internationalization in the GNU project. Technical paper.
- [EC01] European Commission, DG Information Society. Web-based Survey on Electronic Public Services. October 2001. Available at http://europa.eu.int/information_society/eeurope/egovconf/documents/pdf/eeurope.pdf.
- [ESIS01] European Commission, Information Society Promotion Office. European Survey of Information Society Projects and Actions. March 2001. Available at <http://www.eu-esis.org/>.
- [Felix97] Ondřej Felix. PO76 DATA Management — koncept, produkty, průmysl a lidé. Slides for course taught at FI MU. 1997–2002.
- [Fielding94] Roy Thomas Fielding, Larry Masinter, Mark P. McCahill. Uniform Resource Locators. RFC 1738. December 1994. Available at <ftp://ftp.rfc-editor.org/in-notes/rfc1738.txt>.
- [Fielding98] Roy Thomas Fielding, Emmet James Whitehead, Jr., Kenneth M. Anderson, Gregory Alan Bolcer, Peyman Oreizy, Richard Newton Taylor. “Web-Based Development of Complex Information Products”. *Communications of the ACM*, Volume 41, Issue 8, August 1998, pp. 84–92.

- [Fielding99] Roy Thomas Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616. June 1999. Available at <ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt>.
- [Fielding00] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine, CA, 2000. Available at <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [FitzGerald01] Jerry FitzGerald, Alan Dennis. *Business Data Communications and Networking*. John Wiley & Sons Inc., 7th Edition, August 2001.
- [Freier96] Alan O. Freier, Philip Karlton, Paul C. Kocher. The SSL Protocol. Version 3.0. Internet-Draft. Internet Engineering Task Force (IETF), November 1996. Available at <http://wp.netscape.com/eng/ssl3/draft302.txt>.
- [Global02] Global Reach. Global Internet Statistics. Web resource at <http://glreach.com/globstats/>.
- [Goldfarb91] Charles F. Goldfarb. *The SGML handbook*. Clarendon Press, February 1991.
- [Harkins03] Perrin Harkins. Choosing a Templating System. On-line at <http://perl.apache.org/docs/tutorials/tmpl/comparison/comparison.html>.
- [Hiatto2] Edward Hiatt, Robert Mee. “Going Faster: Testing The Web Application”. *IEEE Software*, Volume 19, Issue 2, March 2002, pp. 60–65. On-line at http://www.edwardh.com/ieee_article/goingfaster.pdf.
- [HTML] World Wide Web Consortium (W3C). W3C HTML Home Page. Web resource at <http://www.w3.org/MarkUp/>.
- [HTTP] World Wide Web Consortium (W3C). Hypertext Transfer Protocol Overview. Web resource at <http://www.w3.org/Protocols/>.
- [ISMU02] Informační systém Masarykovy univerzity v roce 2002. Výroční zpráva o provozu a vývoji. Annual report for Information System of Masaryk University, at <http://is.muni.cz/clanky/vyrocka2002.pl>.
- [Jacobson92] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Publishing Company, June 1992.
- [JDBC] Sun Microsystems, Inc. JDBC Technology. Web resource at <http://java.sun.com/products/jdbc/>.

- [Kline00] Kevin Kline. *SQL in a Nutshell*. O'Reilly & Associates, December 2000.
- [Kohouteko1] Roman Kohoutek. Ověření přístupů při definici XSL transformací. Bachelor thesis.
- [Lindholm99] Tim Lindholm, Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley Publishing Company, 2nd Edition, April 1999.
- [Mockapetris87] Paul V. Mockapetris. Domain Names — Concepts and Facilities. RFC 1034. USC/Information Sciences Institute, November 1987. Available at <ftp://ftp.rfc-editor.org/in-notes/rfc1034.txt>.
- [NCSA95] NCSA Software Development Group. The Common Gateway Interface Specification. Web resource at <http://hoohoo.ncsa.uiuc.edu/cgi/interface.html>.
- [NOIE01] National Office for the Information Economy. Advancing with e-commerce. NOIE, September 2001. Available at <http://www.noie.gov.au/projects/CaseStudies/Ecommerce/index.htm>.
- [ODBC] Microsoft Corporation. Microsoft Open Database Connectivity (ODBC). Web resource at <http://msdn.microsoft.com/library/en-us/odbc/hdm/odbcstartpage1.asp>.
- [Ousterhout98] John K. Ousterhout. “Scripting: Higher-Level Programming for the 21st Century”. *IEEE Computer*, March 1998, pp. 23–30. Draft available at <http://home.pacbell.net/ouster/scripting.html>.
- [Pazdziora01a] Jan Pazdziora. Information Systems Development for Serving Environment With Rapidly Growing Numbers of Users. In *Proceedings of SSGRR 2001*, L'Aquila, Italy, Scuola Superiore G. Reiss Romoli, August 2001, pp. 21–25. On-line at <http://www.fi.muni.cz/~adelton/papers/ssgrr2001.ps>.
- [Pazdziora01b] Jan Pazdziora, Michal Brandejs, Iva Hollanová, Miroslava Misáková. Procesy reálného světa a přesnost dat v UIS. *RUFIS 2001: Role univerzit v budoucí informační společnosti: Sborník příspěvků*. Západočeská univerzita v Plzni, 2001. pp. 107–114. On-line at <http://www.fi.muni.cz/~adelton/papers/rufis2001.html>.
- [PHP] PHP Manual: A Simple Tutorial. On-line at <http://www.php.net/manual/en/tutorial.php>.
- [Postel81] Jon Postel (Ed.). Transmission Control Protocol — DARPA Internet Program Protocol Specification. RFC 793. USC/Information Sciences Institute, September 1981. Available at <ftp://ftp.rfc-editor.org/in-notes/rfc793.txt>.

- [Raghunathan97] Madhavarao Raghunathan, Gregory R. Madey. A Firm Level Framework for Electronic Commerce: An Information Systems Perspective. In *Proceedings of the Third Americas Conference on Information Systems*, Jatinder N. D. Gupta (Ed.), August 1997. On-line at <http://hsb.baylor.edu/ramsower/ais.ac.97/papers/raghuna.htm>.
- [Tapscott96] Don Tapscott. *The Digital Economy: Promise and Peril in the Age of Networked Intelligence*. McGraw-Hill Trade, January 1996.
- [Unicode] The Unicode Consortium. The Unicode Standard, Version 4.0.0. Defined by The Unicode Standard, Version 4.0, Addison-Wesley, Reading, MA, 2003, ISBN 0-321-18578-1. On-line at <http://www.unicode.org/versions/Unicode4.0.0/>.
- [URI] World Wide Web Consortium (W3C). Universal Resource Identifier. Web resource at <http://www.w3.org/Addressing/>.
- [Walloo] Larry Wall, Tom Christiansen, Jon Orwant. *Programming Perl*. O'Reilly & Associates, 3rd Edition, July 2000.
- [Walsh02] Norman Walsh, Leonard Mueller. *DocBook: The Definitive Guide*. O'Reilly & Associates, June 2002.
- [WML] Wireless Application Protocol Forum, Ltd. Wireless Markup Language. Version 2.0. Available from <http://www.wapforum.org/what/technical.htm>.
- [XHTML] World Wide Web Consortium (W3C). XHTML™ 1.0: The Extensible HyperText Markup Language. Second Edition. A Reformulation of HTML 4 in XML 1.0. August 2002. W3C Recommendation. Available at <http://www.w3.org/TR/2002/REC-xhtml1-20020801>.
- [Zwass98] Vladimir Zwass. Structure and Macro-Level Impacts of Electronic Commerce: From Technological Infrastructure to Electronic Marketplaces. Research paper at <http://www.mhhe.com/business/mis/zwass/ecpaper.html>.

Curriculum Vitæ

Jan Pazdziora

Education

- 1997: Master degree (Mgr.)
Informatics at Faculty of Informatics, Masaryk University in Brno
Thesis: Algoritmy řádkového a stránkového zlomu v počítačové sazbě (Linebreaking and pagebreaking algorithms in computer typesetting)
Graduated with honors
- May 1995 – September 1996: Teacher Training Program
Supported by Civic Educational Project and Jan Hus Educational Foundation
- 1995: Bachelor degree (Bc.)
Informatics at FI MU

Professional Experience

- 1998 – 2003: IT Analyst and Developer
Masaryk University
Project of Information System of Masaryk University
Electronic applications for student admission, payment processing; interconnections to other university information resources, real-time data transfers; document management system, including conversions of proprietary formats
Support for ECTS (European Credit Transfer System) adoption at university level, including modification of university regulations
Core system modules, applications of study agenda; analysis and design of database schema, database programming; database system administration
Problem solving
- 1995 – 1998: System and Network Administrator
Faculty of Informatics, Masaryk University
Unix servers and computer networks
Development of Faculty administration — on-line administrative agenda for credit-based study system
- 1992 – 1995: System Administrator and Tutor
Střední průmyslová škola stavební, Kudelova, Brno
PC, CAD systems; school management agenda

Refereed Conference Publications

- Jan Pazdziora. Efficient Multilingual Output for Web Components. In *DATAKON 2001: Proceedings of the Annual Database Conference*, Brno, Slovenská technická univerzita v Bratislave, October 2001, pp. 277–284.
- Jan Pazdziora. Converting Web Applications to Data Components: Design Methodology. In *The Twenty-Fifth Annual International Computer Software & Applications Conference (IEEE COMPSAC)*, Chicago, Illinois, IEEE Computer Society, October 2001, pp. 31–36.
- Jan Pazdziora, Michal Brandejs, Iva Hollanová, Miroslava Misáková. Procesy reálného světa a přesnost dat v UIS. In *RUFIS 2001: Role univerzit v budoucí informační společnosti: Sborník příspěvků*, Západočeská univerzita v Plzni, September 2001, pp. 107–114.
- Jan Pazdziora. Information Systems Development for Serving Environment With Rapidly Growing Numbers of Users. In *Proceedings of SSGRR 2001*, L'Aquila, Italy, Scuola Superiore G. Reiss Romoli, August 2001, pp. 21–25.
- Michal Brandejs, Iva Hollanová, Miroslava Misáková, Jan Pazdziora. In-house Developed UIS for Traditional University: Recommendations and Warnings. In *Proceedings of the 7th International Conference of European University Information Systems (EUNIS)*, Humboldt-University Berlin, March 2001, pp. 234–237.
- Michal Brandejs, Iva Hollanová, Miroslava Misáková, Jan Pazdziora. Administrative Systems for Universities Should Be More Than Just a Spreadsheet. In *Proceedings of the 7th International Conference of European University Information Systems (EUNIS)*, Humboldt-University Berlin, March 2001, pp. 84–87.
- Jan Pazdziora, Miroslava Misáková. Open Source Tools in University Operation. In *Proceedings of the IASTED International Symposia – Applied Informatics*, Innsbruck, IASTED/ACTA Press, February 2001, pp. 461–465.
- Jan Pazdziora. Využití XML rozhraní při tvorbě aplikací. In *Sborník SLT 2001*, Skalský dvůr, Konvoj, Brno, February 2001, pp. 153–159.
- Jan Pazdziora. Dokumentový server IS MU. In *Sborník příspěvků RUFIS 2000*, Vysoké učení technické v Brně, September 2000, pp. 90–93.
- Michal Brandejs, Iva Hollanová, Miroslava Misáková, Jan Pazdziora. Univerzitní IS: předsudky, pověry a realita. In *Sborník příspěvků RUFIS 2000*, Brno, Vysoké učení technické v Brně, September 2000, pp. 15–22.

Jan Pazdziora, Michal Brandejs. University Information System Fully Based on WWW. In *ICEIS 2000 Proceedings*, Stafford, UK, Escola Superior de Tecnologia do Instituto Politécnico de Setúbal, July 2000, pp. 467–471.

Michal Brandejs, Iva Hollanová, Miroslava Misáková, Jan Pazdziora. IS pro univerzitu ve třetím tisíciletí. In *UNIFOS 2000, univerzitné informačné systémy*, Slovenská poľnohospodárska univerzita, Nitra, May 2000, pp. 27–32.

Jan Pazdziora. Autentizovaný přístup a systém práv v IS MU. In *Sborník příspěvků konference RUFIS'99*, Vysoké učení technické v Brně, September 1999, pp. 113–120.

Jan Pazdziora. Zajištění efektivního přístupu k informacím v prostředí univerzity. In *Sborník příspěvků konference RUFIS'98*, Liberec, Technická univerzita v Liberci, September 1998, pp. 163–168.

Formal Presentations

Jiří Zlatuška, Michal Brandejs, Jan Pazdziora, Miroslava Misáková. University information system and institutional change. EAIR Forum 2002, Praha, September 2002.

Jan Pazdziora. Data Layer in Dynamic Applications and Presentation Formatting. YAPC::Europe 2001, Amsterdam, August 2001.

Jan Pazdziora. AxKit — integrated XML processing solution. YAPC::Europe 2001, Amsterdam, August 2001.

Jan Pazdziora. Docserver — using the network to overcome proprietary formats. YAPC::Europe 2000, London, UK, September 2000.

Jan Pazdziora. Administrativní server FI. Seminář SLT'98, Jevíčko, November 1999.

Jan Pazdziora. Informační systémy, výzkum a univerzity. Legislativa, komunikace, informační systémy, Nymburk, March 1999.

Teaching Activities

- DB Systems Seminar: P136 (2001 – 2002), PV136 (2003)
- Invited lectures about Perl, in Computer Network Services: P005 (1996 – 2001)
- Seminar tutor for Introduction to the C language: I071 (1999)

Other University Activities

- 1999: Participant of Salzburg Seminar Symposium: Globalization and the Future of the University
- 1998 – 2000: Member of the Academic Senate of Masaryk University
- 1997 – 2000: Member of the Academic Senate of Faculty of Informatics, MU
- September 1996: Academic stay in the Rutherford Appleton Laboratory, Didcot, UK; supported by the Jan Hus Educational Foundation



Honors

- 1997: Rector's Honor List
- 1996: Dean's Honor List

Software

- Perl modules
 - DBD::XBase / XBase.pm
 - Cstools (cstocs, cssort)
 - Docserver
 - Font::TFM
 - TeX::DVI
 - TeX::Hyphen
 - MyConText
 - Tie::STDERR
 - DBIx::ShowCaller
 - Apache::OutputChain
- Localization
 - Czech and UCA collation for MySQL
 - ISO-8859-2 fonts for the GD library
- Numerous patches to open source software projects