XML, Servlets and JavaServer Pages™

an introduction

Florian Heinisch

Vienna University of Economics and Business Administration

2006

To my parents who have stood behind me steadfast and from whom I always have experienced endless support.

Contents

Table of Figures

Table of Listings

Tables of Tables

Acronyms

ANSI	American National Standards Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASF	Apache Software Foundation
ASP	ActiveServer Pages
BSF	Bean Scripting Framework
CDATA	Character Data
CDF	Channel Definition Format
CGI	Common Gateway Interface
CR	Carriage Return
CRUD	Create, Retrieve, Update and Delete
CSS	Cascading Style Sheet
DAO	Data Access Object
DLL	Dynamic Link Library
DNS	Domain Name Service
DTD	Document Type Definition
DTO	Data Transfer Object
EAR	Enterprise Archive
ECMA	European Computer Manufacturers Association
EL	Expression Language
FTP	File Transport Protocol
GIF	Graphic Interchange Format
GML	General Markup Language

GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transport Protocol
IBM	International Business Machines
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISO	International Organization for Standardization
J2EE	Java 2 Platform Enterprise Edition
J2SE	Java 2 Platform Standard Edition
JAR	Java™ Archive
JDK	J2SE Delelopment Kit
JSP	JavaServer Pages™
JSTL	JSP Standard Tag Library
JVM	Java Virtual Machine
LF	Line Feed
MathML	Mathematical Markup Language
MIME	Multipurpose Internet Mail Extension
MVC	Model-View-Controller
ooRexx	Object Rexx
PCDATA	Parseable Character Data
PDA	Personal Digital Assistant
PDF	Portable Document Format
PHP	Hypertext Preprocessor
PI	Processing Instructions

PIM	Personal Information Management
POJO	Plain Old Java Object
RFC	Request for Comments
RSS	Rich Site Summary
SGML	Standard General Markup Language
SMIL	Synchronized Multimedia Integration Language
SVG	Scalable Vector Graphics
TEI	Text Encoding Initiative
TLD	Tag Library Descriptor
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAR	Web Archive
WAS	Websphere Application Server
WML	Wireless Markup Language
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
XHTML	Extensible HyperText Markup Language
XLL	XML Link Language
XML	Extensible Markup Language
XPath	XML Path Language
XSL	Extensible Stylesheet Language
XSL-FO	XSL Formatting Objects
XSLT	XSL Transformations

1 Introduction

Originally, the World Wide Web was “a medium for the broadcast of read-only material” [Bern96]. The “read-only material” took the form of static HTML pages which did not support any form of user interaction. Although static HTML pages were adequate to provide rapid access to information, “service providers recognized the need for dynamic content” [ArBa04] as the Internet also began to be used for delivering services. Since then, various technologies (such as CGI scripts, ASP, PHP or Servlets) have been created to meet the need to provide dynamic content.

The primary goal of this thesis is to introduce the reader to the Extensible Markup Language (XML), Servlets and JavaServer Pages™ (JSP) which are technologies that allow to enhance the development of dynamic content.

The information given in this document is categorized into five parts. As each part separately deals with a specific topic the reader can simply skip any part he or she might be familiar with.

The first part introduces the Extensible Markup Language (XML). After a short introduction of the historical development of XML, XML basics such as the syntax and the structure of XML documents are explained. Finally, it is shown how to validate and display XML documents.

The second part provides a basic introduction to the Hypertext Transfer Protocol (HTTP).

The third part shall give the reader a basic understanding of servlets. Therefore, the servlet's structure and life-cycle are described. Additionally, the provided theory about servlets is highlighted with the use of examples. For that purpose, the deployment of servlets is described so the reader gains the crucial knowledge needed to build Web applications with Java. As servlets strongly resemble the Common Gateway Interface (CGI), this part concludes with a comparison of those two technologies.

The fourth part deals with the discussion of JavaServer Pages™ (JSP). Since JSPs are considered to be an extension of servlets, it is highly recommended

reading first the chapter on servlets for a thorough understanding of this topic in the case the reader is not familiar with servlets. This chapter's purpose is to expound the JSP's life-cycle and the main components a JSP may consist of. In order to illustrate that JSPs can include scripts written in none-Java programming languages, a short digression is made into the Bean Scripting Framework (BSF). The development of script-less JSPs is explained at the end.

The discussion on XML, Servlets and JavaServer Pages™ concludes with a roundup and outlook.

In order to demonstrate how to build powerful Web applications by combining XML, Servlets and JavaServer Pages™, the author developed a simple shopping cart Web application that extensively uses these three technologies. For that reason this Web application is explained in depth in this paper's fifth part, the appendix. As the Web application is built upon a Model-View-Controller (MVC) framework, essential theory on this topic is explained. Furthermore, the Web application's architecture is revealed and a step-by-step installation-instructions is provided so that the reader should be able to easily install the Web application on her or his machine.

2 XML

This chapter will provide an introduction to XML. It starts with a discussion about markup languages, XML's background and development. Next, XML's syntax will be described before moving on to the explanation of how XML documents can be validated. At the end of the chapter, two different ways to format XML documents will be discussed.

2.1 Overview

The discussion about XML starts with the definition of markup languages and a brief introduction to SGML. Next, the reader will be given an explanation about HTML's shortcomings and an abstract about XML's goals and its development.

2.1.1 Definition

XML stands for eXtensible Markup Language. XML is a restricted form (i.e. subset) of SGML. By construction, XML documents are conforming SGML documents [BrPa00].

2.1.1.1 Markup Language

Before delving into the discussion of markup languages the terms markup and markup languages need to be defined.

The Text Encoding Initiative (TEI) defines markup “as any means of making explicit an interpretation of a text” [SpBu94]. In this sense all texts are encoded, for example by punctuation marks or spaces between words. In context of automated text processing, the TEI states that “encoding a text for computer processing is in principle [...] a process of making explicit what is conjectural or implicit” [SpBu94].

A markup language is “a set of markup conventions used together for encoding texts” [SpBu94]. A markup language specifies [SpBu94]:

- what markup is allowed,

- what markup is required,
- how markup is to be distinguished from text,
- and what the markup means.

2.1.2 SGML

In 1969 Charles Goldfarb together with Edward Mosher and Raymond Lorie, invented the General Markup Language (GML) in the context of an IBM research project as a means of sharing documents. Charles Goldfarb introduced with GML “the concept of a formally-defined document type with an explicit nested element structure” [SGML90]. Having completed the development of GML, Goldfarb went on to research document structures and created additional concepts (such as concurrent document types).

In 1978, the American National Standards Institute (ANSI) established a committee which had the task to develop GML and Goldfarb's research results into a standard. The first industry standard of SGML was released in 1983. Finally, after further reviewing and development of SGML it became an ISO standard[1] in 1986. [John99], [SpBu94].

SGML is the mother tongue of all markup languages. It is a standard for describing the structure and content of machine-readable information. The TEI describes SGML as “an international standard for the description of marked-up electronic text” [SpBu94]. Furthermore, SGML is a metalanguage. A metalanguage is defined as a “a means of formally describing a language” [SpBu94] (in this case a markup language).

Jon Busak states that SGML permits “documents to describe their own grammar” [Bosa97]. With SGML it is possible to specify the tag set used in the document as well as the structural relationships that those tags describe.

[1] ISO (International Organization for Standardization, situated in Geneva): ISO 8879:1986(E). Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML). First edition -- 1986-10-15. [BrPa00].

2.1.2.1 Characteristics of SGML

There are three characteristics of SGML that will be discussed shortly [SpBu94]:

1. descriptive markup,
2. document type concept,
3. data independence.

2.1.2.1.1 Descriptive Markup

SGML uses descriptive markup codes which provide names to categorize parts of a document. For example, a markup code like <p> identifies a part of a document's text and declares that the item following the markup code is a paragraph. In contrast to procedural markup which defines “what processing is to be carried out at particular points in a document” [SpBu94], descriptive markup allows that “the same document can readily be processed by many different pieces of software, each of which can apply different processing instructions to those parts of it which are considered relevant” [SpBu94]. For example, one program could retrieve specific content of the document to store it in a database while another could format the document for printing.

2.1.2.1.2 Document Type Concept

SGML introduced the concept of a document type, and as a result a document type definition (DTD). In SGML, “documents are regarded as having types [...]. The type of a document is formally defined by its constituent parts and their structure” [SpBu94]. For example, the definition of a diploma thesis might be that it consists of a title, an author, an introduction followed by chapters containing paragraphs and a summary. Such a logical structure is defined in a document type definition (DTD).

The significant benefit of a document type definition is that “a special purpose program (called a parser) can be used to process a document claiming to be of a particular type and check that all the elements required for that document type are indeed present and correctly ordered” [SpBu94]. Consequently, different documents that are of the same type can be processed in a uniform way.

2.1.2.1.3 Data Independence

A fundamental objective of SGML is to “ensure that documents encoded according to its provisions should be transportable from one hardware and software environment to another without loss of information” [SpBu94]. For this purpose, “SGML provides a general purpose mechanism for string substitution” [SpBu94]. string substitution is “a simple machine-independent way of stating that a particular string of characters in the document should be replaced by some other string when the document is processed” [SpBu94].

There are two obvious applications for string substitution [SpBu94]:

1. Provide for the consistency of nomenclature.
2. Compensate for “the notorious inability of different computer systems to understand each other’s character sets [...] by providing descriptive mappings for non-portable characters” [SpBu94].

2.1.2.2 SGML Applications

According to the W3C, “each markup language defined in SGML is called an SGML application” [RaLe99]. An SGML application consists of [RaLe99]:

1. an SGML declaration[2],
2. a document type definition,
3. a specification that describes the semantics to be ascribed to the markup,
4. a document instance that contains both data (i.e. content) and markup.

To summarize, SGML makes it possible to define ones own formats for ones own documents. However, full SGML contains many optional and complex features that are not needed for Web applications.

[2] An SGML declaration specifies the character set, the codes used for SGML delimiters (e.g. “<, >, /”) and the length of identifiers [SpBu94].

2.1.3 HTML

The Hypertext Markup Language (HTML) was designed for the interchange of hypertext as a data format to be transmitted via the Hypertext Transfer Protocol (HTTP, see chapter 3, p. 57). HTML was invented in 1991 by Tim Berners-Lee in Geneva at the European Laboratory for Particle Physics (CERN), Switzerland [Bern96], [Münz98].

HTML “is an SGML application” [RaLe99], consequently HTML documents are SGML documents. As an SGML application, HTML also has an SGML declaration, a document type definition and a specification. At the time of writing this thesis, the latest HTML specification published by the W3C was “HTML 4.01”[3]. This specification includes an SGML declaration[4] and three document type definitions[5] (the DTDs vary in the elements they support) [RaLe99].

HTML's document type definitions define “a fixed set of document elements with markup” [Culs97] that lets the developer describe simple documents containing headings, paragraphs, illustrations, lists, tables, etc. A major advantage of HMTL is “its built-in support for hypertext and multimedia” which enables “the construction of easy and intuitive user interfaces[6] for accessing published information” [Culs97]. Other advantages include “the ability to exchange information between different computer systems and applications through the use of standard formats and protocols, and the power of hypertext to organize a set of documents to be searched, accessed and consulted interactively” [Culs97].

Here are the following advantages resulting from HTML’s simplicity [Claß1a]:

1. It is quick and easy to learn.
2. It can be viewed with minimal client requirements (e.g. Web browsers).
3. It is well suited for describing the visual appearance of a human-readable document, including text and images.

---

3 W3C defined XHTML as HTML's successor (see chapter 2.1.7.2, p. 24).
4 To view the SGML declaration please refer to [HTML99a].
5 To view the DTDs please refer to [HTML99b].
6 For example such an user interface is built in a Web browser [Culs97].

On the other hand, because of its simplicity there are shortcomings as follows [Bosa97], [Frete98]:

1. HTML is only a presentation technology: “HTML does not necessarily reveal anything about the information to which HTML tags are applied” [Frete98]. For example `<h1>Introduction</h1>` has a predictable appearance in a Web browser but it does not reveal anything about the content itself.

2. As it uses a fixed set of well-defined tags, it is not extensible to allow user-defined tags[7].

Despite of HTML's advantages, HTML seems “to have reached the limit of its usefulness as a way of describing information” [Culs97]. The inherent simplicity of HTML has been a decisive factor for the success of HTML. But as HTML has a limited set of tags it offers only one way to describe documents which imposes a severe limitation to the description of professional documents [Culs97].

Although the use of SGML could compensate for HTML's deficiencies, “the inherent complexity of SGML limits its adoption in full-scale applications by a large number of non-expert users on the Internet” [Culs97]. Developers were looking for “a way of combining the richness of SGML with the simplicity of HTML for publishing and accessing documents online” [Culs97].

This is where XML comes in.

2.1.4 XML is Born

In the summer of 1996 the World Wide Web Consortium (W3C) has created an SGML Working Group (originally called the "SGML Editorial Review Board") to formulate a set of specifications in order to make it easy and straightforward to use the beneficial features of SGML on the WWW [John99].

SGML had its passionate supporters as well as its equally passionate detractors. In order to emphasize its difference from HTML and not to burden the new technology’s name with SGML’s history, the working group decided to turn

[7] Only the W3C could extend HTML's set of tags properly [Frete98].

SGML into something new and named it Extensible Markup Language [Frete98].

The working group members quickly set a schedule in which to specify the features of XML. They planned their work in three phases [Frete98]:

1. XML: the syntax itself.
2. XLL (Extensible Link Language): the linking semantics of XML.
3. XSL (Extensible Stylesheet Language): the presentation of XML.

Finally, the XML 1.0 standard was approved and published by the W3C on February 10th, 1998 [Fret98].

2.1.5 XML's Goals

XML aims to be a simple, open, self describing format, capable of expressing varied types of information.

The XML specification names ten design goals for XML [BrPa00]:

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

2.1.6 XML Principles

XML is a subset (i.e. a restricted form) of SGML whose “goal is to enable generic SGML to be served, received and processed on the Web [...]. XML has been designed for ease of implementation and for interoperability with both SGML and HTML” [BrPa04a]. In other words, XML improves “the functionality of the Web by providing more flexible and adaptable information identification” [Flyn02].

XML is called extensible because it is not a fixed format (as HTML is). XML is a metalanguage (see chapter 2.1.2, p. 17) which lets one design one's own customized markup languages “for limitless different types of documents” [Flyn02].

According to W3C, “XML documents are conforming SGML documents” [BrPa04a].

2.1.7 XML Usage

XML can be used for various applications. Trying to set up an almost complete list of specific XML applications would by far burst the scope of this paper. Therefore, the potential uses of XML will be discussed in a broader and more general sense, that is to say inter-application data exchange. Furthermore, a brief discussion on XHTML will be provided at the end of this sub chapter.

2.1.7.1 Inter-application Data Exchange

In organizations there have been a lot of different applications which could not easily communicate with each other. Consequently, much time and effort “is wasted on duplicate data entry and data integrity checking” [Claß01b].

With the Internet there exists a “world-wide infrastructure for communicating between applications [...] and users” [Claß01b] (e.g. for exchanging XML documents between different applications).

As XML documents “represent both metadata and data” [GoPr05] applications can process the XML document and either

- parse the XML document in order to extract the original data,

- or render the XML document in order to present it in a physical medium that humans can perceive,
- or process the XML document as plain text without parsing (e.g. cut parts of the XML document and paste it into other XML documents) [GoPr05].

Hence, there “is an endless spectrum of application opportunities” [GoPr05] as XML provides an application- and platform-independent way of data exchange.

2.1.7.2 Excursus: XHTML

In January 2000, W3C released the recommendation for XHTML 1.0 which was “a reformulation of HTML 4 as an XML 1.0 application” [PeAu02]. XHTML in general is referred to as “a family of current and future document types and modules that reproduce, subset, and extend HTML, reformulated in XML” [Pemb06]. All documents that belong to the XHTML family are XML documents. XHTML is intended to be the successor to HTML.

As XHTML documents are conforming XML documents, there are significant benefits:

- XHTML documents can be “readily viewed, edited, and validated with standard XML tools” [Pemb06].
- Due to the fact that XHTML is an XML application, it is “relatively easy to introduce new elements or additional element attributes” [PeAu02].
- XHTML documents can “include bits of other markup languages” [Pemb04], such as MathML[8], SMIL[9] or SVG[10].

Furthermore, as “alternate ways of accessing the Internet are constantly being introduced” [PeAu02] XHTML is designed for “general user agent[11] interoperability” [PeAu02]. Consequently, Websites in XHTML shall be accessible with

[8] MathML stands for Mathematical Markup Language. For further information please refer to the W3C specification available at [CaIo03].
[9] SMIL stands for Synchronized Multimedia Integration Language. For further information please refer to the W3C specification available at [BuZu05].
[10] SVG stands for Scalable Vector Graphics. For further information please refer to the W3C specification available at [FeJa03].
[11] A user agent is “any software that retrieves and renders Web content for users” [JaGu02].

non-PC devices such as PDAs[12], mobile phones, television or even refrigerators [Pemb04].

As a detailed discussion about XHTML is outside the scope of this thesis, the reader is asked to refer for further information to W3C's Website on XHTML available at [Pemb06].

2.2 XML Basics

This sub chapter provides an introduction to the structure of XML documents as well as a discussion about markup that can be used with XML.

2.2.1 Simple Example

The following text show in listing 2-1 represents a simple XML document which describes this paper. All examples presented in this chapter are available for download from the author's Web site [Hein06].

```
<?xml version="1.0"?>
<!-- A simple document -->
<thesis>
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build to a web
    application.
  </abstract>
</thesis>
```

Listing 2-1: A simple XML-document "thesis.xml".

2.2.2 Structure of an XML document

According to the XML specification, "each XML document has both a logical and a physical structure" [BrPa04a]. Physically, an XML document "may consist of one or many storage units" [BrPa04a]. These storage units are called entities

[12] A personal digital assistant (PDA) is a battery operated pocket PC which is primarily used for mobile personal information management (PIM) [cf. HaNe02, 65].

(e.g. such an entity can be a file). Each XML document “has one entity called the *document entity*, which serves as the starting point for the XML processor[13] and may contain the whole document” [BrPa04a]. In other words, the document entity is the entity that “contains the main body of the document” [Brow06]. Entities will be discussed in detail in chapter 2.2.3.7 (p. 30).

The logical structure of an XML document is defined by the elements and by the relationships between those elements within the XML document. Elements will be discussed in more detail in chapter 2.2.3.1 (p. 27).

Figure 2-1 illustrates the logical structure of the XML document `thesis.xml` (see listing 2-1, p. 25).

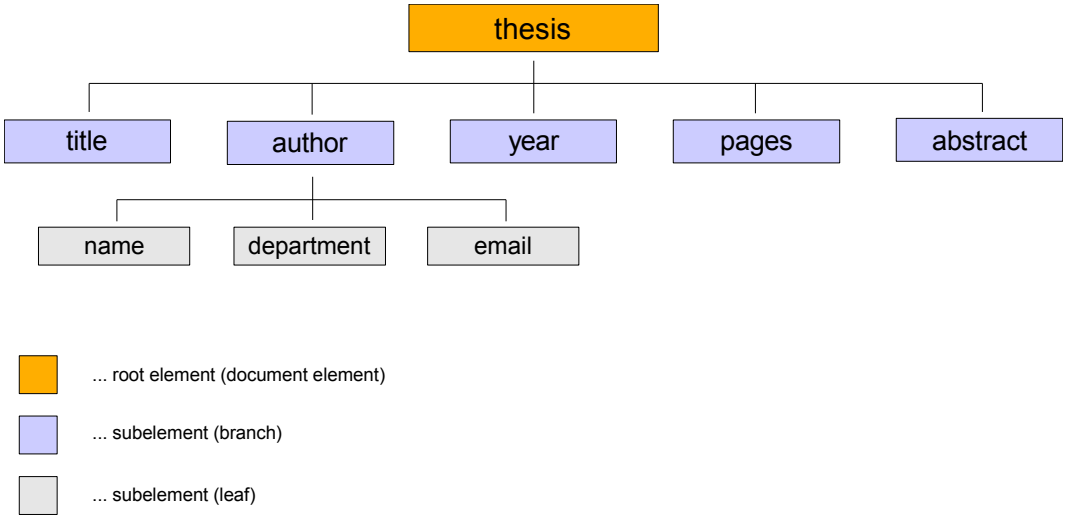

Figure 2-1: Logical (or tree) structure of the XML document “thesis.xml”.

The element that contains all the other elements is known as the root element (e.g. `<thesis>`, see listing 2-1, p. 25). The root element is also referred to as the *document element* because it holds the entire logical document within it. The elements that are contained within the root element are called its subelements (e.g. `<author>`). They may contain subelements themselves. If they do, they are called branches (e.g. `<name>`). If they do not, they are leaves (e.g. `<title>XML, Servlets &amp; JavaServer Pages</title>`) [Gold00].

[13] An XML processor (also called parser) is referred to as a software module that “is used to read XML documents and provide access to their content and structure” [BrPa04a].

2.2.3 Markup

XML documents consist only of character data (i.e. content) and markup. According to the XML specification, “markup takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white space that is at the top level of the document entity” [BrPa04a]. Consequently, all text that is not markup constitutes the XML document's character data.

2.2.3.1 Elements

Elements are pairs of start (e.g. `<title>`) and end tags (e.g. `</title>`). They identify the content they surround. The start tag consists of an opening angle bracket “<” followed by a name. The start tag may contain attributes (see chapter 2.2.3.1.1, p. 28) which are separated by white space. The start tag's end is marked by a closing angle bracket “>” (e.g. `<title>`). The end tag consists of an opening angle bracket “<” followed by a slash “/”, the element's name and a closing bracket “>”. The end tag must exactly match the start tag's name [Ray01].

Empty elements are elements that do not enclose any content. They can be abbreviated with a “/” before the closing angle bracket “>”: `<EmptyTag/>`.

An element's name has to start with a letter or an underscore “_”. The name “can contain any number of letters, numbers, hyphens, periods, and underscores” [Ray01]. Between the opening angle bracket “<” and the element's name there must be no space. However, extra space anywhere else in the element's tag is allowed. Following example

```
<author     id="9603480"     >
            Florian Heinisch
</author                     >
```

is a conforming XML element.

In XML, all space characters are preserved by default. This includes the white-space characters space, tab and new line [Ray01].

Furthermore, XML tags are case sensitive, so for example `<emtpytag/>`, `<Emptytag/>` and `<EmptyTag/>` are three different elements.

XML elements must be nested properly: the end tag must be positioned after the start tag and an element's start and end tag must both reside within the same parent element. For example, this is not allowed as the end tag of the `<title>` element is positioned outside its parent element:

```
<thesis>
  <title>XML, Servlets &amp;amp; JavaServer Pages</thesis>
  </title>
```

2.2.3.1.1 Attributes

Attributes are name-value pairs that can be added to a start tag as in `<author id="9603480">Florian Heinisch</author>`. They provide extra information about elements (this is the element `author` with the attribute `id` which has the value `9603480`). An element can have limitless attributes as long as each attribute has a unique name. Attributes are separated by white space and can be in any order. An attribute's value has to be in single or in double quotes [Gold00], [Ray01].

2.2.3.2 Comments

Comments begin with `<!--` and end with `-->` (see listing 2-1, p. 25). Comments may appear anywhere in a XML document outside other markup [BrPa04a].

2.2.3.3 CDATA Sections

Character data (CDATA) sections are used “to escape blocks of text containing characters which otherwise would be recognized as markup” [BrPa04a] (i.e. the content of CDATA section will not be interpreted by the XML-processor). CDATA sections begin with the string “`<![CDATA[`” and end with the string “`]]>`”. The only character data that cannot be placed inside a CDATA section is the ending delimiter “`]]>`”. CDATA sections cannot be nested [BrPa04a].

2.2.3.4 Processing Instructions

Processing Instructions (PI) are used to provide an application with information. PIs are passed to the application using the parser. PIs begin with "<?" and end with "?>". A PI is used to identify the application to which the instruction is directed and has to be placed after the "<?". E.g. `<?word textfile="chapter_3.doc" ?>`

Processing instructions beginning with "xml" or "XML" have been reserved for standardization in the XML Version 1.0 specification and onwards. PIs must not be nested [Cagl00].

2.2.3.5 XML Declaration

The XML specification for XML 1.0 [BrPa04a] states that an XML document *should* (but does not have to) begin with an XML declaration. The XML specification for XML 1.1 [BrPa04b] states that an XML *must* begin with an XML declaration. Therefore, an XML document without a declaration is an XML 1.0 document.

The XML declaration starts with "<?xml", followed by the required property definition `version` and two optional property definitions `encoding` and `standalone`. The declaration ends with "?>" [BrPa04a]. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

These three properties can be set as follows [Ray01], [Ogbu04]:

1. `version`: Sets the version number. It is used to tell the XML processor which version to use. Possible values are "1.0" or "1.1", e.g. `version="1.0"` (see listing 2-1, p. 25).

2. `encoding`: Sets the character encoding used in the XML document. If the encoding is not set, the XML processor assumes that the encoding is UTF-8 or UTF-16.

3. `standalone`: indicates to the XML processor if there are any other files (which contain external markup) to load. Possible values are "`yes`" or "`no`".

If the property `encoding` is set, it has to follow the property `version`. If the property `standalone` is set, it has to follow the last property definition [Ogbu04].

2.2.3.6 Document Type Declaration

The document type declaration is optional markup that – if present – must be placed between the XML declaration and the root element. The document type declaration contains the *document type definition* (DTD, see chapter 2.3.1, p. 35) which can be defined in the *internal subset* and/or an *external subset* [Bray98]. The DTD for an XML document "consists of both subsets taken together" [BrPa04a].

The document type declaration starts with "`<!DOCTYPE`" followed by the root element. If there is an external subset the document type declaration should point to, the root element is followed by a special external entity declaration. If there is an internal subset, the internal subset begins with a opening square bracket "`[`" and ends with a closing square bracket "`]`". The document type declaration ends with "`>`" [BrPa04a]. Examples are provided in listing 2-3, p. 38 and listing 2-4, p. 39.

2.2.3.7 Entities

As mentioned in chapter 2.2.2, "an XML document can be divided in sections called entities" [Brow06]. Entities can be thought of as placeholders for content which are declared once and then can be referenced in the XML document [Ray01].

The two major entity types are *general entities* and *parameter entities* as they "use different forms of reference and are recognized in different contexts" [BrPa04a] (see chapter 2.2.3.8). A general entity and a parameter entity "with the same name are two distinct entities" [BrPa04a]. A general entity can be a

parsed or unparsed entity, whereas a parameter entity can be a parsed entity only [BrPa04a]:

- Parsed entity: consists of character data. An XML processor “locates, decodes, and imports the contents of each parsed entity as replacement text that replaces references to that entity” [Brow06].
- Unparsed entity: “contains non-XML data of any kind” [Brow06] (e.g. graphics or sound files) and will not be parsed by the XML processor. An XML document can refer to the unparsed entity but cannot contain the entity itself. The XML processor does not replace the reference to the unparsed entity. The XML processor just “make the identifiers for the entity [...] available to the application” [BrPa04a].

Furthermore, an entity is either an internal or an external entity [Brow06]:

- External entity: if the entity's declaration identifies the entity's replacement text by its location or if the entity is unparsed.
- Internal entity: If the entity's declaration includes its replacement text.

To summarize, these are the possible combinations of characteristics for entities [Bray98]:

- internal parsed general entity,
- internal parsed parameter entity,
- external parsed general entity,
- external parsed parameter entity,
- external unparsed general entity.

An entity has to be declared either in the internal or in an external subset before it can be referenced in an XML document. If the XML processor parses an entity reference that has not been declared, the XML processor cannot import the replacement text. Such an error prevents the XML document from being well-formed (see chapter 2.3, p. 34) [Ray01].

As a detailed explanation of all possible entities exceeds the limits of this thesis, the discussion will concentrate on general parsed entities. For further information concerning unparsed and parameter entities please refer to [BrPa04a].

2.2.3.8 General Entities

General entities are used “at the level of or inside the root element of an XML document” [Ray01].

General entities are referenced by an ampersand “`&`”, followed by the entity name and a semicolon “`;`”, e.g.: `&department;`

2.2.3.8.1 Internal Parsed General Entity

Internal parsed general entities are declared in the internal subset of the document type declaration (see chapter 2.2.3.6, p. 30) as follows [Watt03]:

```
<!ENTITY name "entity_value">
```

For example, the character data of the element `<department>` of the XML document `thesis.xml` (see listing 2-1, p. 25) could be declared as an entity and then be referenced:

```
<!DOCTYPE thesis [
  <!ENTITY department "Information Systems and New Media">
]>
<department>&department;</department>
```

2.2.3.8.2 External Parsed General Entity

An external parsed general entity is “useful for creating a common reference that can be shared between multiple documents” [Watt03], e.g. to insert repetitious information. Consequently, any changes concerning the repetitious information only need to be defined at a single place (i.e. where the entity is defined) and not at every place where it is used.

An external parsed general entity can be referenced by[14] [Brpa04a]:

```
<!ENTITY name SYSTEM "system-identifier">
```

[14] The XML specification also defines a public identifier to declare an external entity which will not be discussed in this paper. For further information, please refer to [BrPa04a].

The system identifier is an URI which is “used to retrieve the entity” [BrPa04a]. For example, the character data of the element `<abstract>` of the XML document `thesis.xml` (see listing 2-1, p. 25) could be stored in an external text file `abstract.txt`:

```
<!ENTITY abstract SYSTEM "http://www.heinisch.cc/thesis/abstract.txt">
```

Consequently, the entity `abstract` can be referenced by:

```
<abstract>&abstract;</abstract>
```

2.2.3.8.3 Predefined Entities

Additional to explicit declared general entities, in XML there are *predefined entities*. As discussed in chapter 2.2.3, the left angle bracket “<” and the ampersand character “&” identify the beginning of markup. These characters are therefore reserved and are interpreted as characters introducing markup. In order to be able to insert these characters as content into an XML document, entities are used to represent these special characters.

XML defines five predefined entities, that have to be recognized by all XML processors (whether these entities are declared are not), as shown in table 2-1 [BrPa04a].

Entity	Entity name	Replacement text
Left angle bracket (<)	lt	&#60;
Right angle bracket (>)	gt	>
Ampersand (&)	amp	&#38;
Single quote or apostrophe (')	apos	'
Double quote (")	quot	"

Table 2-1: Predefined entities.

Please note that the characters “<” and “&” in the declarations of `lt` and `amp` “are doubly escaped to meet the requirement that entity replacement be well-formed” [Bray98].

2.2.3.8.4 Character References

In addition to entity references there are character references which are similar to entities references in the way that they represent character data. Character references are intended to allow the insertion of a specified Unicode standard character directly into an XML document. This is essentially useful if a character is not available on the keyboard or not portable across applications and operating systems.

Character references can be inserted by the decimal or hexadecimal representation of the Unicode character number [Kamt00]:

- Decimal: the character reference begins with “`&#`”, followed by the digits up to the terminating “`;`”.
  E.g. the Greek small letter alpha “α” is inserted by the character reference `&#945;`

- Hexadecimal: the character reference begins with “`&#x`”, followed by the digits and letters up to the terminating “`;`”.
  E.g. the Greek small letter alpha “α” is inserted by the character reference `&#x3b1;`

2.3 Well-Formed vs. Validated XML documents

The XML document `thesis.xml` (see listing 2-1, p. 25) is what is referred to as a well-formed XML document. A well-formed XML document is “syntactically correct” [Sun06a]. In order to achieve the status of well-formed, an XML document must obey the following rules [SpBu04], [Ray01], [BrPa04a]:

- an XML document must contain one or more elements,

- each element must be nested in the root element,

- all rules concerning elements and attributes discussed in chapter 2.2.3.1 (p. 27),
- the isolated markup up characters “<” and “&” may not be placed in an element's content.

A well-formed XML document can be parsed by any XML processor [Brow06].

Additional to the rules for well-formedness, one can define “custom rules” for XML documents (e.g. how elements and attributes have to be named or how elements have to be nested). An XML document in which such additional criteria have been checked is referred to as a *valid* XML document. This is essentially useful because it provides the possibility to confirm that XML marked up text follows a predetermined structure (i.e. it can be validated whether an XML document is of a specific type, see chapter 2.1.2.1.2, p. 18). Only then the meaning of the markup used in an XML document can be shared, be it for human or application consumption [Chas03], [SpBu04].

In XML, the criteria for successful validation can be formally stated in a Document Type Definition (DTD) or in an XML Schema [SpBu04].

2.3.1 Document Type Definition

The document type definition (DTD) is used to define all markup languages. With a DTD, it is possible to define the logical structure of an XML document and to specify a content model for each element [Gorm98].

In employing a DTD, it is possible to define [StSa00]:

- elements and their attributes,
- nesting (which elements are allowed to contain other elements),
- elements’ sequence (i.e. the order of elements),
- whether certain elements are optional or required,
- whether certain elements can occur multiple times,

- whether elements can contain character data (non-markup),
- default values or fixed values for attributes.

If an XML document includes or references a DTD, validating XML parsers can detect [StSa00]:

- missing elements (i.e. tags) or attributes,
- misspelled elements or attributes,
- improper nesting of elements,
- incorrect ordering of elements,
- incorrect attribute values from enumerated sets,
- any elements or attributes that are not defined in the DTD.

2.3.1.1 DTD Syntax

The syntax for XML DTDs is based on the syntax of SGML DTDs. However, XML DTDs inherited a limited syntax from SGML in order “to simplify the syntax and to make it easier to write processing software” [Flyn06]. The omitted features will not be discussed in this thesis. For further information please refer to [BrPa04a], [Clar97], [Flyn06].

Listing 2-2 shows a DTD for the XML document `thesis.xml` (see listing 2-1, p. 25).

```
<!ELEMENT thesis (title, author, year, pages, abstract)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name, department, email)>
<!ATTLIST author
          id CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT department (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT pages (#PCDATA)>
<!ELEMENT abstract (#PCDATA)>
```

Listing 2-2: DTD for the XML document “thesis.xml”.

Listing 2-2 contains nine *element declarations* and one *attribute-list declaration*. An element declaration begins with "<!ELEMENT" followed by the name of the element which is being declared and a content model. The declaration ends with ">". The content model is enclosed in parentheses and specifies what "may legitimately be contained within it" [SpBu04].

An attribute-list declaration begins with "<!ATTLIST" followed by the concerned element's name. For each attribute that needs to be declared for the specific element, the attribute's name, the attribute's data type and a statement which specifies how an XML processor "should interpret the absence of the attribute" [SpBu04] have to be declared. The declaration ends with ">" [BrPa04a].

This DTD shown in listing 2-2 is interpreted as follows:

- !ELEMENT thesis defines the thesis element as having five elements: title, author, year, pages and abstract. These elements have to be placed in the listed order.
- !ELEMENT title defines the title element to be of the type "#PCDATA" (the same applies to name, department, email, year, pages, abstract). #PCDATA is an abbreviation for parsed character data. It means that "the element being defined may contain any valid character data" [SpBu04].
- !ELEMENT author defines that the author element must nest three elements: name, department and email (in the listed order).
- !ATTLIST author defines that the element author must include an attribute called id. The attribute's data type is CDATA which implies that the attribute's value "may contain any valid character data, including spaces or punctuation marks" [SpBu04]. The last piece of information #REQUIRED "means that the attribute MUST always be provided" [BrPa04a].

As discussed in chapter 2.2.3.6 (p. 30), DTDs can be defined in the internal and/or external subset. Listing 2-3 shows the DTD from listing 2-2 (p. 36) being placed in the internal subset.

```
<?xml version="1.0"?>
<!DOCTYPE thesis [
  <!ELEMENT thesis (title, author, year, pages, abstract)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (name, department, email)>
  <!ATTLIST author
            id CDATA #REQUIRED>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT department (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
  <!ELEMENT pages (#PCDATA)>
  <!ELEMENT abstract (#PCDATA)>
]>
<!-- A simple document -->
<thesis>
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build a web application.
  </abstract>
</thesis>
```

Listing 2-3: "thesis_int-DTD.xml".

If the DTD is defined in an external subset, the DTD can be referenced with the following syntax:

```
<!DOCTYPE root-element SYSTEM "filename">
```

External DTDs are usually stored in files with the file extension ".dtd" [BrPa04a]. The XML document in listing 2-4 references the DTD from listing 2-2 (p. 36) declared in the external subset `thesis.dtd`.

```
<?xml version="1.0"?>
<!DOCTYPE thesis SYSTEM "thesis.dtd">
<!-- A simple document -->
<thesis>
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build a web application.
  </abstract>
</thesis>
```

Listing 2-4: "thesis_ext-DTD.xml".

2.3.2 XML Schema

XML Schema is an XML-based technology that was designed to describe the content and structure of XML documents in XML. As DTDs have serious limitations for defining the constraints of an XML document, XML Schema is considered to be a replacement for DTDs [PaCh05].

2.3.2.1 XML Schema's Background

With the speedy prevalence of XML, DTD soon became "inadequate to meet the needs of the wide spectrum of applications" [TuGo99] that have been considered to use XML (such as metadata interchange, electronic commerce or document publishing).

Developers were (and still are) faced with following DTD's limitations:

- DTDs are composed of non-XML syntax,
- DTDs do not support "data types beyond character data" [TuGo99],
- In order to be able to use a namespace, "the entire namespace has to be defined within the DTD" [ScVa02],
- With DTDs, the validation "always starts with the outermost XML element, and always validates every element and attribute in the document" [Sper05]. There is no possibility to validate just parts of the document.

Therefore W3C set up a working group in 1999 in order to develop a schema to compensate for these shortcomings. Finally, W3C published XML Schema specification as a W3C Recommendation on 3rd May, 2001.

2.3.2.2 Structure of an XML Schema Document

In contrast to DTDs, an XML Schema document is an XML document with pre-defined elements and attributes describing the structure of another XML document [Chas03].

The XML Schema is defined in a separate file and usually stored with the "`.xsd`" extension [PaCh05].

Listing 2-5 represents the equivalent XML Schema to the DTD shown in listing 2-2 (p. 36).

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="thesis">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="title" type="xsd:string" />
        <xsd:element name="author" type="authorType" />
        <xsd:element name="year" type="yearType" />
        <xsd:element name="pages" type="xsd:integer" />
        <xsd:element name="abstract" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="authorType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" />
      <xsd:element name="department" type="xsd:string" />
      <xsd:element name="email" type="xsd:string" />
    </xsd:sequence>
    <xsd:attribute name="id" type="idType" use="required" />
  </xsd:complexType>
  <xsd:simpleType name="yearType">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:minInclusive value="1970" />
      <xsd:maxInclusive value="2010" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="idType">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:pattern value="[0-9]{7}" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 2-5: XML Schema "thesis.xsd".

The thesis.xsd XML schema is itself an XML document. The root element of every XML Schema document is an element called `schema` that belongs to the `http://www.w3.org/2001/XMLSchema namespace`.

An XML namespace is a W3C-standard that provides a method for naming elements and attributes uniquely in an XML document by associating them with namespaces identified by URI references. This is essentially useful if an XML document contains element or attribute names from more than one XML vocabulary[15], as two (or more) elements or attributes from different DTDs or XML Schemas can have identical names. The ambiguity between identically named elements or attributes in an XML document can be resolved in assigning each XML vocabulary a namespace [BrHo99][16].

As XML is intended to be a self-describing data format, elements are declared by using an element called `element`, and the intended name of the element is specified as a value of an attribute called `name`. Attributes are declared by using an element called `attribute`. The attribute's name is also specified as a value of the element's attribute `name`. For example, in listing 2-5 (p. 40), the root element is called `thesis`, the value of the attribute `name` is `thesis`.

Elements in an XML Schema can be classified as simple or complex types [PaCh05].

2.3.2.2.1 Simple and Complex Types

A simple type element cannot contain any attributes or child elements. In listing 2-5 (p. 40), a simple type is declared by the statement

```
<xsd:element name="thesis">
```

With complex types it is possible to define which child elements an element may contain, how often the child elements can occur and in which order they have to be declared. Complex types are declared by the `complexType` element. To in-

[15] The term XML vocabulary is referred to as a set of elements and attributes defined in a DTD or an XML Schema [AuPe06].

[16] The prefix `xsd:` in `thesis.xsd` (see listing 2-5, p. 40) is used by convention to denote the XML Schema namespace, although any prefix could have be used [FaWa04].

dicate the element's order, the sequence element is used (see listing 2-5, p. 40) [PaCh05].

Consequently, the `thesis` element declared in listing 2-5 (p. 40) has to contain five child elements (`title`, `author`, `year`, `pages` and `abstract`) which have to occur in this defined order. The element `author` has to contain an attribute called `id` and three child elements (`name`, `department` and `email`).

2.3.2.2.2 Restricting Data Types

Both simple and complex types can restrict the data type an element may have. XML Schema provides a wide range of built-in data types. Table 2-2 shows an excerpt of built-in data types [TuGo99].

Data type	Description
`string`	a sequence of characters
`boolean`	true/false
`integer`	numbers without a fractional part
`positiveInteger`	positive whole numbers excluding zero
`date`	string representing date values
`uri`	uniform resource identifier

Table 2-2: Built-in data types[17].

If an element shall be limited to a specific data type, the element simply declares the attribute `type` with the corresponding value, such as:

```
<xsd:element name="pages" type="xsd:integer" />
```

This way, the element `pages`'s data type has to be of the type `integer`.

Furthermore, XML Schema provides the possibility to control the value of XML elements and attributes. The allowable values for the data type may be constrained by using facets. According to W3C, a facet is "a single defining aspect

[17] For a complete list of built-in data types please refer to [BiMa04].

of a value space" [BiMa04]. W3C XML Schema defines 12 facets for simple data types[18] [PaCh05].

The XML Schema thesis.xsd in listing 2-5 (p. 40) makes use of three facets [PaCh05]:

- `minInclusive`: the numeric value of "the data type is greater than or equal to the value specified" [PaCh05].
- `maxInclusive`: the numeric value of "the data type is less than or equal to the value specified" [PaCh05].
- `pattern`: the value of "the data type is constrained to a specific sequence of characters that are expressed using regular expressions" [PaCh05].

In order to constrain the data type, the facets are nested with according values in a restriction element. The restriction tag contains the attribute base that declares a built-in data type. The restriction element again is nested in a `simpleType` element that contains an attribute `name`. The attribute `name`'s value is referenced by the element whose data will be constrained. The following extract from listing 2-5 (p. 40) shows how to constrain data in an XML Schema:

```
<xsd:simpleType name="yearType">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:minInclusive value="1970"/>
    <xsd:maxInclusive value="2010"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="idType">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:pattern value="[0-9]{7}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Consequently, an element's or attributes value that is assigned the type `yearType` has to range between 1970 and 2010 and an element's or attributes value that is assigned the type `idTpye` has to be composed of exactly 7 digits.

[18] As the explanation of all facets would be beyond the scope of this paper, only three facets will be discussed to give the reader an idea of how to constrain data values. For further information on facets please refer to [BiMa04].

2.3.2.3 Linking the XML Schema

An XML document that has to be validated against an XML Schema either uses the `schemaLocation` or the `noNamespaceSchemaLocation` attribute to link to the corresponding schema. Strictly speaking, these attributes just provide “hints [...] to a processor regarding the location of schema documents” [FaWa04].

The `schemaLocation` attribute's value “consists of one or more pairs of URI references” [FaWa04] which are separated by white space. The first member of such a pair is a namespace name and the second member is a “hint where to find an appropriate schema document for that namespace” [FaWa04]. The declaration of the `schemaLocation` attribute requires that the referenced schema defines a target namespace[19] (please note that in the schema `thesis.xsd` no target namespace was declared, see listing 2-5 p. 40). By declaring the `schemaLocation` attribute, the referenced XML schema will be used to check the XML document's validity “on a namespace by namespace basis” [FaWa04]. For example, listing 2-6 illustrates how the XML file `thesis.xml` (see listing 2-1, p. 25) needs to be modified in order to reference an XML Schema where a target namespace was declared[20].

```
<thesis
  xmlns="http://www.heinisch.cc/thesis"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.heinisch.cc/thesis thesis.xsd">
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build a web
    application.
  </abstract>
</thesis>
```

Listing 2-6: “thesis_schemaLocation.xml”.

[19] A target namespace enables the distinction “between definitions and declarations from different vocabularies” [FaWa04], refer to [FaWa04] for further information.

[20] The `xsi:` prefix is used by convention only [FaWa04].

Listing 2-7 shows the according XML Schema declaring a target namespace.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.heinisch.cc/thesis"
  xmlns="http://www.heinisch.cc/thesis"
  elementFormDefault="qualified">
  <xsd:element name="thesis">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="author" type="authorType"/>
        <xsd:element name="year" type="yearType"/>
        <xsd:element name="pages" type="xsd:integer"/>
        <xsd:element name="abstract" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="authorType">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="department" type="xsd:string"/>
      <xsd:element name="email" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="idType" use="required"/>
  </xsd:complexType>
  <xsd:simpleType name="yearType">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:minInclusive value="1970"/>
      <xsd:maxInclusive value="2010"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:simpleType name="idType">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:pattern value="[0-9]{7}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 2-7: "thesis_targetNamespace.xsd".

The `noNamespaceSchemaLocation` attribute "is used to provide hints for the locations of schema documents that do not have target namespaces" [FaWa04]. An XML Schema without a target namespace implies that "the definitions and declarations from that schema [...] are referenced without namespace qualification" [FaWa04]. That means, there is no "implicit namespace applied to the reference by default" [FaWa04] (e.g. as opposed to the example provided in listing 2-6 and listing 2-7 where the implicit namespace "`http://www.heinisch.cc/thesis`" is applied to references by default). In order to be able to validate traditional XML documents which do not use namespaces at all, an XML Schema with no target namespace must be provided.

Consequently, the XML Schema `thesis.xsd` is referenced in the XML document `thesis_noNamespaceSchemaLocation.xml` as listing 2-8 illustrates.

```
<thesis
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="thesis.xsd">
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build a web
    application.</abstract>
</thesis>
```

Listing 2-8: "thesis_Schema.xml".

2.4 Displaying XML Documents

As discussed in chapter 2.1.6 (p. 23), XML is a metalanguage that's purpose is to describe information. XML documents do not carry information about how to display the data. Consequently, most Web browsers will just display the XML document tree. For example, the XML document `thesis.xml` (see listing 2-1, p. 25) displayed in the Web browser Mozilla Firefox [Mozi05a] is illustrated in figure 2-2.

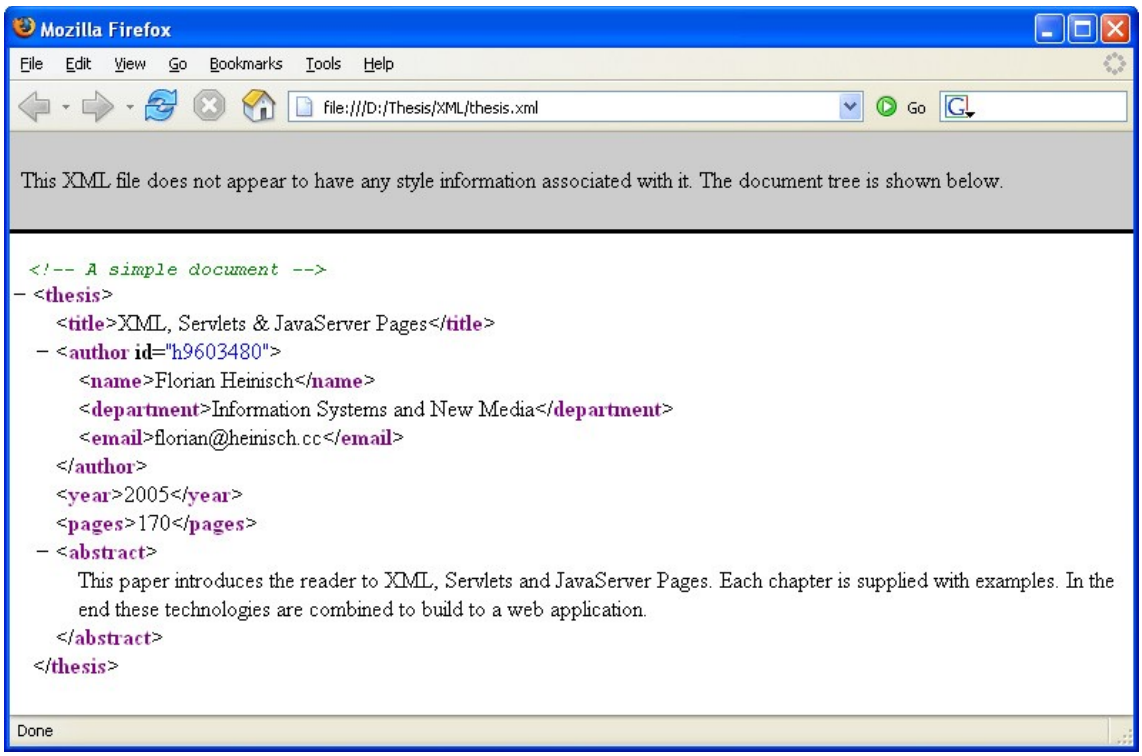

Figure 2-2: "thesis.xml" displayed in a Web browser.

2.4.1 Style Sheets

In order to display an XML document's content in a particular design, the XML document needs to be supplied with additional "display-information" which can be provided by style sheets. Style sheets provide the possibility to describe "how documents are presented on screens, in print, or perhaps how they are pronounced" [Bos06a]. If style sheets are attached to structured documents on the Web (such as HTML or XML documents), the presentation of these documents can be influenced "without sacrificing device-independence" [Bos06a].

Within the scope of this thesis two style sheet languages to present XML documents will be discussed:

- CSS (Cascading Style Sheets), and
- XSL (eXtensible Stylesheet Language).

In chapter 2.4.1.1 and 2.4.1.2 it will be discussed how to use CSS and XSL to present XML documents' content in Web browsers.

2.4.1.1 CSS

The W3C defines CSS as “a simple mechanism for adding style (e.g. fonts, colors, spacing) to Web documents” [Bos06b]. With CSS, it is possible to attach “style properties to the elements of a source document” [LiBo98]. For example, following CSS rule sets the text color of the element `year` of the XML document `thesis.xml` (see listing 2-1, p. 25) to green:

```
year { color: green }
```

A CSS rule consists of a selector (e.g. “`year`”) and of one or many declarations (e.g. “`color: green`”) which are grouped within curly braces and separated by semicolons. A declaration has two parts: property (e.g. “`color`”) and value (“`green`”). If several selectors have the same declarations, they can be grouped into a comma-separated list, such as

```
department, email {
  display: block;
  font-size: 16px;
}
```

Listing 2-9 represents a CSS file thats specifies style properties for every element in the XML document `thesis.xml` (see listing 2-1, p. 25).

```
thesis {
  display: block;
  width: 400px;
  border: 5px solid red;
  margin: 10px;
  padding: 5px;
  font-family: Arial, Helvetica, sans-serif;
}
title {
  display: block;
  color: black;
  font-size: 24px;
  margin: 0px 0px 10px 0px;
}
author {
  display: block;
  color: blue;
  font-size: 20px;
}
name {
  display: inline;
}
department, email {
  display: block;
  font-size: 16px;
}
year {
  display: block;
  color: green;
  font-weight: bold;
}
pages, abstract {
  display: block;
}
```

Listing 2-9: "thesis.css".

In order to display an XML document in a document-like fashion, it must be declared “which elements are inline-level and which are block-level” [BoÇe05]. In CSS, for each element of the source document an “invisible” rectangular box is generated[21]. These boxes belong to a formatting context which “may be block or inline, but not both simultaneously” [BoÇe05]. According to the CSS specification, “block boxes participate in a block formatting context” [BoÇe05] in which boxes are laid out vertically (i.e. they cause a line break) whereas inline boxes participate in an inline formatting context” [BoÇe05] in which “boxes are laid out horizontally” [BoÇe05] (i.e. they do not cause a line break).

As an in depth discussion about CSS would be beyond the scope of this thesis the reader is asked to refer to [BoÇe05] for further information.

[21] This is referred to as the CSS box model which will not be discussed in detail (refer to [BoÇe05] for further details).

In order to display an XML document with the formatting styles defined in `thesis.css` the CSS document can be referenced by the processing instruction [Clark99]:

```
<?xml-stylesheet type="text/css" href="thesis.css"?>
```

Listing 2-10 shows the modified XML document `thesis.xml` that references the CSS document `thesis.css`.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="thesis.css"?>
<thesis>
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build a web application.
  </abstract>
</thesis>
```

Listing 2-10: "thesis_css.xml".

Figure 2-3 illustrates how the XML document `thesis_css.xml` is displayed in a Web browser[22].

[22] In order to display CSS formatted XML-documents in Web browsers, they must be able to process XML and CSS. Web browsers that comply with these requirements are: MS Internet Explorer 5.0+, Netscape 6.0+, Opera 5.12, Mozilla Firefox 1.0+ and Safari 1.2 [Münz01b].

Figure 2-3: The document "thesis_css.xml" displayed in a Web browser.

2.4.1.2 XSL

The Extensible Stylesheet Language (XSL) is a stylesheet language for defining how XML documents can be transformed and presented. XSL's syntax is completely based on XML's syntax, therefore an XLS style sheet is an XML document itself. XSL consists of three parts [Quin05]:

1. XSL Transformations (XSLT): Is a language for transforming XML documents (e.g. into other XML documents, HTML documents, WML-documents, etc.).

2. XML Path Language (XPath): Is an expression language that is used by XSLT to “access or refer to parts of an XML document” [Quin05][23].

3. XSL Formatting Objects (XSL-FO): Is an XML vocabulary for formatting XML documents. XSL-FO provides “facilities to achieve high-quality typographical output” [Kay04] for XML documents[24].

Originally, XSLT was part of XSL. With the ongoing development of XSL the process of transforming and presenting XML documents turned out to be a distinct two-stage process. Therefore, XSL was split into XSLT for defining transformations and into XSL-FO for defining formatting [Kay01].

[23] XPath will not be discussed in detail in this thesis. For further information please refer to XPath's W3C specification available at [ClDe99].

[24] XSL-FO is outside the scope of this thesis. For further information please refer XSL-FO's W3C specification available at [AdBe01].

The formatting part (XSL-FO) and transformation part (XSLT) work independently. So XSL can be considered as two languages. In practice, an XML document is often transformed before it is formatted because the transformation process lets one add the tags the formatting process requires [Holz00].

XSL is a standard recommended by the World Wide Web Consortium. The first two components of the language (XSLT and XPath) became a W3C Recommendation in November 1999. The full XSL Recommendation including XSL-FO became a W3C Recommendation in October 2001 [Quin05].

2.4.1.2.1 Transformation with XSLT

The W3C specification for XSLT defines XSLT as “language for transforming XML documents into other XML documents” [Clar99b]. Michael Kay extends this definition by defining XSLT more generally as “a language for transforming the structure of an XML document” [Kay01]. In fact, with XSLT it is possible to transform XML documents into other text-based formats than XML, for example to HTML or to plain text with comma separated values.

XML documents can be transformed in three ways [Holz00]:

- Server-side: a server program (e.g. Java servlet) can use an XSLT stylesheet to transform a document automatically and serve it to the client.
- Client-side: a client program (e.g. Web browser) can perform the transformation, reading in the style sheet that is specified with the `<?xml-stylesheet?>` processing instruction.
- Separate program: several standalone programs, usually based on Java, will perform XSLT transformations.

In the XSLT language, a transformation “is expressed as a well-formed XML document [...] which may include both elements that are defined by XSLT and elements that are not defined by XSLT” [Clar99b]. XSLT-elements are distinguished from non-XSLT-elements by belonging to the XSLT namespace which has the URI `http://www.w3.org/1999/XSL/Transform`. The prefix of

`xsl:` is used by convention to refer to elements in the XSLT namespace [Clar99b].

The transformation is done by an XSLT processor. The processor applies an XSLT stylesheet to an XML document and produces a result document. In fact, XSLT “relies on a parser [...] to convert the XML document into a tree structure” [Kay01]. XSLT “takes a tree structure as its input” [Kay05b] and manipulates the tree representation of the XML document. The tree structure of the XML document is referred to as the source tree [Clar99].

XSLT uses XPath to refer to nodes in the source tree. With XPath, it is possible to access specific nodes “while preserving the hierarchy and structure of the document” [Kay01].

Each XSLT stylesheet consists of a number of templates rules. A template rule has a pattern that identifies which nodes in the source tree the template rule applies to. Additional to the pattern, a template rule describes a template that gets added to the result document “when the XSLT processor applies that template to a matched node” [DuCh01].

In the scope of this thesis, Web browser based client-side transformation will be discussed to demonstrate how to transform XML documents with XSLT. Listing 2-11 shows an XSLT stylesheet to transform the XML document `thesis.xml` (see listing 2-1, page 25) to HTML.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="thesis">
    <html>
      <head>
        <title><xsl:value-of select="title"/></title>
        <style type="text/css">@import "thesis_styles.css";</style>
      </head>
      <body>
        <div id="border">
          <h1><xsl:value-of select="title"/></h1>
          <p id="author">
            Author: <xsl:value-of select="author/name"/>
            (Student ID: <xsl:value-of select="author/@id"/>,
            Email: <xsl:value-of select="author/email"/>)
          </p>
          <p>Department: <a href="http://wi.wu-wien.ac.at/">
             <xsl:value-of select="author/department"/></a>
          </p>
          <p>Year: <xsl:value-of select="year"/></p>
          <p>Pages: <xsl:value-of select="pages"/></p>
          <p><b>Abstract:</b><br/>
             <xsl:value-of select="abstract"/>
          </p>
        </div>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Listing 2-11: "thesis.xsl".

The `<xsl:stylesheet>` start tag identifies the document as a stylesheet. The `xmlns:xsl` attribute indicates that the prefix "`xsl:`" is used to identify XSLT elements. The attribute version indicates that the stylesheet only uses features from version 1.0 of the XSLT specification[25].

The stylesheet `thesis.xsl` contains only one template rule which is defined with the element `<xsl:template match="thesis">`. The element's attribute `match="thesis"` describes the template rule's pattern. Its value is an XPath expression that identifies the node `thesis`. Consequently, when the element `<thesis>` of the source document is being processed, the template rule will be applied. The body of the template rule (i.e. the template) describes what output to generate. All elements, that do not belong to the XSLT namespace (such as `<html>`, `<style>` or `<p>`) are copied to the result document without being processed. The element `<xsl:value-of>` copies the value of a node in

[25] At the time of writing this thesis, there is version 2.0 of XSLT in the form of a W3C Candidate Recommendation available at [Kay05a].

the source document to the result document. Its attribute `select` contains an XPath expression that specifies the node from which to get the value from. For example, the XPath expression "`title`" instructs the XSLT processor to find the "`<title>`" element that is a child of "the node that this template rule is currently processing" [Kay01]. In order to retrieve an attribute's value, the XPath expression "`@attribute`" is used, such as "`author/@id`" in `thesis.xsl` (see listing 2-11, p. 54).

The `<style>` element in `thesis.xsl` import a CSS stylesheet that is listed in listing 2-12.

```
#border {
  width: 700px;
  border: 5px solid red;
  margin: 10px;
  padding: 5px;
  font-family: Arial, Helvetica, sans-serif;
}
#author {
  color: blue;
  font-size: 20px;
}
```

Listing 2-12: "thesis_styles.css".

Listing 2-13 shows the XML document `thesis_xsl.xml` to which the stylesheet from listing 2-11 (p. 54) is applied to.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="thesis.xsl"?>
<thesis>
  <title>XML, Servlets &amp; JavaServer Pages</title>
  <author id="9603480">
    <name>Florian Heinisch</name>
    <department>Information Systems and New Media</department>
    <email>florian@heinisch.cc</email>
  </author>
  <year>2005</year>
  <pages>170</pages>
  <abstract>This paper introduces the reader to XML, Servlets and
    JavaServer Pages. Each chapter is supplied with examples. In the
    end these technologies are combined to build a web application.
  </abstract>
</thesis>
```

Listing 2-13: "thesis_xsl.xml".

The stylesheet `thesis.xsl` is identified by the processing instruction `<?xml-stylesheet?>`. The XML document `thesis_xsl.xml` from listing 2-13 can

be invoked in a Web browser with a built-in XSLT processor. For example, the Web browser Internet Explorer 6.0 from Microsoft is shipped with an XSLT processor[26] and Mozilla's Web browser Firefox [Mozi05a] is equipped with the XSLT processor "TransforMiiX"[27]. Figure 2-4 shows the XML document `thesis_xsl.xml` displayed in the Web browser Firefox.

Figure 2-4: The transformed XML document "thesis_xsl.xml" displayed in a Web browser.

[26] For further information please refer to [Micr06].
[27] For further information please refer to [Mozi06].

3 Hypertext Transfer Protocol

In this chapter, a discussion about the HTTP protocol is provided. First, the reader will be introduced to resources on the World Wide Web. Next, a description of HTTP messages such as HTTP request and HTTP response will be given.

3.1 Introduction

Hypertext Transfer Protocol (HTTP) is the network protocol used to request and deliver files and other data (which are collectively called *resources*) on the World Wide Web, e.g. such as HTML files, image files, PDF-files or query results [Mars97].

3.1.1 Resources

A resource is a piece of information that can be identified by a Uniform Resource Identifier (URI). For example, a resource can be an electronic document (e.g. an HTML- or PDF-file), an image, a source of information with a consistent purpose (e.g., "today's weather report for Vienna") or a service (e.g. an HTTP-to-SMS gateway) [BeFi05].

3.1.2 Uniform Resource Identifier

A Uniform Resource Identifier (URI) is a “compact sequence of characters that identifies an abstract or physical resource” [BeFi05]. It conforms to a certain syntax that is codified by the Internet Engineering Task Force (IETF) as RFC 3986[28].

The URI syntax is a URI scheme name such as “`http`”, “`ftp`” or “`mailto`” that “refers to a specification for assigning identifiers within that scheme” [BeFi05]. The scheme name is followed by a colon character and a scheme-specific part. The syntax and semantics of the scheme-specific part are determined by each scheme’s specification [BeFi05].

[28] RFC stands for Request for Comments. For detailed information concerning the URI standard please refer to RFC 3986 available at [BeFi05].

The general syntax for URIs provides the following template:

`<scheme>:<scheme-specific part>`

For instance, URIs can take the following forms:

- ftp://ftp.heinisch.cc
- http://www.theserverside.com/news/thread.tss?thread_id=32014
- mailto:florian@heinisch.cc

3.1.2.1 Uniform Resource Locator

Uniform Resource Locators (URLs) are a subset of URIs that “provide a means of locating a resource by describing its primary access mechanism” [BeFi05] (e.g. its network location). According to the W3C/IETF URI Planning Interest Group, “an HTTP URI is a URL” [CoCo01]. Additionally, the contemporary point of view among the interest group is that the term URL is a context-dependent aspect of URI and rarely needs to be distinguished [CoCo01].

The general syntax for URL schemes “that involve the direct use of an IP-based protocol to a specified host on the Internet” [BeMa94] defines the following pattern:

`<scheme>://<user>:<password>@<host>:<port>/<url-path>`

The scheme-specific part starts with a double slash “to indicate that it complies with the common Internet scheme syntax” [BeMa94].

The declaration of the parts “`<user>:<password>@`”, “`:<password>`”, “`:<port>`” and “`/<url-path>`” is optional. Table 3-1 describes the different components of a URL [BeMa94].

URL component	Description
`scheme`	Defines the name of the scheme being used.
`user`	Is an optional user name that some schemes (e.g. ftp) allow to be specified.
`password`	Is an optional password that (if present) follows the user name separated by a colon.
`host`	Is a “fully qualified domain name[29] of a network host” or [BeMa94] an IP address[30].
`port`	Indicates the port number to connect to. If the port number is present it has to be separated from the host by a colon.
`url-path`	Is specific to the scheme. The url-path “supplies the details of how the specified resource can be accessed” [BeMa94].

Table 3-1: Description of URL components.

In the following sub chapter the HTTP URL scheme will be discussed briefly. For information on other specific URL schemes the reader is to asked to refer to RFC 1738 available at [BeMa94].

3.1.2.1.1 HTTP URL Scheme

“The HTTP URL scheme is used to designate Internet resources” [BeMa94] that are accessible by using HTTP. The HTTP URL scheme takes the form [BeMa94]:

`http://<host>:<port>/<path>?<searchpart>`

For the definition of `<host>` and `<port>` please refer to chapter 3.1.2.1 (p. 58). If no port is declared the default port is 80. The component `<path>` is an HTTP selector and `<searchpart>` is a query string. Both `<path>` and `<searchpart>` are optional [BeMa94].

[29] For further information on fully qualified domain names please refer to RFC 1034 available at [Mock87] and to RFC 1123 available at [Brad89].

[30] Such an IP address takes the form as a set of four decimal digit groups that are separated by dots [BeMa94].

A typical HTTP URL can take the form of:

```
http://localhost:9080/MusicStore/products.do?select=13
```

The URL scheme is `http`, the host is `localhost`, the port is `9080`, the path is `MusicStore/products.do` and the searchpart is `select=13`.

3.1.3 HTTP Defintion

Hypertext Transfer Protocol (HTTP) is an “application-level protocol for distributed, collaborative, hypermedia information systems” [FiGe99] which usually runs on top of Transport Control Protocol (TCP) and Internet Protocol (IP)[31]. It is a generic and stateless protocol.

The HTTP protocol is based on a request/response paradigm: an HTTP client (e.g. Web browser) opens a connection and sends a request message to an HTTP server (e.g. Web server) which returns a response message according to the received HTTP request. After delivering the response, the server closes the connection [Mars97].

At the time of writing, the latest HTTP version was 1.1. This version brought an import improvement: prior to this version a TCP connection was opened and closed after each request/response transaction. For instance, if an HTML file with five images was requested this resulted in six requests and consequently in six TCP connections. HTTP/1.1 introduced persistent connections so one TCP connection can be used for multiple transfers (provided that the embedded elements reside on the same server). Nevertheless, the TCP connection will be closed by the server as HTTP servers usually have some time-out value beyond which they will no longer maintain an inactive TCP connection [FiGe99], [ReKi01].

The following simplified example demonstrates how HTTP works in practice when a user agent (also referred to as the client, e.g. a Web browser) requests the URL [Heth97], [cf. HaNe02, 1193]:

---

[31] Please note that HTTP can be “implemented on top of any other protocol on the Internet [...]. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used” [FiGe99].

`http://localhost:9080/MusicStore/products.do?select=13`

1. The client resolves the IP address for `localhost`[32].

2. The client opens a TCP connection to the IP address and to the default port 80 (as no port was declared).

3. Once the connection is established, the client sends a request (see chapter 3.2.1).

4. The server processes the request and sends back a response (see chapter 3.2.2, p. 65) over the TCP connection.

3.2 HTTP Messages

Both HTTP request and HTTP response messages consist of a start-line, zero or more header fields (which are also known as HTTP headers), an empty line indicating the end of the header fields, and possibly a message-body. Each line has to be delimited by Carriage Return/Line Feed pairs [FiGe99].

3.2.1 HTTP Request

Figure 3-1 illustrates an HTTP request's format [Fisc01].

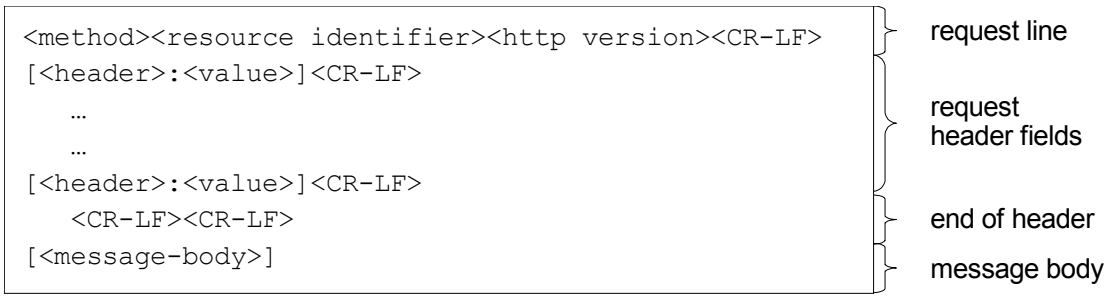

Figure 3-1: HTTP request format.

According to the simplified example mentioned above the request the client sends to the server by invoking the URL is shown in figure 3-2[33].

[32] This is accomplished via Domain Name Service (DNS) [cf. HaNe02, 1193]. For further information on DNS, please refer to RFC 1034 available at [Mock87a] and RFC 1035 available at [Mock87b].

[33] The request shown in figure 3-2 was collected by the use of the software "LiveHTTPHeaders" which is an extension for the Mozilla Firefox or Netscape Web browser available at [Live05].

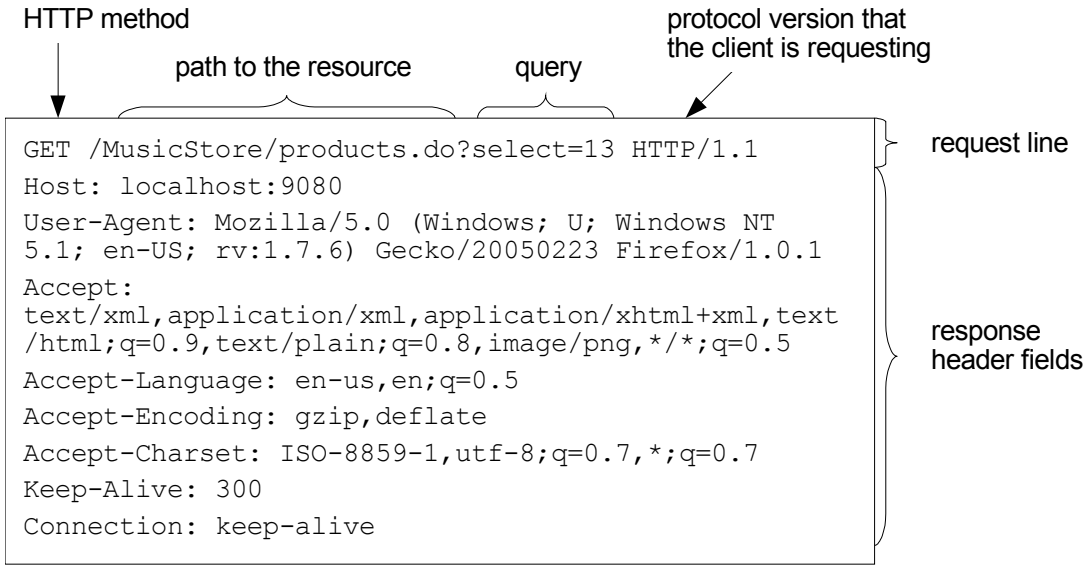

Figure 3-2: Sample HTTP request (GET method).

In the context of HTTP requests, the start line is called the request line. A request line has three parts, separated by spaces: an HTTP method name, the local path of the requested resource and the version of HTTP being used. Optionally, the request line may contain a query that contains parameters. The query is appended to the path and starts with a question mark [cf. BaSi04, 15].

HTTP Methods

The HTTP/1.1 specification defines eight HTTP methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. As in the context of this thesis GET and POST methods will be used, the remaining six HTTP methods will be not be discussed[34] [FiGe99].

3.2.1.1.1 GET

GET is the default HTTP method used by Web browsers for invoking URLs when hyperlinks are clicked by a user or for sending form-data. The GET method means “retrieve whatever information [...] is identified by the request-URI” [FiGe99] (i.e. the HTTP URL). The point of GET is to get something (i.e. a resource) from the specified server. The HTTP request’s message body remains empty as any parameters are included in the query which is appended to the path in the initial line (i.e. the request line) [FiGe99].

[34] For further information about HTTP methods please refer to RFC 2616 available at [FiGe99].

The usage of GET implicates the following drawbacks:

- The total amount of characters sent in a request with a GET method is limited (depends on the Web server). For example, if a user types a long text into an HTML-form's textfield the GET might not work.
- The data that is sent with the GET method is appended to the URL (see figure 3-2, p. 62) in the Web browser bar so this data is exposed.
- The request line is usually stored in Web servers' log-files [W3C95]. As form data are appended to the URL, that data will be collected and stored in log-files. For critical data such as user-ids or passwords this might be a security issue[35].

3.2.1.1.2 POST

POST is designed to provide a block of data (such as the result of submitting a form) to a data handling process. For example, POST is used for posting a message to a bulletin board, newsgroup, mailing list or for "extending a database through an append operation" [FiGe99].

The data is enclosed in the request's message body (in contrast to the GET method where the data is appended the URL).

The Web application discussed in the appendix (see chapter 7, p. 149) includes a form where the user needs to enter personal data. This form is shown in figure 3-3[36].

[35] Professor Flatscher (Vienna University of Economics and Business Administration) provided this advice.

[36] Figure 3-3 was taken as a screenshot from the MusicStore Web application (see chapter 7, p. 149) which was developed by this thesis' author himself.

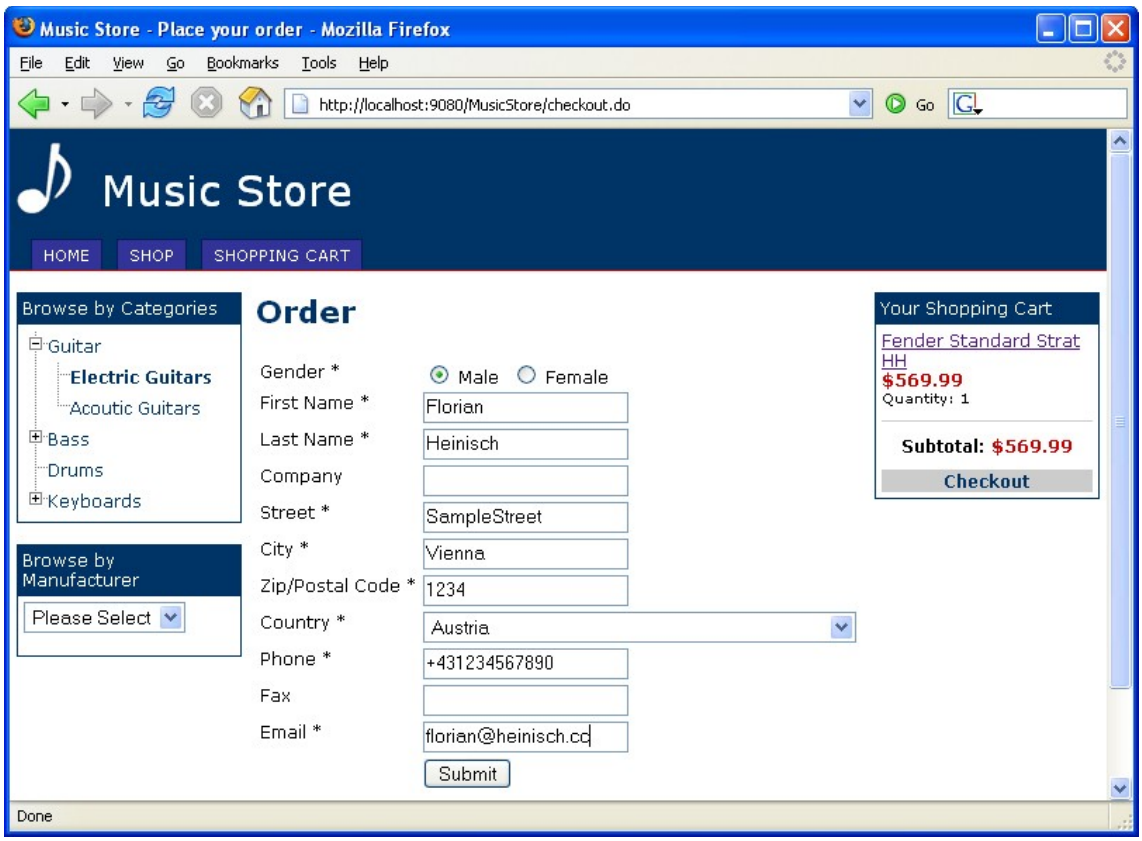

Figure 3-3: Order form of the MusicStore Web application.

The form's field names are (from top to bottom): `gender`, `firstName`, `lastName`, `company`, `street`, `city`, `zip`, `country`, `phone`, `fax` and `email`.

If the user enters the personal data illustrated in figure 3-3 in the form located at `http://localhost:9080/MusicStore/checkout.do` and subsequently hits the submit-button, the request the Web browser sends to the Web server may look as as presented in figure 3-4[37].

[37] The request shown in figure 3-4 was collected by the use of the software "LiveHTTPHeaders" which is an extension for the Mozilla Firefox or Netscape Web browser available at [Live05].

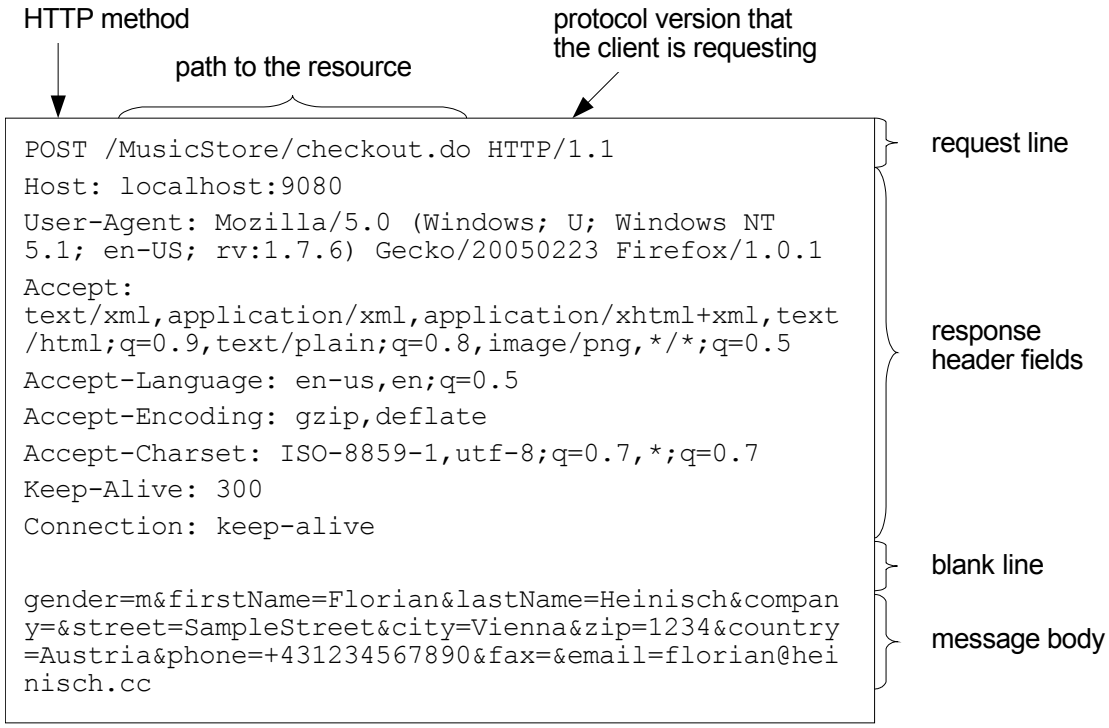

Figure 3-4: Sample HTTP request (POST method).

The request body contains the form data that the user entered. Each form-entry consists of a name/value-pair (such as `firstName=Florian`). The name specifies the form's field name and the value is the data the user entered. The name/value-pairs are separated by an ampersand. The form data is delivered to the server specified in the path for further processing.

3.2.2 HTTP Response

Figure 3-5 illustrates the HTTP response format [Fisc01].

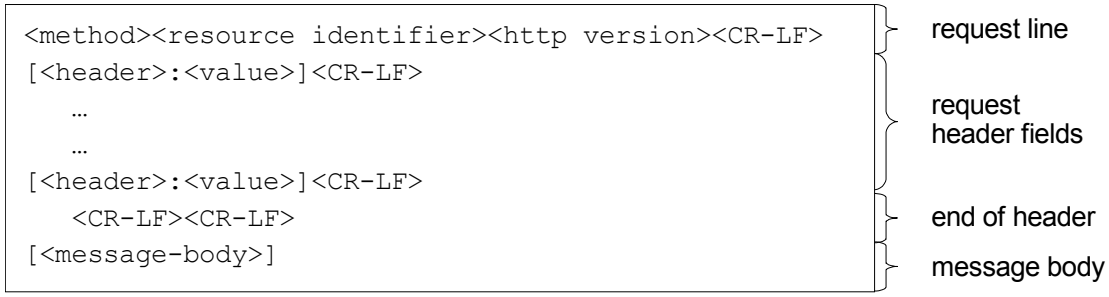

Figure 3-5: HTTP response format.

If a client sends the request from figure 3-2 (p. 62) to the server it may get the response as depicted in figure 3-6.

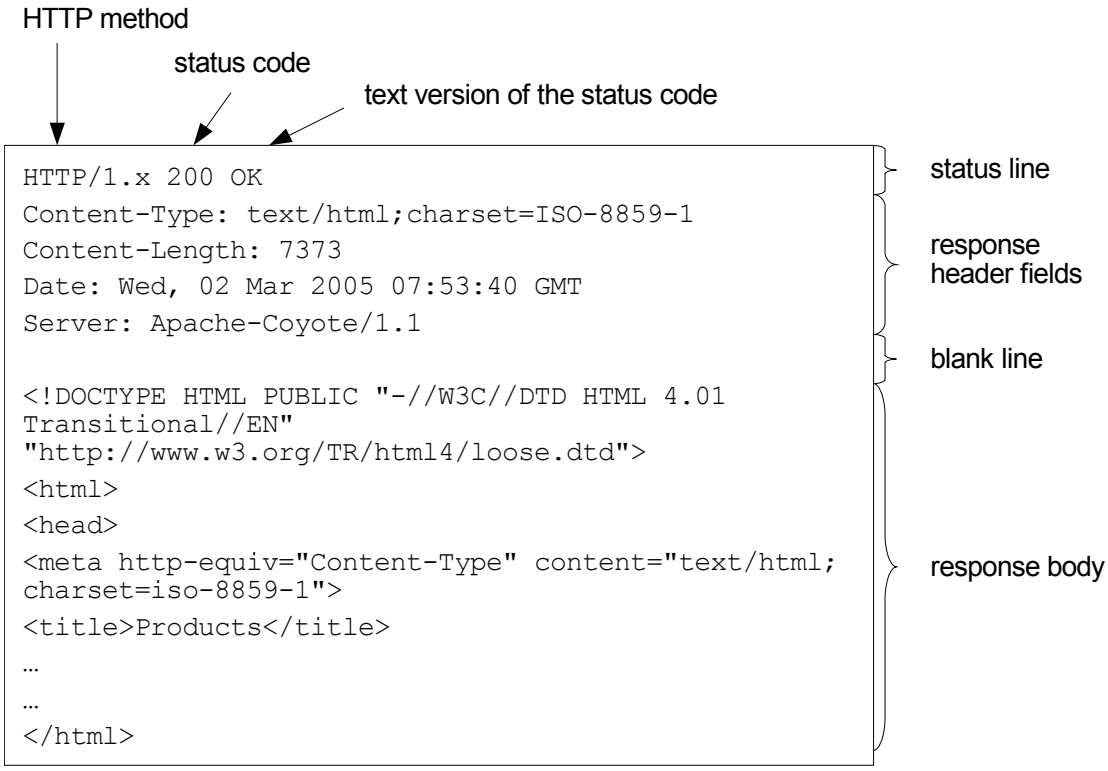

Figure 3-6: Sample HTTP response.

In the context of an HTTP response the initial line is called the status line. It consists of the protocol version followed by a numeric status code and its associated textual phrase, with each element separated by a blank space [FiGe99].

The response body contains the resource that the client requested (provided that the URL identified the resource correctly).

3.2.2.1 Status Code

The status code is a three-digit integer and provides the result of the request. The first digit of the status line identifies the general category of response [FiGe99]:

- 1xx: Informational (e.g. 100 Continue, 101 Switching Protocols),
- 2xx: Successful (e.g. 200 OK, 201 Created),
- 3xx: Redirection (e.g. 300 Multiple Choices, 301 Moved Permanently),
- 4xx: Client Error (e.g. 400 Bad Request, 404 Not Found),

- 5xx: Server Error (e.g. 500 Internal Server Error, 501 Not Implemented).

The Content-Type field is referred to as MIME-type. MIME stands for Multipurpose Internet Mail Extension which is a specification for formatting non-ASCII messages so that they can be sent over the Internet. There are many predefined MIME types, such as GIF graphics files and PostScript files [FrBo96a].[38]

The MIME-type relates to the values listed in the HTTP request's Accept-header (see figure 3-4, p. 65). The MIME-type indicates what kind of content the client is going to receive (i.e. which is located in the HTTP response's body) so the client knows how to render the content [cf. BaSi04, 17].

[38] For further information on MIME, please refer to RFC 2045 available at [FrBo96a], RFC 2046 available at [FrBo96b], RFC 2047 available at [FrBo96c], RFC 2048 available at [FrBo96d] and RFC 2049 available at [FrBo96e].

4 Servlets

In this chapter, the reader will be introduced to Java servlets. To begin with a discussion about the Java Servlet API will be given after which the servlet's “life-cycle” will be explained. In order to enhance the theoretical discussion on servlets, two sample servlets will be presented. Then, Web applications in general will be described. The chapter closes with a short discussion about servlets' major advantages over CGI.

4.1 Introduction

Servlets are Java technology based server-side software components that are protocol and platform independent. Servlets are Java classes that are “compiled to platform neutral bytecode that can be loaded dynamically into and run by a Java enabled Web server” [cf. Cowa01, 17].

Servlets are not bound to a specific client-server protocol (therefore protocol independent) but as they are most commonly used with HTTP, the word “servlet” is often used in the meaning of `HttpServlet` [Zeig99].

Servlets run inside a Java enabled server or an application server[39]. On these servers, servlets are loaded and executed within the server's Java Virtual Machine (JVM). Servlets are to servers what applets are to browsers (in contrast to servlets, applets are loaded and executed within the Web client's JVM). As servlets run inside servers they do not need a graphical user interface (GUI) – consequently servlets are “faceless objects” [cf. WaFi00, 42].

Considering the functionality of servlets they strongly resemble the Common Gateway Interface (CGI)[40]. As with CGI programs, servlets are designed to re-

[39] For a list of servers that enable the execution of servlets please refer to [Sun05a].

[40] CGI is a “a standard for interfacing external applications with information servers” [NCSA98] (such as an HTTP server). CGI is not a language, it is a set of rules for the communication between an HTTP server and an external application (for instance, an external application could be an external database application). As CGI programs are “executed in real-time” [NCSA98], they can output dynamic information. CGI programs can be written in “any language that allows it to be executed on the system” [NCSA98]. The CGI standard is maintained by the National Center for Supercomputing Applications (NCSA) Software Development Group (at the University of Illinois at Urbana) [Conn99]. For further information please refer to [NCSA98].

spond to an HTTP request from the client (e.g. Web browser) and then dynamically construct an HTTP response that is sent back to the client.

4.1.1 Servlet Container

In a broader sense, a servlet container is much like a Web server that's only task is to handle servlets and no other files. Servlet containers (also called servlet engines) “are Web server extensions that provide servlet functionality” [cf. Cowa01, 17]. A servlet container contains and manages servlets through their whole life-cycle. “Servlets interact with Web clients via a request/response paradigm” [cf. Cowa01, 17] that is implemented by the Servlet container.

A Servlet container can be implemented in three ways:

1. being directly built into a host Web server,
2. being installed as an add-on component to a Web server,
3. being built into or installed into web-enabled application servers.

4.1.2 A Servlet's Process

Figure 4-1 below demonstrates a typical servlet process flow [cf. Cowa01, 18]:

1. A client (e.g. Web browser) makes an HTTP request to the Web server.
2. The Web server transfers the request to the servlet container, which sends the request to the appropriate servlet.
3. The servlet dynamically builds a response according to the client's request and transfers it to the server.
4. The server sends the response back to the client.

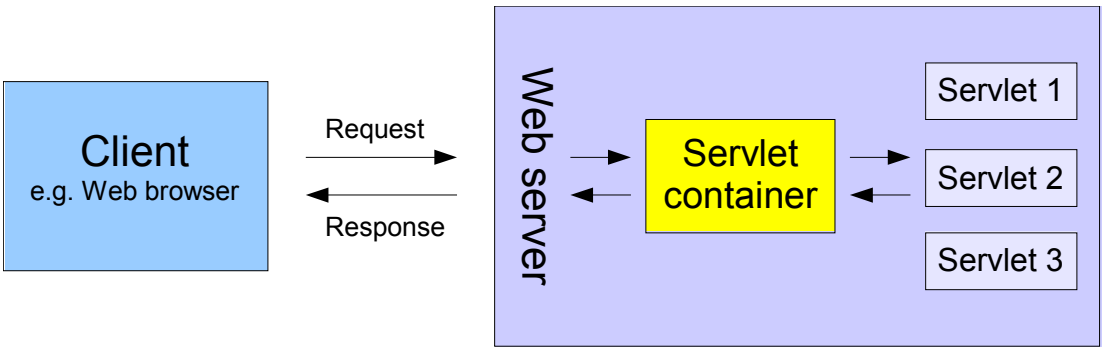

Figure 4-1: Servlet process flow.

4.2 Java Servlet API

The Java Servlet API[41] is a Standard Java Extension API. This means, that the Java Servlet API is not part of the core Java framework but is available as an add-on set of packages [Sun02a].

The Java Servlet API is a set of Java classes. These classes define a standard interface[42] between the Web client and a servlet. The API consists of two packages [cf. WaFi00, 43]:

1. `javax.servlet`: contains classes to support generic protocol-independent servlets (servlets can be used for different protocols, such as HTTP and FTP).

2. `javax.servlet.http`: contains classes and interfaces that build upon these generic servlets to provide support for HTTP protocol and HTML generation.

"The Servlet interface is the central abstraction" [cf. Cowa, 119] of the Java Servlet API. All servlets must implement the servlet interface `javax.servlet.Servlet` which defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These methods are known as "life-cycle methods" [cf. Cowa, 131].

---

41 API stands for application programming interface. It is the specification how a programmer who writes an application "accesses the behavior and state of classes and objects" [Sun02b].

42 This term used in the Java programming language refers to a collection of method definitions and constant values. The interface can later be implemented "by classes that define this interface with the *implements* keyword" [Sun02b].

The servlet interface can be implemented either directly or by extending a class that implements this interface. The `javax.servlet` package provides two classes which implement the Servlet interface [cf. WaFi00, 44], [cf. Cowa01, 21], [BlBo02], [Bodo02b]:

1. `GenericServlet`: is a simple class which implements the `javax.servlet.Servlet` interface and provides all necessary methods a servlet needs for its whole life-cycle.

2. `HttpServlet`: provides additional methods for the processing of HTTP requests.

As servlets are used to demonstrate how to develop Web applications the servlet's discussion in this thesis will be limited to this usage of HTTP servlets. Figure 4-2 depicts a servlet's class diagram that inherits from `HttpServlet` [cf. BaSi04, 98].

Figure 4-2: A servlet's class diagram.

4.3 Basic Servlet Structure

Figure 4-3 illustrates a servlet's basic structure that handles GET requests.

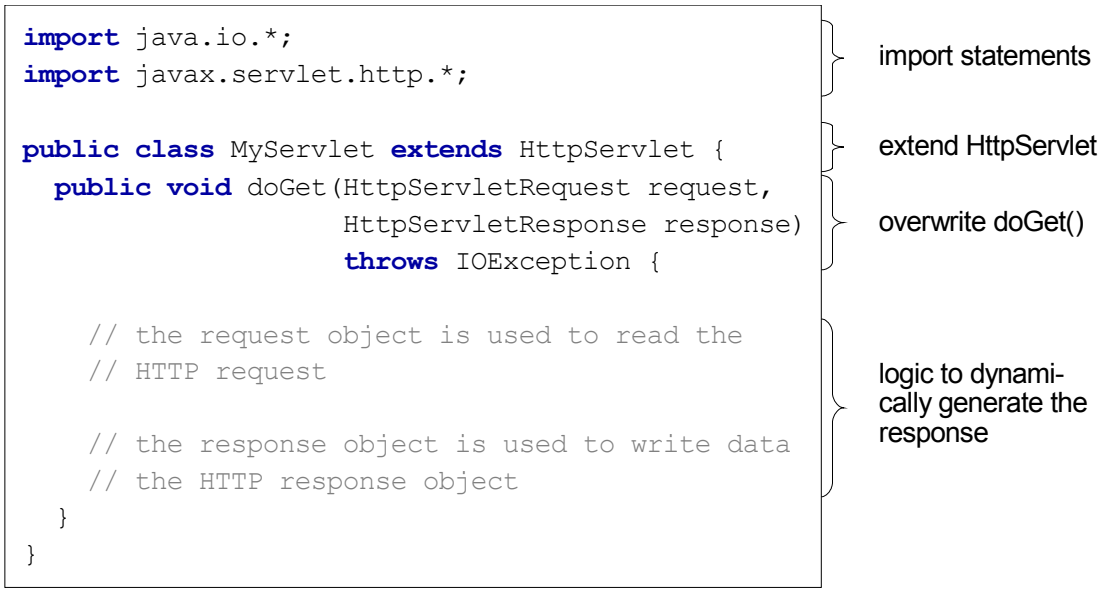

Figure 4-3: Basic servlet structure.

The servlet's developer extends the `HttpServlet` abstract class and consequently inherits all methods needed to handle HTTP requests. According to the HTTP request's method, the developer needs to overwrite the specific method provided by the `HttpServlet` class. For example, if the HTTP request method is a GET then `doGet()` must be overwritten – if it is a POST `doPost()` must likewise be overwritten.

All doXxx() methods inherited from the `HttpServlet` class take an `HttpServletRequest` and an `HttpServletResponse` object as arguments.

The `HttpServletRequest` object provides methods to read the HTTP request such as HTTP request headers, parameters appended to the URL or form data sent in the HTTP request message's body.

The `HttpServletResponse` object is intended to specify the HTTP response that is sent back to the client such as HTTP response headers (e.g. status code, content-type) and the HTTP response body that contains the resource that the client requested (please see chapter 4.4.2, p. 76 for further details) [cf. Hall00a, 22].

4.4 The Servlet's Life-Cycle

Servlets are normal Java classes which are created when needed and destroyed when they are not needed anymore. A client of a servlet-based application does not communicate directly with the servlet, this is accomplished via the Servlet container. The servlet's life-cycle is always managed and processed by the Servlet container [Star02].

There are three main stages in the life of a servlet:

1. "Birth": the servlet is loaded, instantiated and initialized.
2. "Life": the servlet services requests (in this case HTTP requests).
3. "Death": the servlet is destroyed and garbage collected.

The servlet's life cycle is expressed in the API by three main methods:

1. `init()` (→ "Birth")
2. `service()` (→ "Life")
3. `destroy()` (→ "Death")

of the `javax.servlet.Servlet` interface that all servlets must implement directly, or indirectly (by inheritance) through the `GenericServlet` or `HttpServlet` abstract classes (see chapter 4.2, p. 70). The Web server communi-cates with the servlet through this `javax.servlet.Servlet` interface. Every servlet container must adhere to the well-defined servlet life-cycle [cf. Cowa01, 23].

Figure 4-4 depicts a servlet's life-cycle [cf. BaSi04, 97].

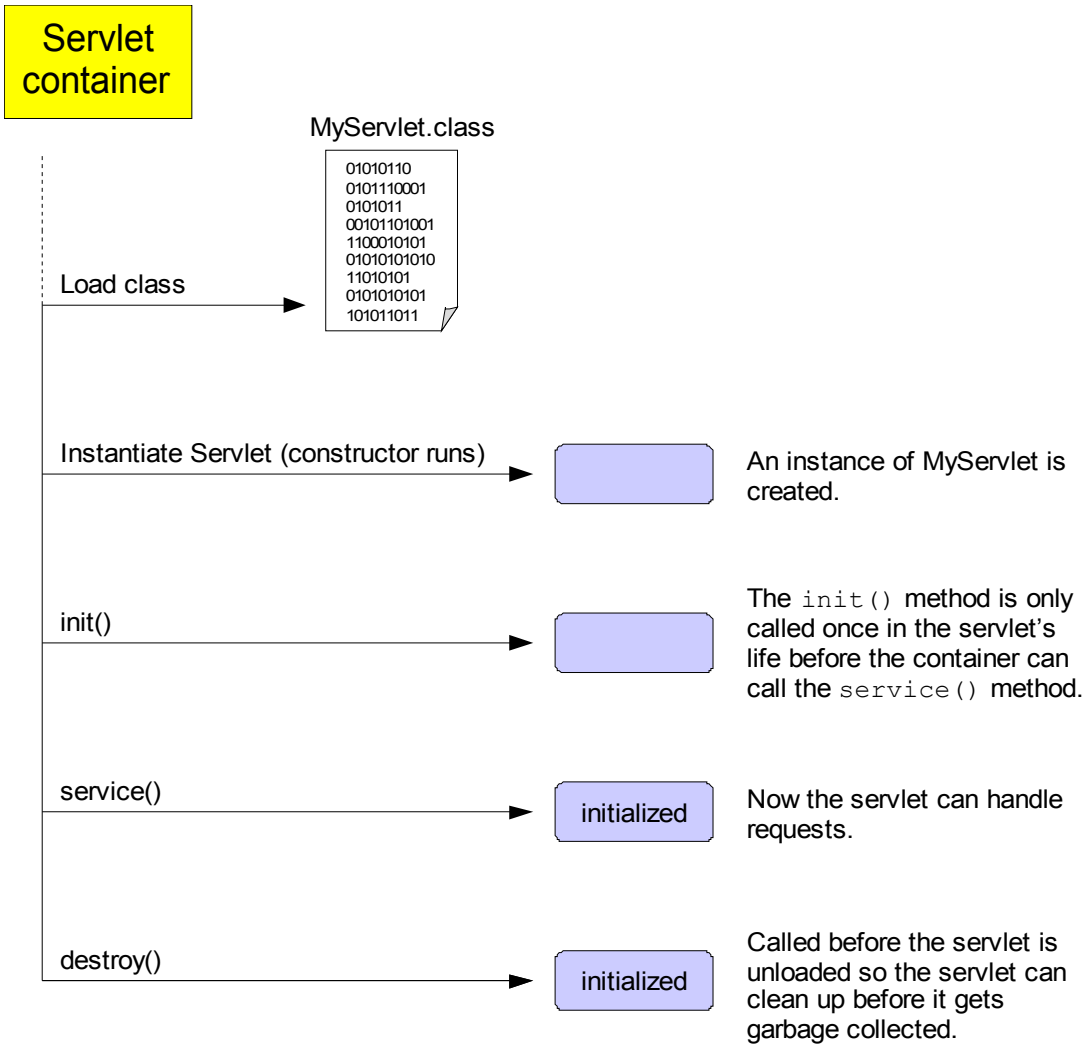

Figure 4-4: A servlet's life-cycle.

4.4.1 Servlet's "Birth": Loading, Instantiation and Initialisation

The servlet's life starts when the servlet container finds the servlet's class file. Therefore, the servlet container must locate the servlet class before a servlet can be loaded. A servlet class can be located on the local filesystem or on a remote file-system. The servlet container looks for deployed Web applications and then searches for the servlet's class file (see chapter 4.6, p. 85 for further details). As soon as the class file is found the container uses the "usual Java class loading facilities to load the servlet class into the JVM" [Gibb00].

Once the servlet has been successfully loaded, the servlet container instantiates a single class instance of that particular servlet class. That single instance handles every request that it is handed to process [HuCr98], [Gibb00].

Servlets can be loaded and instantiated in two ways:

1. The servlet container is configured to preload the servlet when the servlet container starts.

2. The servlet container loads the servlet the first time the servlet is requested.

There is always only one instance of the instantiated servlet. This one instance handles all browser requests. Once a servlet is initialized, it is kept in memory. This way every request is transferred to the servlet in memory which then generates a response. The fact that the servlet is kept in memory, “makes Java servlets an extremely fast and efficient method” for building Web applications [Star02].

Strictly speaking a servlet is just a simple object after it was loaded and instantiated: the servlet’s constructor just instantiates a regular object which is not a servlet in a restricted sense. Immediately after the servlet’s instantiation the container calls the servlet’s `init()` method. Then, the servlet is initialized and can process requests received from clients. Now it can be regarded as a servlet in a more restricted sense [cf. BaSi04, 103].

A servlet's `init()` method is called only once in a servlet’s life. In any case, the `init()` method is guaranteed to be called by the servlet container before the servlet handles its first request [Gibb00].

4.4.2 Servlet's “Life”: Request Handling

Once a servlet has been initialized it is ready to handle client requests. The next listing shows what happens when a client makes a request to a servlet. Therefore it is assumed that a user (the client) will click a link that has an URL to a servlet [cf. BaSi04, 42]:

1. The Servlet container accepts the request for a servlet and consequently creates two objects of the types `HttpServletRequest` and `HttpServletResponse`.

2. The container looks for the correct servlet based on the request's URL and creates or allocates a thread for that request. Then the container calls the servlet's `service()` method and passes the `HttpServletRequest` and the `HttpServletResponse` objects as arguments.

3. Now the servlet's `service()` method by default checks which method the client's HTTP request includes. If the request's method is GET it calls the servlet's `doGet()` method – if the request's method is POST it calls the servlet's `doPost()` method. For now it is assumed that the request's method is GET. So the `service()` method calls the servlet's `doGet()` method and passes the `HttpServletRequest` and `HttpServletResponse` objects as arguments.

4. The `doGet()` method generates the dynamic page and puts it into the `HttpServletResponse` object.

5. The container converts the `HttpServletResponse` object into an HTTP response message and sends it back to the client. The `service()` method completes, consequently the thread either dies or returns to a container-managed thread pool. Afterwards, the `HttpServletRequest` and `HttpServletResponse` objects are deleted.

4.4.2.1 HttpServletRequest Object

`HttpServletRequest` is an interface that extends the interface `ServletRequest`. There are no classes in the Java Servlet API that implement these interfaces. The implementation of those interfaces is left to the servlet container vendor. The `HttpServletRequest` object's purpose is to provide request information for HTTP servlets [cf. BaSi, 107].

Figure 4-5 depicts the class diagrams that illustrate the `ServletRequest` and `HttpServletRequest` interfaces[43].

[43] Due to lack of space the class diagram does not list all methods defined by its interfaces neither cannot each method's functionality be explained. For a complete listing and explanation please see Java™ 2 Plat-form Enterprise Edition, v 1.4 API Specification available at [J2EE05].

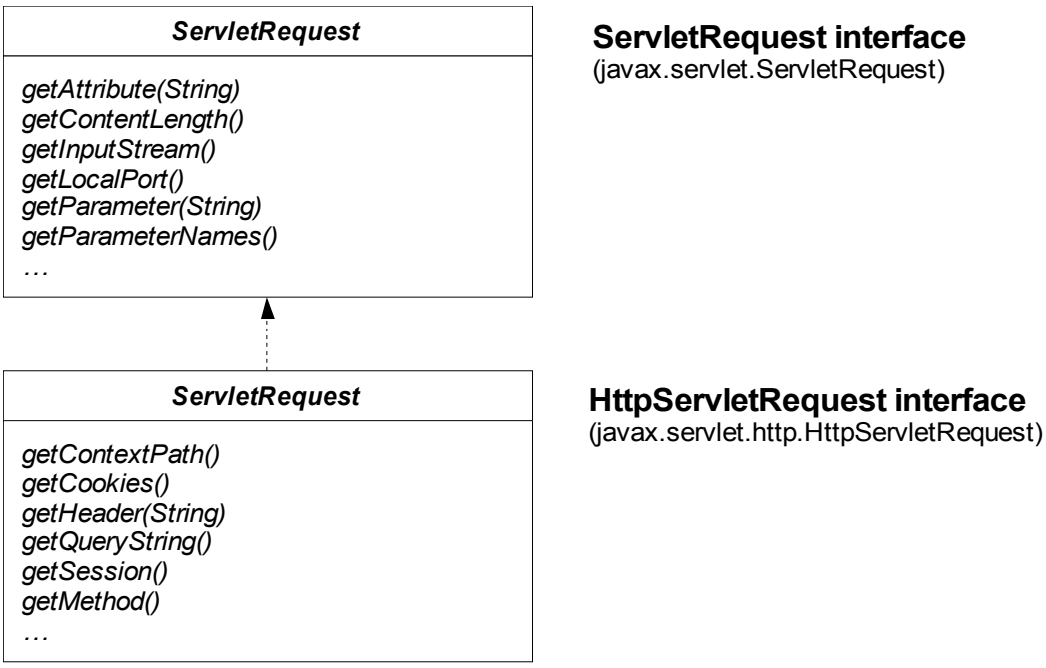

Figure 4-5: HttpServletRequest calls diagram.

To provide an example of how to use the `HttpServletRequest` object listing 4-1 shows how to retrieve a parameter's value according to the HTTP request from figure 3-2 (p. 62).

```
…
public void doGet(HttpServlet request,
                  HttpServletResponse response)
                  throws IOException {

  String selectId = request.getParameter("select");
}
…
```

Listing 4-1: Get a parameter's value.

The method `getParameter()` takes a string as argument. This string has to equal the parameter's name sent by the HTTP request. The string `select` gets the value "13" (see the HTTP request's path in figure 3-2, p. 62).

4.4.2.2 HttpServletResponse Object

`HttpServletResponse` is an interface that extends `ServletResponse`. Simillar to `HttpServletRequest`, there are not any classes in the Java Servlet API that implement these interfaces – the implementation of these interfaces is left to the servlet container vendor. The main function of the

`HttpServletResponse` object is to send data from the server to the client [Bodo02b].

The class diagrams in figure 4-6 illustrate the `ServletResponse` and `HttpServletResponse` interfaces.

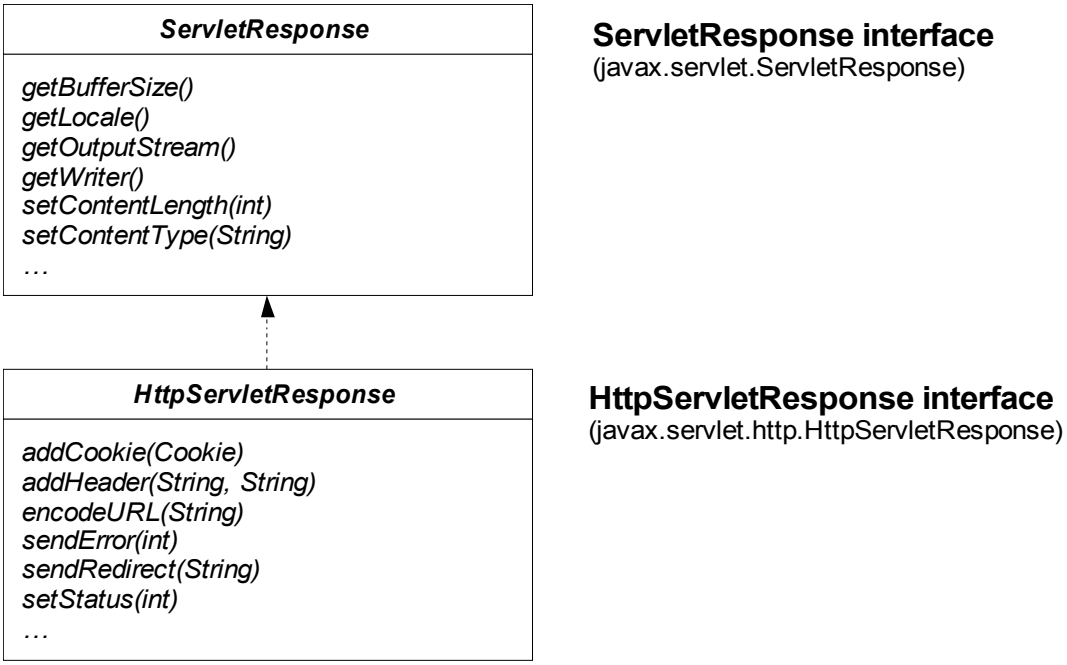

Figure 4-6: HttpServletResponse class diagram.

The most commonly used methods of the `HttpServletResponse`'s objects are `setContentType(String)`, `getWriter()` and `getOutputStream()` [cf. BaSi, 126].

4.4.2.2.1 Content-type

Before any data is sent to the client, the HTTP response's content type header value (i.e. the MIME-type) needs to be set. If it is not set the client (i.e. the Web browser) will not know how to render the data contained in the HTTP response message's body [cf. BaSi04, 130].

Common MIME-types are: `text/html`, `application/pdf`, image/jpeg, `application/x-zip`.

The following Java-code demonstrates how to set the content-type:

```
response.setContentType("text/html");
```

4.4.2.2.2 Writing Data to the HTTP Request

The `ServletResponse` interface offers two streams to write data to the HTTP response message's body: `PrintWriter` for character data and `ServletOutputStream` for bytes. Only one of these two methods may be called to write to the body.

PrintWriter

The PrintWriter object (`java.io.PrintWriter`) is designed to handle character data. Consequently, it is mainly used to send text (such as HTML) to the client [cf. BaSi, 132]. The `PrintWriter` object could be used as follows:

```
PrintWriter out = response.getPrintWriter();
out.println("<h1>Hello World!"</h1>");
```

ServletOutputStream

The `ServletOutputStream` object provides an output stream for sending binary data to the client such as dynamically generated graphics, PDF-files or archives (such as a ZIP-file). E.g.:

```
ServletOutputStream out = response.getOutputStream();
out.write(aByteArray);
```

4.4.3 Servlet's "Death": destroy() Method

The servlet interface provides the `destroy()` method that is called by the servlet container before it can unload a servlet. For instance, a servlet could use the `destroy()` method to update log files or to close database connections before it will be garbage collected.

4.5 Servlet Examples

The purpose of this section is to present two servlets in order to demonstrate how the theory explained above is applied in reality. The provided examples can be downloaded from the author's Web site at [Hein06]. The examples are

packaged in a Web application and available both as a WAR[44]- and as an EAR[45]-file.

4.5.1 Data Servlet

The servlet in figure 4-7 delivers the current date to the client.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class DateServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws IOException {

    response.setContentType("text/html");
    Date date = new Date();
    PrintWriter out = response.getWriter();
    out.println("<html>\n<head>");
    out.println("<title>Current Date</title>");
    out.println("<body>");
    out.println("Today's date: " + date);
    out.println("</body>\n</html>");
  }
}
```

import statements

extend HttpServlet

overwrite doGet()

set ContentType

get a PrintWriter and write a String with current date to the response

Figure 4-7: "DateServlet.java".

As with all servlets that intend to service HTTP request this servlet extends `HttpServlet` to obtain the needed HTTP functionality.

The servlet overrides the method `doGet()` because it expects the HTTP request's method to be of type GET. Since `getWriter()` could throw an `IOEx-`

[44] A WAR (Web ARchive) file is a "JAR archive that contains a Web module" [Sun06b]. A Web module is defined as "deployable unit that consists of one or more Web components, other resources, and a Web application deployment descriptor contained in a hierarchy of directories and files in a standard Web application format" [Sun06b]. For further information please refer to the Java Servlet specification available at [CoYo03].

[45] An EAR (Enterprise ARchive) file is a single JAR archive that contains one or more J2EE modules. A J2EE module is defined as a "software unit that consists of one or more J2EE components of the same container type" [Sun06b]. For example, a Web module is a J2EE module. For further information please refer to the J2EE specification available at [Shan03].

`ception`, `doGet()` has to declare the exception (it also could wrap its content within a `try/catch` block).

As the servlet wants to write character data to the response, it uses the `Print-writer` object for data output. The content that needs to be sent to the client is simply written in `println()` statements.

Figure 4-8 illustrates the DateServlet's output.

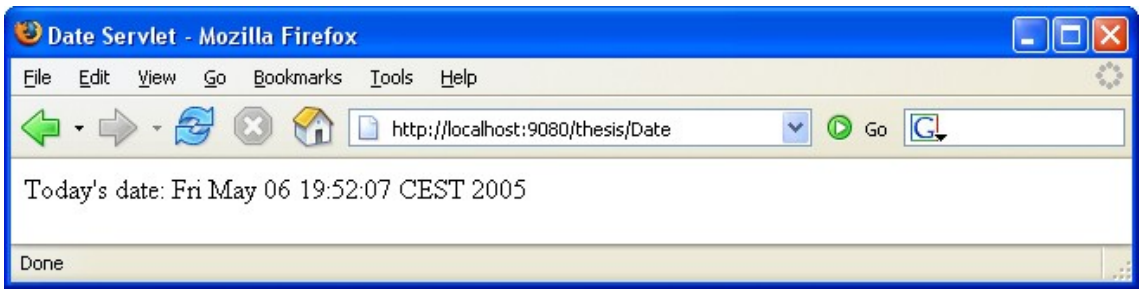

Figure 4-8: DateServlet displayed in a Web browser.

4.5.2 Watermark Servlet

The next servlet demonstrates how to send data to the client using the `ServletOutputStream`. This example servlet loads an image from the server and places a string into the image's center. In practice a servlet such as this could be used to dynamically put a watermark or copyright notice on photos.

```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.OutputStream;
import java.net.URL;
import javax.imageio.ImageIO;
import javax.servlet.http.*;

public class WatermarkServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                    throws IOException {

    StringBuffer path = request.getRequestURL();
    path.replace(path.indexOf(request.getRequestURI()), path.length(),
    request.getContextPath());
    URL imgUrl = new URL (path + "/images/seaside.jpg");
    BufferedImage bufferedImg = ImageIO.read(imgUrl);
    final int WIDTH = bufferedImg.getWidth();
    final int HEIGHT = bufferedImg.getHeight();

    String s = "Copyright by Florian Heinisch";
    Graphics2D g = bufferedImg.createGraphics();
    Font font = new Font("Sans-Serif", Font.PLAIN, 45);
    FontMetrics fontMetrics = g.getFontMetrics(font);
    int sWidth = fontMetrics.stringWidth(s);
    int sAscent = fontMetrics.getAscent();
    int sDescent = fontMetrics.getDescent();
    g.setPaint(Color. getHSBColor(240f, 100f, 0f));
    g.setFont(font);
    g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
    g.drawString(s, (WIDTH-sWidth)/2, (HEIGHT+sAscent-sDescent)/2);

    response.setContentType("image/jpeg");
    OutputStream out = response.getOutputStream();
    ImageIO.write(bufferedImg, "jpg", out);
    out.close();
  }
}
```

Listing 4-2: A servlet that watermarks a picture.

The servlet overrides `doGet()` as it is intended to service HTTP requests of type GET. The servlet's logic is divided into three parts:

1. Load the image: therefore an URL that points to the photo is created. Based on that URL a `BufferedImage` (this class allows off-screen drawing) is created using the `ImageIO` class's `read()` method to read in the image data. Then the photo's dimensions are elicited.

2. Perform graphic operations: the string that will be placed into the photo is declared. The `BufferedImage` class provides a graphics context that can be drawn upon, which is obtained by calling `createGraphics()`.

Next the font to display the string is instantiated. In order to place the string into the middle of the image (both horizontally and vertically) `getFontMetrics(font)` is called so the string's width and height can be measured. Consequently the x and y position is calculated in order to place the string approximately in the photo's center. Afterwards, the color, font and antialiasing is set. Then, the string is drawn upon the `Graphics2D` object.

3. Send the modified photo to the client: before any data is sent to the client, the content-type needs to be set which in this case is `image/jpeg`. Then an `OutputStream` is obtained by calling `getOutputStream()` on the response object. The class `ImageIO` provides a very convenient method with which to write the `BufferedImage` object to the servlet's `OutputStream`. Finally, `close()` on the `OutputStream` is called so any system resources associated with this stream are released.

Figure 4-9 depicts the result when the WatermarkServlet gets executed.

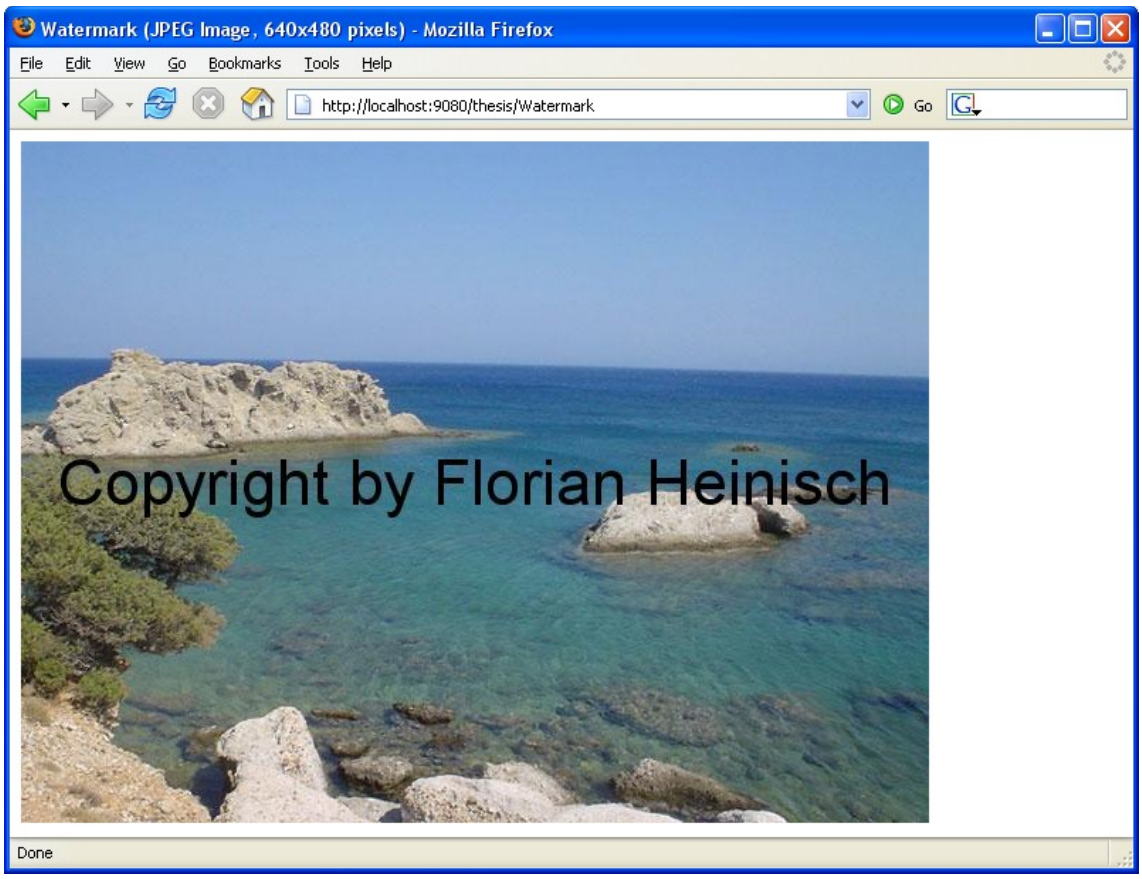

Figure 4-9: WatermarkServlet displayed in a Web browser.

4.6 Deploying Servlets

In order to be able to deploy servlets to a servlet container, the developer needs to know how Java Web applications have to be structured.

4.6.1 Definition of a Web Application

The Java™ Servlet Specification defines the concepts of Web applications. According to the specification “a Web application is a collection of servlets, HTML pages, classes, and other resources that make up a complete application on a Web server” [cf. CoYo03, 68]. Such resources are commonly referred to as Web components.

The following items may be included in a Web application [cf. CoYo03, 69]:

- Servlets,
- JavaServer Pages™,

- utility classes (e.g. database logic),
- static documents (e.g. HTML-files, cascading style sheets, images, sounds),
- client side classes (e.g. applets),
- descriptive meta information that ties all of the above elements together (e.g. XML-files).

4.6.2 Directory Structure

Web applications exist as a structured hierarchy of directories which is defined in the Servlet specification[46]. This clearly defined directory structure makes the Web application portable to any servlet container.

Figure 4-10 illustrates a sample Web application directory structure [cf. BaSi04, 73].

[46] Since the release of Java Servlet Specification 2.2 Servlet, containers have to accept a Web application in a standard format. Prior to version 2.2, “there was little consistency between server platforms” [Apac02].

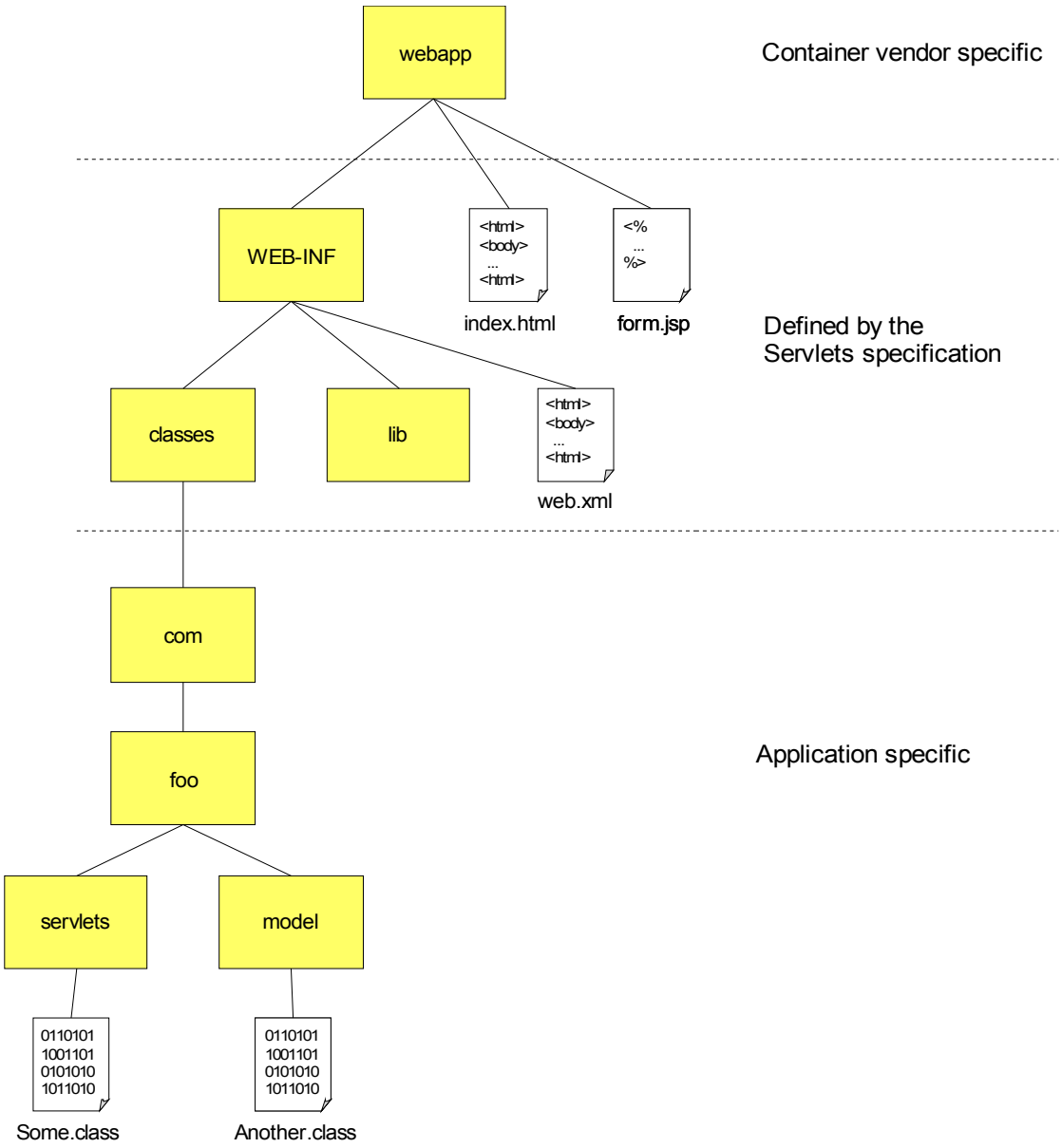

Figure 4-10: Web application directory structure.

The figure is divided into three parts:

1. Container vendor specific part: it depends on the servlet container vendor where to place the Web application on the Web server.

2. Servlets specification specific part: the directory structure of this part has to be the same for all Web applications (defined by the Servlet specification).

3. Application specific part: this part mainly consists of compiled Java classes (although other files such as property or configuration files may be

placed here). It is up to the Web developer(s) how to structure (or “package”) this part of the Web application.

The directory `webapp` is the root directory for the Web application which “serves as the document root for all files that are part of the Web application” [cf. CoYo03, 69]. Usually static HTML files, images, cascading style sheets, JavaScript files and JSPs are placed into the root directory.

In the document root there is a special directory named `WEB-INF`. The `WEB-INF` directory’s content “is not part of the public document tree of the application” [cf. CoYo03, 70]. No file contained in the `WEB-INF` directory can be served directly to a client by the container.

The contents of the `WEB-INF` directory are:

- The XML-file `web.xml`: this is the deployment descriptor that configures the Web application.
- The directory `classes` where servlets and utility classes are placed. If the Java classes are organized in the form of packages, the package names are subdirectories in the `classes` directory.
- The directory `lib` for Java archive files (`*.jar` files) that are used by the Web application (such as third party class libraries, JDBC drivers and JSP tag libraries).

Java classes may be located in the `WEB-INF`’s classes or `lib` directory. The class loader checks first in the `classes` directory. If it does not find the class there it looks for the class in the `lib` directory [cf. CoYo03, 71].

The Web application can be deployed to the Web server in two ways [cf. Bea04, 1-3]:

- As an “exploded” directory format which is recommended primarily for use while developing the Web application.
- As a Web application archive file (`*.war`): Web applications can be packaged into a Web ARchive format (WAR) file using the standard Java

archive tools (e.g. with the command `jar -cvf mywebapp.war *.*`). This format is mainly recommended for production environments [cf. CoYo03, 71].

4.6.3 Deployment Descriptor

The deployment descriptor is an XML-file named `web.xml` that is located in the `WEB-INF` directory. It describes configuration information for the entire Web application and may include the following elements[47] [cf. CoYo03, 103]:

- ServletContext Init Parameter,
- Session Configuration,
- Servlet Declaration,
- Servlet Mappings,
- MIME Type Mappings,
- Welcome File list,
- Error Pages.

The deployment descriptor is a powerful tool to configure the Web application such as mapping URLs to servlets, defining error pages and configuring security roles. As this chapter's main topic concerns servlets only, the configuration of the deployment descriptor for servlets will be discussed. Any other issues will be treated in the appropriate context.

According to the Servlet specification, the servlet's classes reside in the `webapp/WEB-INF/classes` directory. As HTTP servlets are invoked via an URL, the servlet container needs to know where to find the servlet classes. Therefore the `web.xml` file contains XML elements that map URLs to the servlets' classes [cf. BaSi04, 48].

[47] The listing is not complete. The `web.xml` elements depend on the Web components used in the Web application (e.g. if no sessions are used no session properties need to be configured in the `web.xml`).

4.6.3.1 Sample web.xml File

Listing 4-3 illustrates the `web.xml` file that was created to deploy the servlets `DateServlet` and `WatermarkServlet` to the servlet container.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <display-name>thesis</display-name>

  <servlet>
    <servlet-name>DateServlet</servlet-name>
    <servlet-class>cc.heinisch.thesis.DateServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>WatermarkServlet</servlet-name>
    <servlet-class>cc.heinisch.thesis.WatermarkServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>DateServlet</servlet-name>
    <url-pattern>/Date</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>WatermarkServlet</servlet-name>
    <url-pattern>/Watermark</url-pattern>
  </servlet-mapping>

</web-app>
```

Listing 4-3: "web.xml" deployment descriptor.

The root element of the deployment descriptor is always `<web-app>`. A servlet deployment descriptor has to indicate the XML schema by using the J2EE namespace `xmlns="http://java.sun.com/xml/ns/j2ee"` and has to indicate the version of the schema as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
  xsi:schemaLocation="..."
  version="2.4">
...
</web-app>
```

The indication of the XML schema's published version using the `xsi:schemaLocation` attribute is not obligatory, nevertheless it is commonly used [cf. CoYo03, 108].

The two elements needed for URL mapping in the deployment descriptor are [cf. BaSi04, 48]:

1. `<servlet>` maps the internal name to the fully qualified class name.
2. `<servlet-mapping>` maps the internal name to a public URL name.

The `<servlet>` element has two child elements:

1. `<servlet-name>` contains an internal name which is used to link a `<servlet>` element to a specific `<servlet-mapping>` element. The client never sees the `<servlet-name>` value – it is only used within the deployment descriptor.
2. `<servlet-class>` contains the fully qualified servlet's class name (excluding the `.class` extension).

The `<servlet-mapping>` element also has two child elements:

1. `<servlet-name>` is used to link the `<servlet-mapping>` element to the corresponding `<servlet>` element.
2. `<url-pattern>` is the servlet's "pseudonym": this is a made-up name the client uses to get to the servlet, e.g. a hyperlink in an HTML-file could look like this: `<a href="Watermark">click here</a>`.

For example, a user clicks the hyperlink `<a href="Watermark">click here</a>`. The servlet container receives the request and looks at runtime in the `web.xml` file within the `<servlet-mapping>` elements for the `<url-pattern>` element with the value `/Watermark`. Once found, it gets the `<servlet-name>` value `WatermarkServlet` and looks within the `<servlet>` elements for the same `<servlet-name>` value. Finally, it finds the fully qualified servlet's class name in order to invoke the servlet.

4.7 Servlets vs. CGI

To conclude the discussion of servlets, this subchapter discusses the serlvet's major advantages over CGI.

4.7.1 Efficiency

Common Gateway Interface (CGI) is a traditional way to handle web-to-database applications. A CGI program needs to create a separate process for each user request. This process will be terminated as soon as the data transfer is completed. Spawning a separate program instance for each client request takes extra time. The operating system has to load the program, allocate memory for the program, and then deallocate and unload the program from the memory. If the CGI program is relatively short, the overhead of starting the process can take longer then the execution time [WuWa00].

When using servlets, the JVM continues to run and handles each request using a Java thread – instead of a complete operating system process as CGI does. For every request to the CGI program the code for the CGI program is loaded into memory. That means that for e.g. 10 requests to a particular CGI program, the code will be loaded into memory 10 times. Servlets in contrast, would have 10 threads for 10 requests but only a single copy of the servlet class. In addition, as a CGI program is unloaded from memory when it finishes, optimization "that relies on persistent data" [cf. Hall00, 7] cannot be issued. However, servlets can remain in memory after they have completed a response and so they can store complex data between requests.

4.7.2 Portability

As servlets are written in Java and follow a standard API servlets can run virtually unchanged on every server that provides a servlet engine. As a matter of fact, "servlets are supported directly or by a plug-in on virtually every major Web server" [cf. Hall00, 8].

5 JavaServer Pages™

This chapter will introduce the reader to JavaServer Pages™. Therefore, the JSP's "life-cyle" and JSP elements will be explained. Next, the development of "script-free" JSP pages will be described. Finally, it will be demonstrated how to develop ones own custom tags. The presented theory will be endorsed by short and simple examples.

5.1 Introduction

JavaServer Pages (JSP) technology provides the possibility to "create Web content that has both static and dynamic content" [Bodo02a]. JSP technology features all the dynamic capabilities of Java Servlet technology and furthermore provides "a more natural approach for creating static content" [Bodo02a]. The JSP-technology allows mixing static HTML/XML with "dynamically generated content from servlets" [cf. Hall00, 231].

JSP technology was developed by Sun Microsystems. The JSP specification is a standard extension defined on top of the Servlet API in order "to separate the development of dynamic Web page content from static HTML page design" [cf. WaFi00, 96].

A JSP page is a text-based document that contains two types of text [Bodo02a]:

1. static template data which can be expressed in any text-based format, such as HTML, SVG, WML, and XML,

2. JSP elements, which construct dynamic content.

5.1.1 Simple JSP Example

The examples provided in this chapter are available for download at the author's Web site [Hein06] in the form a WAR- or EAR-file. Figure 5-1 shows a JSP that delivers the same content to the client as the servlet listed in figure 4-7 (p. 81).


```
<html>
  <head>
    <title>Current Date</title>
  </head>
  <body>
    Today's date: <%= new java.util.Date() %>
  </body>
<html>
```

Figure 5-1: "date.jsp".

The JSP `date.jsp` demonstrates that JSPs provide a more convenient way to build dynamic Web sites (i.e. Web applications) compared to servlets.

5.2 JSP Container

Similar to the Servlet container the JSP container is a Web server extension that provides JSP functionality. The JSP container is responsible for capturing requests for JSP pages. The JSP container is often implemented as a servlet which is configured to handle all requests for JSP pages. In fact, the Servlet container and the JSP container are usually combined in one package by the name Web container [Berg02].

5.2.1 JSP Advantages over Competing Technologies

The approach used by JavaServer Pages provides several advantages over competing technologies (such as CGI, ASP and PHP). In a nutshell, these advantages are:

1. JSP are widely supported and therefore do not lock developers into a particular operating system or Web server.

2. JSP provide “full access to Servlet and Java technology” [cf. Hall00, 231].

5.3 JSP's Life-Cycle

A JSP is made operable by having its contents (i.e. HTML tags, JSP tags and scriptlets) “translated into a servlet by the JSP container” [cf. WaFi00, 97]. Both dynamic and static elements that are declared within the JSP file are translated

into Java servlet code. The JSP container delivers the translated contents “through the Web server output stream to the browser” [cf. WaFi00, 97].

JSPs go through two phases: a translation phase, and a request phase. The translation phase is carried out once per page (unless the JSP page changes). The request phase is carried out once per request [cf. PeRo03, xxxiii].

Figure 5-2 outlines the tasks performed on a JSP file on the first invocation of the file [cf. WaFi00, 97].

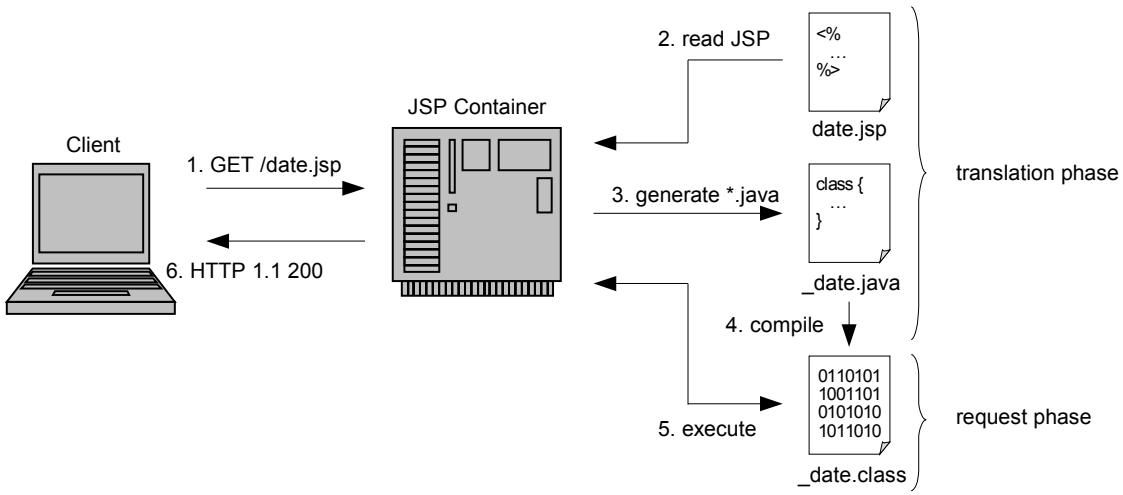

Figure 5-2: Translation and request phase.

Translation phase

1. The Web server gets a request from a Web browser for a JavaServer Page. The Web server transfers the request to the JSP container.

2. Then the JSP container parses the content of the JSP page. JSP tags are translated into Java source code; static HTML-code is converted into Java strings (which is written unchanged to the servlet’s output stream); scripting elements are taken over unchanged.

3. Depending on the content of the JSP page, the JSP container creates a temporary servlet source code (i.e. a Java file).

4. Then the servlet source code is compiled into a servlet class file. The result is a JSP page implementation class file that implements the servlet interface. Once the JSP page implementation class file exists, the servlet

is instantiated (the result is a JSP page object) and the servlet's `init()` method is called.

Request phase

5. The servlet's `service()` method is called, and the servlet logic is executed. The `service()` method “is dispatched on a separate thread by the [...] container in processing concurrent client requests” [Shesh00].

6. The dynamically generated Web content is sent to the Web browser “through the output stream of the servlet’s response object” [cf. WaFi00, 97].

As long as the underlying JSP file remains unchanged all requests are directly transferred to the `service()` method of the servlet that was created in the translation phase to deliver the content to the Web browser. If the JSP file was changed the translation phase has to be repeated – a new JSP page implementation class file will be created by the JSP container. The servlet remains in service until the JSP container is stopped or the servlet is manually unloaded [cf. WaFi00, 98].

As the translation phase might take some time, the first client to request a JSP page will notice a slight delay. Therefore, the JSP specification defines *precompilation* of JSP pages so the translation phase can be initiated explicitly. As a result, the first client will not be faced with this slight delay caused by the translation phase [Berg02].

5.3.1 The Generated Servlet Java-file

The concrete implementation of the servlet source file (i.e. the generated Java-file) depends on the JSP container vendor. Usually, there is no need to look at the container-generated code but it certainly helps to understand JSP thoroughly. Listing 5-1 shows the servlet source file that was generated by the IBM Websphere Application Server 6.0 when it translated the JSP `date.jsp` into a servlet.

```
package com.ibm._jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class _date extends com.ibm.ws.jsp.runtime.HttpJspBase implements
com.ibm.ws.jsp.runtime.JspClassInformation {

  private static String[] _jspx_dependants;
  public String[] getDependants() {
    return _jspx_dependants;
  }

  private static String _jspx_classVersion;
  private static boolean _jspx_isDebugClassFile;
  static {
    _jspx_classVersion = new String("6.0.0.1");
    _jspx_isDebugClassFile = false;
  }

  public String getVersionInformation() {
    return _jspx_classVersion;
  }
  public boolean isDebugClassFile() {
    return _jspx_isDebugClassFile;
  }
  private final static char[] _jsp_string1 = "<html>\r\n<head>\r\n<title>Current
    Date</title>\r\n</head>\r\n<body>\r\nToday's date: ".toCharArray();
  private final static char[] _jsp_string2 = "\r\n</body>\r\n</html>".toCharArray();

  static {}

  private static org.apache.jasper.runtime.ProtectedFunctionMapper _jspx_fnmap = null;

  public void _jspService(HttpServletRequest request, HttpServletResponse  response)
  throws java.io.IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;

    try {
      _jspxFactory = JspFactory.getDefaultFactory();
      response.setContentType("text/html;charset=ISO-8859-1");
      pageContext = _jspxFactory.getPageContext(this, request, response, null, true,
      8192, true);

      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write(_jsp_string1);
      out.print( new java.util.Date() );
      out.write(_jsp_string2);

    } catch (Throwable t) {
      if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (pageContext != null) pageContext.handlePageException(t);
      }
    } finally {
      if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
    }
  }
}
```

Listing 5-1: Auto-generated "servlet_date.java".

As most of the class and interface types of the generated servlets are vendor-specific only the relevant parts (in bold) which are needed to understand the translation phase will be discussed:

> The JSP's template data (see figure 5-1, p.94) is put into char-arrays `_jsp_string1` and `_jsp_string2`. The service method's actual name is `_jspService()` which is called by the servlet's superclass overridden `service()` method and receives the `HttpServletRequest` and `HttpServletResponse` objects as arguments [cf. BaSi04, 294].

Finally, the char-arrays and the current date are written to the servlet's output-stream.

5.4 JSP's Components

A JSP page consists of elements and template data:

- "An element is an instance of an element type known to the JSP container" [cf. PeRo03, 1-10].
- Template data is anything that is not an element, i.e. "anything that the JSP translator does not know about" [cf. PeRo03, 1-10].

There are three types of elements:

- **Directive elements**

  A JSP directive is a global definition that is sent to the JSP container. It remains valid "regardless of any specific requests made to the JSP page" [cf. WaFi00, 99]. Directives do not produce any visible output, they provide information to the container for the translation phase [cf. PeRo03, 1-10). In other words, directives "control the overall structure of the resulting servlet" [cf. Hall00, 233]. There are three types of directives:

  1. page,
  2. include,

3. taglib.

- **Scripting elements**

  Scripting elements are Java code that will become part of the resultant servlet. There are three types of scripting elements [cf. Pele01, 35]:

  declarations,

  scriptlets,

  expressions.

- **Action elements**

  Actions provide information for the request phase. The interpretation of an action depends on the “details of the specific request received by the JSP page” [cf. Pele01, 35]. In other words, actions let the developer “specify existing components that should be used and otherwise control the behavior of the JSP engine” (i.e. the JSP container) [cf. Hall00, 233]

5.4.1 Directive Elements

JSP directives affect “the overall structure of the servlet that results from the JSP page” [cf. Hall00, 247].

The syntax of a directive is: `<%@ directive attribute="value" %>`

5.4.1.1 Page Directive

The page directive defines page dependent attributes to the JSP container. A directive always appears on the top of the JSP file, before any other JSP tags. Any number of page directives can be defined within a JSP page, as long as each attribute-value pair is unique. Attributes or values that are not recognized by the JSP container result in a translation error [Sesh00].

Example: `<%@ page import="java.util.Date" %>`

This page directive imports the class `Date` from the package `java.util`. So this class is made available to the scripting environment within the JSP. The import statement `import java.util.Date;` will be placed in the resulting servlet beneath the servlet's default import statements:

```
package com.ibm._jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

import java.util.Date;

public final class _date extends
com.ibm.ws.jsp.runtime.HttpJspBase implements
com.ibm.ws.jsp.runtime.JspClassInformation {

...
```

For a complete listing of the available attributes and their functions please refer to the latest JSP specification which is available at [PeRo03][48].

5.4.1.2 Include Directive

The include directive is used "to include a file in the main JSP document at the time the document is translated into a servlet" [cf. Hall00, 268]. It is placed in the document at the point at which the file must be inserted.

Example: `<%@ include file="header.html" %>`

In applying the include directive, elements (e.g. navigation elements, headers, footers) can be reused in multiple pages. The included document is inserted before the JSP page is parsed and compiled into a servlet. In other words, it is just as if the code of the included file will has been duplicated into the JSP – except that the JSP container does this at translation time [cf. BaSi04, 402].

Listing 5-2 shows the included file `header.html`.

```
<img src="images/florian.jpg">
<b>Florian Heinisch's personal Website<b>
```

Listing 5-2: The included file "header.html".

[48] Please note that this is the latest JSP specification at the time of writing this thesis.

If the file is included in a JSP, the JSP container simply places the `header.html`'s content into the resulting servlet at the time of translation:

```
private final static char[] _jsp_string2 = "<img
src=\"images/florian.jpg\">\r\n<b>Florian Heinisch's personal
Website<b>".toCharArray();
```

This char-array is written to the servlet's outputstream:

```
out.write(_jsp_string2);
```

If the included document is changed, the JSP has to be retranslated otherwise the changes will not be reflected[49] [cf. Hall00, 267].

5.4.1.3 Taglib Directive

The `taglib` directive in a JSP page declares that the page uses a tag library. The `taglib` directive "uniquely identifies the tag library using a URI, and associates a tag prefix that will distinguish usage of the actions in the library" [cf. Pele01, 52].

The usage of tag libraries will be discussed in detail in chapter 5.5.2 (p. 130).

5.4.2 Scripting Elements

Scripting elements are used to manipulate objects.

Each scripting element has a "`<%`"-based syntax as follows:

```
<%! this is a declaration %>

<% this is a scriptlet %>

<%= this is an expression %>
```

5.4.2.1 Declarations

Declarations are used to declare variables and methods that are available to all other scripting elements. According to the JSP specification, "a declaration

[49] Most of the newer JSP containers detect if the included file has been changed and retranslate the included file automatically. Nevertheless, this automatic retranslation is not guaranteed by the JSP specification [cf. BaSi04, 402].

should be a complete declarative statement" [cf. Pele01, 56]. A declaration is initialized when the JSP page is initialized and is "made available to other declarations, scriptlets and expressions" [cf. Pele01, 56]. As declarations do not generate any output, they are normally used in conjunction with JSP expressions or scriptlets [cf. Hall00, 242].

A declaration is always placed inside the servlet's class but outside the `service()` method [cf. BaSi04, 293].

The following example declares an integer global to the page (the integer's value will be persistent throughout multiple requests until the servlet has been be unloaded or recompiled):

```
<%! int count = 0; %>
```

5.4.2.2 Scriptlets

Scriptlets are used to embed code blocks within the JSP page. Scriptlets are executed at request-processing time and are inserted into the servlet's `service()` method [cf. Pele01, 57].

These code blocks can be used for e.g. setting response headers and status codes, invoking side effects (writing to log-files or updating a database), or executing code that contains loops [cf. Hall00, 238].

The listing 5-3 shows a JSP with an embedded scriptlet.

```
<%@ page import="java.util.Calendar" %>
<html>
  <head>
    <title>AM/PM</title>
  </head>
  <body>
    <%
    if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) {
    %>
      Good morning!
    <% } else { %>
      Good afternoon!
    <% } %>
  </body>
</html>
```

Listing 5-3: "AM_PM.jsp".

5.4.2.3 Expressions

Expressions are used to insert values directly into the output. They are evaluated at HTTP processing time, converted to a string and inserted at the according position in the JSP file [cf. Pele01, 58].

Example: `<%= new java.util.Date() %>`

This expression inserts the current time.

Expressions become the argument to an `out.print()` statement. The JSP container takes every character between `<%=` and `%>` and puts it as the argument to the `PrintWriter` out object [cf. BaSi04, 287]:

```
out.print(new java.util.Date());
```

For this reason an expression must never end with a semicolon! For example, the following expression

```
<% new java.util.Date(); %>
```

results in

```
out.print(new java.util.Date(););
```

this will never compile.

5.4.2.4 Excursus: Bean Scripting Framework

Additionally to the above discussed scripting elements, JSPs can include scripts written in non-Java programming languages with the aid of the Bean Scripting Framework (BSF). Generally, the BSF is “a set of Java classes which provides scripting language support within Java applications [not only within JSPs], and access to Java objects and methods from scripting languages” [Bsf05a]. The BSF provides “an API that permits calling scripting languages from within Java as well as an object registry that exposes Java objects to these scripting languages engines” [BSF05a].

Initially, the BSF was an opensource IBM alphaworks project [IBM99]. In 2002, IBM donated the entire project to the Jakarta project of the Apache organization. At that time, the BSF project got released with the version number “BSF 2.3”. Since then, no further updates of the BSF project have been publicly available [Flat03], [BSF05a].

At the time of writing this thesis, BSF supplies the following scripting language engines [BSF05a]:

- Javascript (using Rhino ECMAScript, from the Mozilla project),
- Python (using either Jython or JPython),
- Tcl (using Jacl),
- NetRexx (an extension of the IBM REXX scripting language in Java),
- XSLT Stylesheets (as a component of Apache XML project's Xalan and Xerces).

In addition, at least the following languages are supported with their own BSF engines [BSF05a], [Flat03]:

- ooRexx (using BSF4Rexx),
- Java (using BeanShell, from the BeanShell project),
- JRuby,

- JudoScript,
- Groovy,
- ObjectScript.

As a complete demonstration of the BSF's usage for its supported scripting languages would be far beyond this thesis' scope the discussion will be limited to how to include JavaScript and ooRexx programs into a JSP. Primarily, BSF's architecture will be discussed.

5.4.2.5 BSF's Architecture

BSF primarily consists of two components:

- BSFManager: it handles "all scripting engines that run under its control and maintains the object registry that permits scripts access to Java objects" [BSF05a].
- BSFEngine: it provides "an interface that must be implemented for the language that the developer wants to use with BSF" [BSF05a] (for the above mentioned programming languages this interface was already implemented). The interface "provides an abstraction of the scripting language's capabilities that permits generic handling of script execution and object registration within the execution context of the scripting language engine" [BSF05a].

5.4.2.5.1 Executing JavaScript inside a JSP

In order to be able to execute JavaScript within a JSP, the developer needs to add the jars `bsf.jar` (available at [BSF05a]) and `js.jar` (available at [Mozi05b])[50] to the Web application's `lib` directory.

Listing 5-4 shows a JSP with embedded JavaScript code.

[50] Please note that the current release of Rhino cannot be used because the Apache BSF project has "not yet released an official version incorporating all the necessary changes to work with Rhino 1.5R4 or later" [Mozi05b]. This fact was confirmed by tests done on Apache's Tomcat Servlet/JSP Container 5.5.7 [Tomc04], IBM's Websphere Application Server 6.0 [IBM05b] and BEA's Weblogic Server 9.1 [Bea06].

```
<%@page contentType="text/html"%>
<%@page import="org.apache.bsf.BSFEngine" %>
<%@page import="org.apache.bsf.BSFException" %>
<%@page import="org.apache.bsf.BSFManager" %>
<html>
  <head>
    <title>BSF JavaScript example</title>
  </head>
  <body>
  <%
  BSFManager mgr = new BSFManager();

  String pathInfo = (request.getPathInfo() != null) ?
    request.getPathInfo() : "";
  String path = request.getContextPath() +
    request.getServletPath() + pathInfo;

  try {
    mgr.declareBean("out", out, out.getClass());
    String script = "var date = new Date();\n" +
                    "var day = date.getDay();\n" +
                    "var weekday = new Array('Sunday', 'Monday', " +
                    "'Tuesday', 'Wednesday', 'Thursday', 'Friday', " +
                    "'Saturday');\n" +
                    "out.println(weekday[day]);";

    BSFEngine engine = mgr.loadScriptingEngine("javascript");
    engine.exec(path, 1, 0, script);
  }
  catch(BSFException ex) {
    ex.printStackTrace();
  }
  %>
  </body>
</html>
```

Listing 5-4: "bsf_javascript.jsp".

Firstly, a new instance of `BSFManager` is created. Secondly, the JSP's path is obtained that will be needed to pass as argument to execute the JavaScript. Next, the implicit object `out` (see chapter 5.4.4, p. 121) is registered with the `BSFManager` object in order to be able to print the script's result to the JSP. The string `script` contains the actual JavaScript that needs to be executed. Then the `BSFEngine` scripting engine for JavaScript is loaded and finally the JavaScript is executed which prints the current week day to the JSP.

5.4.2.5.2 Executing Object Rexx inside a JSP

The execution of Object Rexx (ooRexx) is not by default supported by the Bean Scripting Framework. In order to be able to use ooRexx within a Web application, the system on which the Web server is running needs to have a supported

Rexx interpreter installed[51]. Furthermore, the developer needs to obtain the BSF4Rexx software package which allows BSF to deploy ooRexx as an additional scripting language for Java (the latest distribution is available at [BSF406]). The BSF4Rexx software package primarily consists of two Java class libraries (`bsf-rexx-engine.jar` and `bsf-v205-20060203.jar`[52]) and a compiled C++ programm[53]. The libraries `bsf-rexx-engine.jar` and `bsf-v205-20060203.jar` need to be added to the Web application's `lib` directory.

Listing 5-5 shows a JSP with embedded ooRexx code.

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@page import="org.apache.bsf.BSFEngine" %>
<%@page import="org.apache.bsf.BSFException" %>
<%@page import="org.apache.bsf.BSFManager" %>
<html>
  <head>
    <title>BSF ooRexx example</title>
  </head>
  <body>
  <%
  BSFManager mgr = new BSFManager();

  try {
    mgr.declareBean("out", out, out.getClass());

    BSFEngine engine = mgr.loadScriptingEngine("rexx");
    String script =
      "out=bsf.lookupBean('out')                                  \n" +
      "directions=.array~of('North', 'South', 'East', 'West')     \n" +
      "do entry over directions                                   \n" +
      "   out~write(entry'<br>')                                  \n" +
      "end                                                        \n" +
      "::requires BSF.CLS                                         \n" ;
    engine.eval("rexx", 0, 0, script);
  }
  catch(BSFException ex) {
    ex.printStackTrace();
  }
  %>

  </body>
</html>
```

Listing 5-5: "bsf_oorexx.jsp".

[51] For further information on how to install an ooRexx interpreter on your system please refer to [ooRe06].

[52] Please note that the name of this jar-file might vary with upcoming releases.

[53] As a detailed description of the BSF4Rexx's architecture and installation is far beyond the scope of this thesis, the reader is asked to refer to the documentation at [Flat03], [Flat04] and [Flat06].

Similar to the JSP with embedded JavaScript (see listing 5-4, p. 106) a new instance of `BSFManager` is created. Again, the implicit object `out` is registered with the `BSFManager` object. The string `script` contains the actual ooRexx code that needs to be executed. In the first line of the script a reference to the implicit Java object `out` is obtained. The ooRexx object `out` now represents the Java object `out` and allows the developer to call (within the ooRexx script) the Java object's methods on the proxy ooRexx object `out`. Next, an ooRexx array with four values is created. In line three to five the script iterates over the array. In line four the method `write()` is called on the ooRexx object `out` in order to print the array's values to the JSP. With the last line of the ooRexx script support for Object Rexx is loaded [Flat04].

5.4.3 Action Elements

Actions provide the possibility "to perform sophisticated tasks like instantiating objects and communicating with server-side resources without requiring Java coding" [Sesh00]. Even though the same objective can be achieved with scriptlets, using action tags endorses "reusability of [...] components and enhances the maintainability of applications" [Sesh00]. With actions, files can be inserted dynamically, JavaBeans can be reused, the user can be forwarded to another page, or HTML for the Java plugin can be generated. Available actions include [Hall00b][54]:

- `<jsp:useBean>` – find or instantiate a JavaBean,
- `<jsp:getProperty>` – insert the property of a JavaBean into the output,
- `<jsp:setProperty>` – set the property of a JavaBean,
- `<jsp:include>` – include a file at the time the page is requested,
- `<jsp:forward>` – forward the requester to a new page,

[54] Only common Action elements will be discussed. For a complete listing please refer to the current JavaServer Pages™ 2.0 specification available at [PeRo03].

- `<jsp:plugin>` – generate browser-specific code that makes an `<object>` or `<embed>` tag for the Java plugin.

5.4.3.1 jsp:useBean

This action provides the possibility to load in a JavaBean to be used in the JSP page.

5.4.3.1.1 JavaBean - Definition

“The JavaBeans API provides a standard format for Java classes” [cf. Hall00, 287]. A JavaBean is often referred to simply as a Bean. A Bean is a “reusable software component[55] that is written in Java programming language” [Stear00]. Visual composition/manipulation tools can automatically discover information about classes that follow this format and can then create and manipulate the classes without having to write any code [Stear00].

Listing 5-6 shows the JavaBean `UserBean.java`.

```
public class UserBean {
  private String name;
  private int id;
  private String email;

  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```

Listing 5-6: “UserBean.java”.

[55] “Software components are self-contained, reusable software units” [Stear00].

Detailed explanation of JavaBeans would go beyond the scope of this thesis. For further information and documentation concerning JavaBeans please refer to Sun Microsystem's JavaBean Web site at [Bean05].

5.4.3.1.2 Basic Bean Use

Before a Bean can be accessed within a JSP page, it is necessary to identify the Bean and obtain a reference to it.

The syntax for inserting a JavaBean is:

```
<jsp:useBean id="beanInstanceName" class="bean class"
scope = "page | request | session |application" />
```

When the `<jsp:useBean>` tag is processed, the JSP container "performs a lookup of the specified given Java object using the values specified in the `id` and `scope` attributes. If the object is not found, it will attempt to create it using the values specified in the `scope` and `class` attributes" [cf. WaFi00, 111].

Table 5-1 lists the `<jsp:useBean>` attributes [Hall00b].

Attribute	Description
`id`	Assigns “a name to the variable that will reference the bean. A previous bean object is used instead of instantiating a new one if one can be found with the same `id` and `scope`” [Hall00b].
`class`	Indicates the “fully qualified name of the class that defines the implementation of the object” [cf. PeRo03, 1-104].
`scope`	“Indicates the context in which the Bean should be made available” [Hall00b]. There are four possible values: 1. `page`: the bean is accessible only within the page where it was created, 2. `request`: “the bean is only available for the current client request” [Hall00b], 3. `session`: the bean “is available to all pages during the life of the current `HttpSession`” [Hall00b], 4. `application`: the bean “is available to all pages that share the same `ServletContext`” [Hall00b]. The `scope` attribute is important because “a `jsp:useBean` entry will only result in a new object being instantiated if there is no previous object with the same `id` and `scope`” [Hall00b].
`type`	Defines “the type of the variable that will refer to the object” [Hall00b]. The type must be ”either the class itself, a superclass of the class, or an interface implemented by the class specified” [cf. PeRo03, 1-105].
`beanName`	This is the “name of a bean, as expected by the `instantiate` method of the `java.beans.Beans` class” [cf. PeRo03, 1-104].

Table 5-1: "jsp:useBean” attributes.

Example: `<jsp:useBean id="user" class="thesis.UserBean" />`

This example tries to locate an instance of the `UserBean` class. If no instance exists, a new instance will be created. The instance can then be accessed within the JSP page using the specified `id` (in this example `user`) of `UserBean`. The equivalent scriptlet to that `<jsp:useBean>` action tag would be:

```
<% thesis.UserBean user = new thesis.UserBean (); %>
```

5.4.3.2 jsp:getProperty

As soon as a Bean has been declared with `<jsp:useBean>`, its properties can be accessed through the `<jsp:getProperty>` tag [cf. WaFi00, 113].

The syntax for the tag `<jsp:getProperty>` is:

```
<jsp:getProperty
      name="beanInstanceName"
      property="propertyName"
/>
```

Table 5-2 lists `<jsp:getProperty>` attributes [cf. WaFi00, 114], [Hall00b].

Attribute	Description
`name`	This required attribute specifies “the name (id) of the bean instance specified in the `jsp:useBean` tag” [cf. WaFi00, 114].
`property`	This required attribute indicates the name of the bean's property to be obtained.

Table 5-2: "jsp:getProperty" attributes.

Example:

```
<jsp:getProperty name="user" property="id" />
```

The JavaBean `user`’s property `id` will be printed to the output.

5.4.3.3 jsp:setProperty

The `<jsp:setProperty>` tag is used to set values “to properties of Beans that have been referenced earlier” [Hall00b]. This can be done in two contexts:

1. `<jsp:setProperty>` can be used outside of the corresponding `<jsp:useBean>` element. In this case, the `<jsp:setProperty>` “is executed regardless of whether a new Bean was instantiated or an existing Bean was found” [Hall00b].

2. `<jsp:setProperty>` can be placed inside the body of a `<jsp:useBean>` element. The `<jsp:setProperty>` “is executed only if a new object [i.e. a bean] was instantiated [by the surrounding `<jsp:useBean>` action element], not if an existing one was found” [Hall00b]. That means, that if the bean referenced by `<jsp:useBean>` already existed, the `<jsp:setProperty>` action will not be executed (i.e. it is conditional), consequently the existing bean's property value will not be reset.

The syntax for the `<jsp:setProperty>` tag is:

```
<jsp:setProperty name="beanInstanceName" prop_expr />
```

Table 5-3 lists `<jsp:setProperty>` attributes [Hall00b], [cf. PeRo03, 1-105].

Attribute	Description
name	This required attribute specifies “the name (id) of the bean instance specified in the jsp:useBean tag” [cf. WaFi00, 115]. “The bean instance must contain the property” [cf. PeRo03, 1-107] that is intended to be set.
property	This required attribute indicates the property that is intended to be set. If a value of “*” is set, “all request parameters whose names match Bean property names will be passed to the appropriate setter methods” [Hall00b].
value	This optional attribute sets the value of the defined property. It “can accept a request-time attribute expression as a value” [cf. PeRo03, 1-107]. An action may not have both param and value attributes.
param	This optional attribute defines “the name of a request parameter whose value is given to a bean property” [cf. PeRo03, 1-107]. If param is omitted, “the request parameter name is assumed to be the same as the Bean property name” [cf. PeRo03, 1-107].

Table 5-3: "jsp:setProperty" attributes.

Example:

```
<jsp:setProperty name="user" property="email"
value="florian@heinisch.cc" />
```

5.4.3.4 jsp:include

The jsp:include action includes a file at request time. So the JSP page does not have to be retranslated into a servlet (in contrast to the include directive) when the included file is changed [cf. Hall00, 270].

The page that is included can not set HTTP headers (such as setting the HTTP response status code). Any attempts to set HTTP header will be simply ignored [PeRo03, 1-109].

The syntax for the <jsp:include> tag is:

```
<jsp:include page="UrlSpec" flush="true|false" />
```

Table 5-4 lists `<jsp:include>` attributes [cf. Hall00, 271], [cf. PeRo03, 1-109].

Attribute	Description
`page`	A relative URL (i.e. relative to the current JSP page) referencing the file that has to be included.
`flush`	Optional boolean attribute which controls flushing. If the value is `true`, the buffer is flushed prior to the inclusion (presuming that the page output is buffered)[56]. The default value is `false`.

Table 5-4: "jsp:include" attributes.

Example:

```
<jsp:include page="header.html" />
```

In contrast to the include directive where the source code of the included file is simply copied (see chapter 5.4.1.2, p. 100), the include action inserts the response of the included file at runtime. In the resulting servlet the include directive is translated into following statement (in bold) and placed within the `service` method at the according place [cf. BaSi04, 402]:

```
…
out.write(_jsp_string1);
org.apache.jasper.runtime.JspRuntimeLibrary.include(request, response, "header.html", out, false);
out.write(_jsp_string2);
…
```

5.4.3.5 jsp:forward

The `jsp:forward` element "allows the runtime dispatch of the current request to a static resource, a JSP page or a Java servlet class in the same context as the current page. A `jsp:forward` tag effectively terminates the execution of the current page" [cf. Pele01, 79].

The syntax for the `jsp:forward` tag is:

[56] The output of a JSP page can be buffered. Buffering is outside the scope of this thesis. For further information please refer to [PeRo03, 1-46].

```
<jsp:forward page="relativeURLspec" />
```

Example:

```
<jsp:forward page="shopping_cart.jsp" />
```

5.4.3.6 jsp:plugin

The `jsp:plugin` action provides the possibility to insert the Web browser-specific `OBJECT` or `EMBED` element that is needed to be declared in order to embed an applet in a Web browser [cf. Pele01, 81].

The syntax for the `<jsp:plugin>` tag is:

```
<jsp:plugin type="bean|applet" code="objectCode" />
```

The attributes of the `<jsp:plugin>` tag supplies “configuration data for the presentation of the element” [cf. Pele01, 81]. The most commonly used attributes[57] are listed in table 5-5 [cf. Hall00, 275].

Attribute	Description
`type`	Identifies the type of the component: a Bean or an Applet.
`code`	Specifies “the top-level applet class file that extends `Applet` or `JApplet`”.
`width`	Specifies “the width in pixels to be reserved for the applet” [Hall00b]. “Accepts a run-time expression value” [cf. PeRo03, 1-115].
`height`	Specifies “the height in pixels to be reserved for the applet” [Hall00b]. “Accepts a run-time expression value.” [cf. PeRo03, 1-114].

Table 5-5: "jsp:plugin" attributes.

[57] For a complete listing of the `<jsp:plugin>` attributes please refer to the JavaServer Pages™ 2.0 specification available at [PeRo03].

5.4.3.7 Example: JSP Using a JavaBean

This example shall demonstrate how to use standard actions in order to create a JavaBean, store an HTML-form's data in the JavaBean and finally display the JavaBean's properties to the user.

Therefore created two files were created:

1. `userForm.html`: contains a simple HTML-form where the user enters his name, his id and his email address (see listing 5-7).

2. `userBean.jsp`: creates a JavaBean, assigns the form-data to the according JavaBean properties and displays the data to the user (see listing 5-8).

```
<html>
  <head>
    <title>User Form</title>
  </head>
  <body>
    <h1>User Form</h1>
    <form method="post" action="userBean.jsp">
      Name<br>
      <input name="name" type="text" id="name" size="20"><br>
      ID<br>
      <input name="id" type="text" id="id" size="20"><br>
      Email<br>
      <input name="email" type="text" id="email" size="20"><br>
      <input name="submit" type="submit" value="Submit">
    </form>
  </body>
</html>
```

Listing 5-7: "userForm.html".

The HTML tag `<form method="post" action="userBean.jsp">` indicates that the form-data will be transmitted via the HTTP method POST to be processed by the file `userBean.jsp`, as shown in listing 5-8.

```
<html>
  <head>
    <title>User Bean</title>
  </head>
  <body>
    <h1>User data</h1>
    <jsp:useBean id="user" class="cc.heinisch.thesis.UserBean"
        scope="session">
      <jsp:setProperty name="user" property="*" />
    </jsp:useBean>
    Name: <jsp:getProperty name="user" property="name" /><br>
    ID: <jsp:getProperty name="user" property="id" /><br>
    Email: <jsp:getProperty name="user" property="email" /><br>
  </body>
</html>
```

Listing 5-8: "userBean.jsp".

The `userForm.html`'s filled out form displayed in a Web browser looks as shown in figure 5-3.

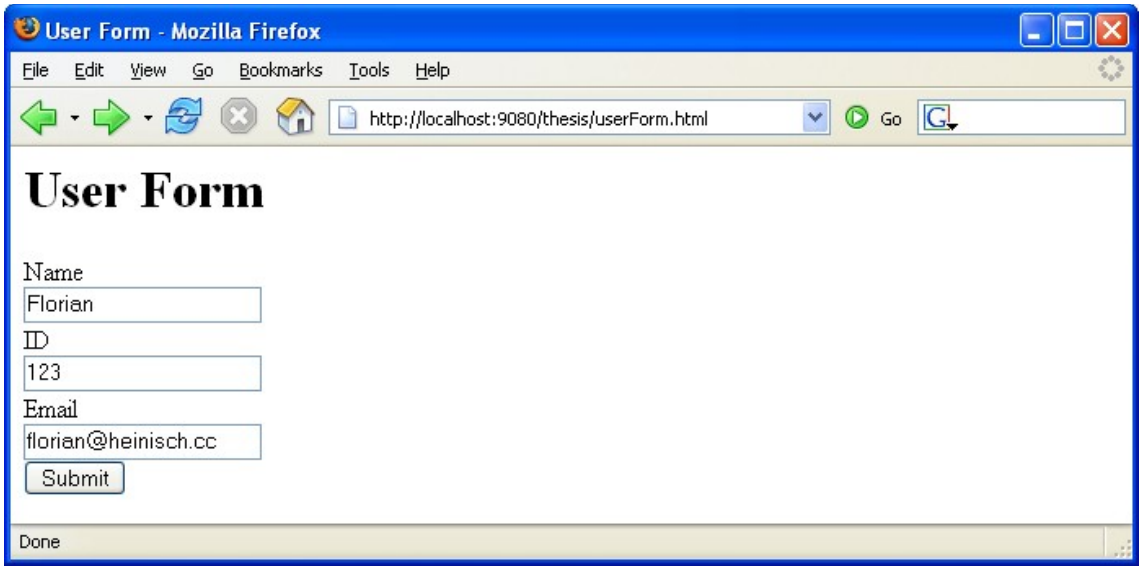

Figure 5-3: "userForm.html" displayed in a Web browser.

As explained in chapter 5.4.3.1 (p. 109), the action `<jsp:useBean />` causes the JSP container to look for a Java object specified by the attribute `id` in the specified scope. If there is no scope attribute, the default scope is `page`. If the container cannot find the referenced JavaBean in the specified scope (based on the attributes `id` and `scope`) it will create a new instance of the class that is specified in the `class` attribute. In this example, the container does not find an instance of `UserBean` referenced by `id="user"` in the scope `session` so it creates a new instance and assigns it to the scope `session`.

Next, the nested `<jsp:setProperty name="user" property="*" />` tag will pass all request parameters (i.e. `name`, `id` and `email`) to the according setter methods of the JavaBean `UserBean` (see chapter 5.4.3.3, p. 112). Consequently, the JavaBean is populated with the form-data.

The next step is to retrieve the JavaBean's properties. Since the scope of the JavaBean was assigned the value `session`, the JavaBean's instance will be accessible as long as the session remains valid. In order to retrieve one of the JavaBean's properties, the statement `<jsp:getProperty name="user" property="name" />` is used in the JSP where the property shall be displayed. The attribute name references the JavaBean instance that was specified by the id attribute within the `<jsp:useBean id="user" … />` tag and the attribute `property` defines the JavaBean's property that shall be retrieved.

If the form's button "Submit" of the file `userForm.html` (see 5-3, p. 118) is hit, the page as shown in figure 5-4 is displayed.

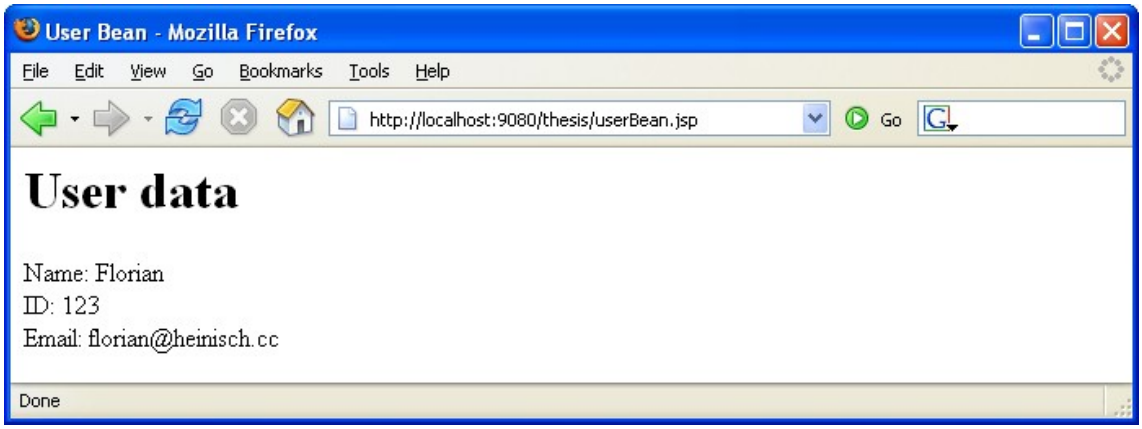

Figure 5-4: "userBean.jsp" diplayed in a Web browser.

If scriptlets and expressions were used instead of actions the JSP `userBean.jsp` would like as shown in listing 5-9.

```
<html>
  <head>
    <title>User Bean Scriptlet</title>
  </head>
  <body>
    <h1>User data</h1>
    <%
    cc.heinisch.thesis.UserBean user =
      new cc.heinisch.thesis.UserBean();
    session.setAttribute("user", user);
    user.setName(request.getParameter("name"));
    user.setId(Integer.parseInt(request.getParameter("id")));
    user.setEmail(request.getParameter("email"));
    %>
    Name: <%= user.getName() %> <br>
    ID: <%= user.getId() %> <br>
    Email: <%= user.getEmail() %> <br>
  </body>
</html>
```

Listing 5-9: "userBean_scriptlet.jsp".

First, a new UserBean instance is instantiated and assigned it to the implicit object (see chapter 5.4.4, p. 121) `session`.

In order to set the form field's data to according bean properties, the corresponding JavaBean's setter method is called and the field's data (which is obtained by the statement `getParameter("fieldname")`) is passed as the argument. As the retrieved form-data are of type string and the JavaBean's property `id` is of type `int`, it needs to be converted to int before it can be passed as an argument to the `setId()` setter method. The `<jsp:setProperty />` tag converts primitive properties automatically which makes the populating of JavaBeans very convenient [cf. BaSi04, 361].

Finally, the JavaBean's properties are displayed with expressions by calling the according JavaBean's getter methods.

As the JavaBean was assigned the scope `session`, the JavaBean's properties can be retrieved in any other JSP as long as the session is still active. With actions, such a property can simply be retrieved by

```
<jsp:getProperty name="user" property="name" />
```

If scriptlets and expressions are used, this is not as simple as with actions:

```
<%
cc.heinisch.thesis.UserBean user =
(cc.heinisch.thesis.UserBean)session.getAttribute("user");
%>
Name: <%= user.getName() %>
```

At first, the JavaBean needs to be referenced. Because the statement `session.getAttribute("user")` returns a object of type `Object` it needs to be casted to the required type (in this case `UserBean`). Then the according getter method can be called.

5.4.4 Implicit Objects

When the JSP container translates the JSP into a servlet, it declares and assigns implicit objects at the beginning of the `service()` method. With implicit objects, developers can "access container-provided services and resources" [Gabh03b]. These objects are called implicit because developers do not have to explicitly declare them. As they are declared automatically by the container, developers only need to use the reference variable associated with the implicit object to begin calling methods on it [cf. BaSi04, 296].

In listing 5-1 (see p. 97), the following implicit objects are declared:

```
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
```

Implicit objects are "always available for use within scriptlets and scriptlet expressions" whereas "each implicit object has a class or interface type that is defined in a core Java technology or in the Java Servlet API package" [cf. PeRo03, 1-40].

According to the JSP Specification, there are nine implicit objects. Table 5-6 provides a brief description of each one's function and the according Java type [Gabh03b], [cf. PeRo03, 1-41].

Variable name	Type	Description
`application`	`javax.servlet.ServletContext`	“Allows the JSP page's servlet and any Web components contained in the same application to share information” [Gabh03b].
`config`	`javax.servlet.ServletConfig`	“Allows initialization data to be passed to a JSP page's servlet” [Gabh03b].
`exeption`	`java.lang.Throwable`	Contains “exception data that can be accessed only by designated JSP error pages” [Gabh03b].
`out`	`javax.servlet.jsp.JspWriter`	“Provides access to the servlet's output stream” [Gabh03b].
`page`	`java.lang.Object`	“Is the instance of the JSP page's servlet processing the current request[58]” [Gabh03b].
`pageContext`	`javax.servlet.jsp.PageContext`	Provides access to the `pageContext` object for this JSP page.
`request`	`javax.servlet.http.HttpServletRequest`	“Provides access to HTTP request data” [Gabh03b].
`response`	`javax.servlet.http.HttpServletResponse`	Provides “direct access to the `HTTPServletResponse` object” [Gabh03b].
`session`	`javax.servlet.http.HttpSession`	Is a reference to “the session object created for the requesting client (if any)” [cf. PeRo03, 1-41].

Table 5-6: Implicit objects.

[58] `page` is a synonym for `this` in the body of the page [cf. PeRo03, 1-41].

5.5 Script-free JSP Pages

Although scriptlets provide a fast and convenient way to add dynamic content to a JSP, scriptlets “introduce more long-term complexity [to JSP pages] than they offer in terms of short-term benefit” [McLa03]. Scriptlets pose the following problems [McLa03]:

- As scriptlets mix HTML with Java code, authoring and debugging becomes problematic.
- Scriptlets are not reusable. As a consequence, if the same scriptlet is needed in different JSP pages, the scriptlet has to be placed into each JSP page which “results in multiple versions of the same Java code” [McLa03].
- Since scriptlets have no clean-cut way to display script errors, they make error reporting difficult.

In chapter 5.4.3 (p. 108) JSP standard actions were introduced which provide a way to reduce the usage of scriptlets. But usually, developers need more functionality than the JSP standard actions offer. However, there is no need to resort to scripting: since JSP 1.1 it is possible to create ones own JSP actions in the form of custom tags (which are extensions of the JSP language).

Fortunately, there is a standard library of custom tags known as the JSP Standard Tag Library (JSTL 1.1) which offers a wide range of common functionality so that there should be no need to develop ones own custom tags in order to produce script-less JSPs [cf. BaSi04, 435].

JSTL makes extensive use of the JSP expression language (EL). But as the expression language is now part of the JSP Specification (since version 2.0) it will be discussed separately before moving to the discussion of how to use custom tags, how to use the JSTL libraries and how to create ones own custom tags.

5.5.1 Expression Language

The JSP Specification 2.0 came up with an important new feature: the JSP expression language (EL). The concept of the expression language was primarily introduced with the JSP Standard Tag Library (JSTL) but its use was limited to the JSTL tags only (see chapter 5.5.3, p. 132). However, since JSP 2.0 was finalized, the expression language is supported throughout JSP pages [Thom05].

5.5.1.1 Expression Language Definition

The Expression Language (EL) is "inspired by [...] the ECMAScript and the XPath expression languages" [cf. PeRo03, 1-64]. It provides a way to simplify expressions in JSP. The expression language can be used in attribute values for standard and custom actions as well as within template text. The expression language is invoked consistently via the construct `${expr}` (`expr` stands for a valid expression). Consequently, EL expressions are always contained within curly braces and prefixed with the dollar sign.

5.5.1.2 Valid Expressions

Valid expressions can include variables (i.e. object references), literals and operators, reserved words or predefined implicit objects [Thom05].

5.5.1.2.1 Variables for Object Access

The data of a Web application in a JSP usually consists of objects that "comply with the JavaBeans specification, or that represent collections such as lists, maps or arrays" [cf. Deli03, p. 15].

Therefore, "a core concept in the EL is the evaluation of a variable name into an object" [cf. PeRo03, p.1-75]: the EL evaluates the variable name "by looking up its value as an attribute" [cf. PeRo03, p.1-75]. For example:

```
${user}
```

This expression looks for the attribute named `user` by "searching the page, request, session and application scopes" [cf. PeRo03, p. 1-75]. If it finds the object it will return its value – if not, `null` is returned. But if the variable name

matches one of the implicit objects (see chapter 5.4.4, p. 121), the implicit object instead of the variable value will be returned. Consequently, the variable name in an expression is either an implicit object or an attribute in one of the four scopes.

Furthermore, the EL provides two operators in order to access the data that are encapsulated in the returned object: "`.`" and "`[]`".

The dot operator

The dot operator can be used to access properties of a JavaBean or values of a map (this is valid for both implicit objects or attributes). For example:

```
${user.name}
```

If the EL expression contains a variable followed by a dot there are three rules:

1. the variable must be a map or JavaBean,
2. the identifier after the dot "must be a map key or a Bean property" [cf. BaSi04, p. 368],
3. the identifier after the dot "must follow normal Java naming rules for identifiers" [cf. BaSi04, p. 368].

The variable in the expression above is either a JavaBean that has the property `name` and the according getter- and setter-methods (i.e. `getName()`, `setName()`) or a map that has a key `name` [cf. BaSi04, p. 368].

The expression `${user.name}` produces the same result as the JSP action `<jsp:getProperty name="user" property="name" />` that was used in the example to demonstrate the usage of actions (see chapter 5.4.3.7, p. 117).

The bracket operator

The bracket `[]` operator can be used to access JavaBeans' properties, map values (as with the dot-operator) and also to access lists and arrays. Additionally, the characters within the brackets do not have to follow Java naming rules

for identifiers. JavaBean properties or map values are simply accessed by placing the property's name or the map's key as a string literal into the brackets [cf. BaSi04, p. 370]. For example:

```
$user["name"]
```

This expression produces the same result as `${user.name}`.

The EL for accessing an array or a list is the same. For example:

```
$array[0]
```

Provided that `array` is an array (or a list) who's first value has the index `0` this expression returns the array's first value. The following expression produces the same result:

```
$array["0"]
```

If the characters within the brackets (i.e. the index) are a string literal then "the index is coerced to an int" [cf. BaSi04, p. 372].

5.5.1.2.2 Arithmetic, Logical and Relational Operators

EL expression may also contain calculations and logic. The following sections list the available operators.

Arithmetic Operators

Arithmetic operators are provided "to act on integer [...] and floating point [...] values" [cf. PeRo03, p. 1-69]. There are five operators :

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/` and `div`
- Remainder: `%` and `mod`

Logical Operators

There are three logical operators [cf. PeRo03, p. 1-73]:

- AND: `&&` and `and`
- OR: `||` and `or`
- NOT: `!` and `not`

Relational Operators

There are six relational operators [cf. PeRo03, p. 1-71]:

- Equals: == and `eq`
- Not equals: `!=` and `ne`
- Less than: < and `lt`
- Greater than: > and `gt`
- Less than or equal to: <= and `le`
- Greater than or equal to: >= and `ge`

5.5.1.2.3 Reserved Words

There are currently 16 reserved words defined by the JSP Specification that “should not be used as identifiers” [cf. PeRo03, p. 1-75]. Eleven of these words were already introduced in chapter 5.5.1.2.2, p. 126 (i.e. `div`, `mod`, `and`, `or`, `not`, `eq`, `ne`, `lt`, `gt`, `le`, `ge`). Although the JSP Specification 2.0 states that “many of these words are not in the language now but they may be in the future” [cf. PeRo03, p. 1-75] only one word is not in the language at the time of writing. Table 5-7 lists the remaining five reserved words.

Reserved word	Description
`empty`	An operator to see if something is null or empty. E.g. `${empty user}` returns null if `user` is empty or null.
`false`	Boolean literal.
`instanceof`	Currently not defined.
`null`	Simply means `null`.
`true`	Boolean literal.

Table 5-7: Reserved words.

5.5.1.2.4 Implicit Objects

The expression language defines a set of eleven implicit objects – but these are not the same as the JSP implicit objects (except for the `pageContext` object). All but one (the `pageContext` object) are simple Maps [cf. BaSi04, p. 381].

Table 5-8 provides an overview and a description of the EL implicit objects [cf. PeRo03, p. 1-66].

EL implicit object	Description
`pageContext`	This is the context for the JSP page which can be used to access the JSP implicit objects (such as `request`, `response`).
`param`	“A map that maps parameter names to a single String parameter value” [cf. PeRo03, p. 1-66].
`paramValues`	“A map that maps parameter names to a `String[]` [i.e. a String array] of all values to that parameter” [cf. PeRo03, p. 1-66].
`header`	“A map that maps header names to a single String header value” [cf. PeRo03, p. 1-66].
`headerValues`	“A map that maps header names to a `String[]` [i.e. a String array] of all values for that header” [cf. PeRo03, p. 1-66].
`cookie`	“A map that maps cookie names to a single Cookie object” [cf. PeRo03, p. 1-66].
`initParam`	“A map that maps context initialization parameter names to their String parameter value” [cf. PeRo03, p. 1-67].
`pageScope`	“A map that maps page-scoped attribute names to their values” [cf. PeRo03, p. 1-66].
`requestScope`	“A map that maps request-scoped attribute names to their values” [cf. PeRo03, p. 1-66].
`sessionScope`	“A map that maps session-scoped attribute names to their values” [cf. PeRo03, p. 1-66].
`appliciationScope`	“A map that maps application-scoped attribute names to their values” [cf. PeRo03, p. 1-66].

Table 5-8: EL implicit objects.

Examples:

`${pageContext.request.method}` returns the HTTP request's method.

`${param.id}` is equivalent to `<%= request.getParameter("id") %>`

`${header.host}` is equivalent to `<%= request.getHeader("host") %>`

`${sessionScope.user.name}` only looks for a Map or a JavaBean in the scope session (if `sessionScope` is omitted then the expression would look in all four scopes).

5.5.2 Using Customs Tags

"Custom tags are user-defined JSP language elements" [ArBa04] (i.e. actions) that encapsulate a specific functionality. Custom tags accomplish the same goal as JSP standard actions: complex code is encapsulated into a simple and accessible form that is available for (re)use [cf. Hall00, 309].

Custom tags "do not offer more functionality than scriptlets, they simply provide better packaging" [Mahm01] of source code in order to "improve the separation of business logic and presentation logic" [Mahm01]. The core benefits of custom tags are:

- Scriptlets can be reduced or rather eliminated: consequently, the syntax of the JSP page is kept clean and simple.
- Custom tags are reusable: consequently, "they save development and testing time" [Mahm01].

Custom tags are distributed in a tag library which contains two components that work together [Spiel01]:

- Tag handlers: a Tag handler is a Java class that contains the functional logic for a custom tag.
- Tag library descriptor (TLD): the TLD is an XML document that describes the library. It contains "information about the library as a whole and about each tag contained in the library" [Spiel01].

In order to use a custom tag in a JSP, the developer must [ArBa04]:

- declare the tag library that contains the tag,
- “make the tag library implementation available to the Web application” [ArBa04].

5.5.2.1 Syntax

There are two types of custom tags and both can have attributes [ArBa04]:

- Bodyless custom tags: a bodyless tag is an empty element. It has the syntax:

  ```
  <prefix:tag attr1="value" ... attrN="value" />
  ```

- Custom tags with a body: a custom tag with a body has a start tag and a matching end tag. It has the syntax:

  ```
  <prefix:tag attr1="value" ... attrN="value" >
    body
  </prefix:tag>
  ```

  `prefix` distinguishes tags for a library, tag identifies the library’s tag and `attri1 ... attriN` are attributes that modify the behavior of a tag.

5.5.2.2 Declaring the Tag Library

In order to declare that a JSP page will use tags defined in a tag library, the developer must include a taglib directive in the JSP “before any custom tag from that library is used” [ArBa04]:

```
<%@ taglib prefix="tt" uri="URI" %>
```

The attribute `prefix` is used to define the prefix that “distinguishes the tags defined by a given tag library from those provided by other tag libraries” and “the attribute `uri` refers to a URI that uniquely identifies the tag library descriptor (TLD)” [ArBa04].

5.5.2.3 Including the Tag Library Implementation

Finally, in order to be able to use custom tags in ones JSP, the developer “must make the tag library implementation available to the Web application” [ArBa04]. The tag library can be included in an unpacked or packed format:

- Unpacked format: the TLD-file has to be placed either directly in the `/WEB-INF` directory or inside a sub-directory of `/WEB-INF` (e.g. `/WEB-INF/tld`). The tag handlers have to be packaged in the `/WEB-INF/classes` directory.

- Packed Format: the tag library is packaged into a JAR-file[59] and simply has to be included in the `/WEB-INF/lib` directory.

5.5.3 Java Standard Tag Library (JSTL)

There are many development teams that “implemented a set of custom tag libraries to accomplish basic Java Bean manipulation and conditional logic functionality” [cf. Holm04, 333] for their projects. Even though many of these implementations are quite similar, each is a little different in the scope of functionality and each uses different names for its tags and attributes. Due to the “duplication of effort across multiple projects, the need arose for a common, standardized set of tag libraries that can be used universally” [cf. Holm04, 333]. Furthermore, a standardized set of tags eliminates “the need to learn the details of different tag libraries” [cf. Holm04, 333]. Therefore, the JSP Standard Tag Library (JSTL) was created.

The JSTL’s tag libraries provide “a set of tags that implement general-purpose functionality for

- iteration and conditional processing,

- data formatting and localization,

- XML manipulation,

- database access” [cf. Holm04, 333],

[59] For detailed information on JARs please refer to [Sun05b].

- String manipulation [cf. Deli03, 2].

5.5.3.1 JSTL Tag Libaries

The JSTL's tag libraries "offer the base functionality needed by most applications" [cf. Holm04, 337]. Like any other JSP tag libraries, the JSTL tag libraries work "with the added functionality of the expression language for tag attribute values" [cf. Holm04, 337].

Table 5-9 lists each of the JSTL tag libraries.

Library	Prefix	Description
Core	`c`	"Provides tags for conditional logic, loops, output, variable creation, text imports and URL manipulation" [cf. Holm04, 337].
Format	`fmt`	"Provides tags for formatting dates, numbers, localizing text messages" [cf. Holm04, 337].
SQL	`sql`	"Provides tags for making SQL queries to databases" [cf. Holm04, 337].
XML	`x`	"Provides tags for parsing of XML documents, selection of XML fragments, flow control based on XML, XLST transformation" [cf. Holm04, 337].
Functions	`fn`	Provides tags for getting a Collection's size and for string manipulation [cf. Deli03, 167].

Table 5-9: JSTL tag libraries.

5.5.3.2 Using JSTL Tag Libraries

In order to use JSTL in JSPs, JSTL-files (i.e. `jstl.jar` and `standard.jar`)[60] need to be added to the Web applications library directory (i.e. `/WEB-INF/lib`, see chapter 4.6, p. 85) and the Tag Library Descriptors (TLDs) have to be referenced from the JSP. Since JSP 2.0, referencing the Tag Library Descriptor is done by simply placing a taglib directive (see chapter 5.4.1.3, p. 101) at the beginning of the JSP, such as [cf. Holm04, 340], [cf. BaSi04, 475]:

```
<%@
taglib prefix="c" uri=" http://java.sun.com/jsp/jstl/core"
%>
```

Table 5-10 lists the URI for each of the JSTL libraries [cf. Holm04, 340], [cf. Deli03, 2].

Library	Prefix	URI
Core	`c`	`http://java.sun.com/jsp/jstl/core`
Format	`fmt`	`http://java.sun.com/jsp/jstl/fmt`
SQL	`sql`	`http://java.sun.com/jsp/jstl/sql`
XML	`x`	`http://java.sun.com/jsp/jstl/xml`
Functions	`fn`	`http://java.sun.com/jsp/jstl/functions`

Table 5-10: JSTL URIs.

The URI is just a unique name for the tag library and not any actual location (such as a path or an URL). The JSP container will not “try to request something from the URI” [cf. BaSi04, 474]. It is a convention Sun Microsystems uses for the URI in order “to help ensure that it is a unique name” [cf. BaSi04, 474].

This chapter does not cover every JSTL tag. Only the usage of a few tags from the Core library will be discussed to give the reader an idea how to use the JSP Standard Tag Library. Please refer to the reference pages at [Sun05c] for a complete list of the JSTL tags and their attributes.

[60] These jars are available at [JSTL04].

5.5.3.2.1 Conditional Tags

The core library provides two different conditionalization tags: `<c:if>` and `<c:choose>`.

`<c:if>` “simply evaluates a single test expression and then processes its body content only if that expression evaluates to true” [Kolb03]. If not, the tag's body content is simply ignored. Listing 5-10 shows the syntax for `<c:if>`.

```
<c:if test="expression" var="name" scope="scope">
  body content
</c:if>
```

Listing 5-10: Syntax for the <c:if> action.

Listing 5-10 shows that `<c:if>` can “optionally assign the result of the test to a scoped variable through its `var` and `scope` attributes” [Kolb03]. This can be especially useful “if the test is expensive: the result can be cached in a scoped variable and retrieved in subsequent calls to `<c:if>` or other JSTL tags” [Kolb03].

In chapter 5.4.3.7 (p. 117), an explanation of how to use a JavaBean was provided. The JavaBean is stored in session scope so the user’s data can be displayed in every JSP as long as the session remains valid. If the session is not valid any more the Java Bean’s data will not be accessible. Therefore, any JSP trying to display the user’s data should test whether the data is still available, as depicted in listing 5-11.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
  <head>
    <title>User data</title>
  </head>
  <body>
    <h1>User data</h1>
    <c:if test="${user != null}">
      Name: ${user.name} <br>
      ID: ${user.id} <br>
      Email: ${user.email} <br>
    </c:if>
  </body>
</html>
```

Listing 5-11: "userInfo1.jsp".

`userInfo1.jsp` performs a simple conditional test whether the user's data can be accessed using the JSTL's `<c:if>` tag. The attribute `test` does the conditional testing using an expression language statement: so if the variable `user` is not `null` then the content within the `<c:if>` tags will be displayed – otherwise it simply will be ignored.

Listing 5-11 demonstrates that the `<c:if>` tag provides a compact notation for simple cases of conditionalized content. The second conditionalization tag provided by the JSTL core library `<c:choose>` is for cases "in which mutually exclusive tests are required to determine what content should be displayed" [Kolb03]. Listing 5-12 shows the syntax for the `<c:choose>` action.

```
<c:choose>
  <c:when test="expression">
    body content
  </c:when>
  ...
  <c:otherwise>
    body content
  </c:otherwise>
</c:choose>
```

Listing 5-12: Syntax for the <c:choose> action.

"Each condition that has to be tested is represented by a corresponding `<c:when>` tag" [Kolb03]. There can be several `<c:when>` tags but there must be at least one. The body content of the first `<c:when>` tag whose test evaluates to `true` will be processed. If none of the `<c:when>` tests return `true` then the body content of the `<c:otherwise>` tag will be processed.

Listing 5-13 shows an example using the `<c:choose>` tag. In the case that the session is not valid any more and consequently the user's data cannot be displayed, the JSP displays an according error message.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
  <head>
    <title>User data</title>
  </head>
  <body>
    <h1>User data</h1>
    <c:choose>
      <c:when test="${sessionScope.user != null}">
        Name: ${user.name} <br>
        ID: ${user.id} <br>
        Email: ${user.email} <br>
      </c:when>
      <c:otherwise>
        The user's data can not be accessed!
      </c:otherwise>
    </c:choose>
  </body>
</html>
```

Listing 5-13: "userInfo2.jsp".

5.5.4 Creating Custom Tags

If JSP standard actions and the JSP Standard Tag Library (or any other tag library) do not provide the functionality needed in a JSP, there is still the possibility to create ones own custom tags.

In this chapter it will be demonstrated how to implement a simple custom tag `date` that is intended to display the current date. Additionally this custom tag can include the attribute `format` that provides the possibility to format the date by declaring a date pattern, such as:

```
<simple:date format="d/MM/yyyy" />
```

The following steps are necessary, to implement a custom tag [cf. BaSi04, 503]:

- write the tag handler class,
- create the tag library descriptor,
- deploy the tag handler and the tag library descriptor,
- write a JSP that uses the tag.

5.5.4.1 Tag Handler Class

The tag handler class determines what to do when a tag is referenced. The functionality of a custom tag is packaged within this class. It is “invoked by the JSP runtime to evaluate the custom tag during the execution of a JSP page that references the tag” [Mahm01].

The JSP Specification defines two different types of tag handlers:

- Simple tag handlers: “can be used only for tags that do not use scripting elements in attribute values or the tag body” [ArBa04].
- Classic tag handlers: “must be used if scripting elements are required” [ArBa04].

Simple tag handlers were introduced with JSP 2.0. Compared to classic tag handlers (which were introduced with JSP 1.1) whose API and invocation protocol is rather complex, simple tag handlers provide “an easier way to implement custom actions” [Thom05].

As the purpose of this thesis is to provide an overview of JSP, the discussion will be limited to the development of simple tag handlers without bodies. For further information concerning the development of custom tags please refer to [Sun05d].

A simple tag handler must implement the `SimpleTag` interface. A convenient way to do so is to extend the `SimpleTagSupport` class that provides a default implementation for all methods in `SimpleTag`. The class diagram in figure 5-5 the `SimpleTag` interface and the convenience class `SimpleTagSupport`.

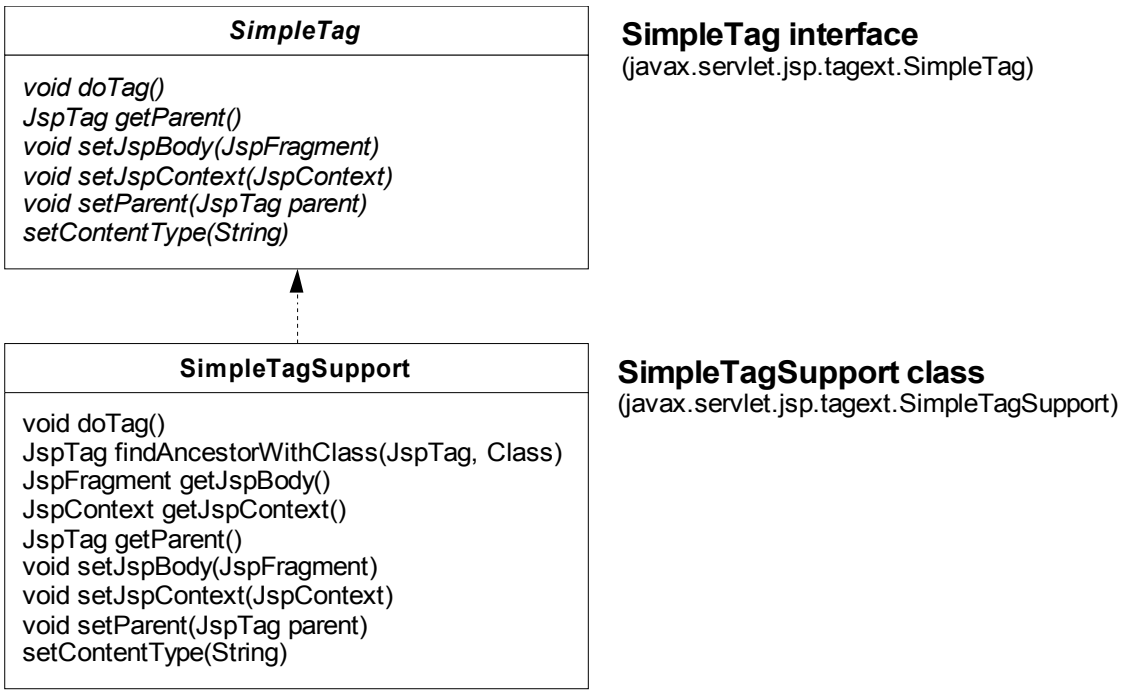

Figure 5-5: SimpleTag class diagram.

The “core” of the simple tag handler is the method `doTag()` which gets called when the tag’s end element is encountered. Please note, that the default implementation of the `doTag()` method of the `SimpleTagSupport` class does not do anything. Therefore, this method needs to be overwritten.

The simple tag handler class has access to the JSP context object `javax.serlvlet.jsp.JspContext` that “allows it to communicate with the JSP page” [ArBa04]. As the `PageContext` class extends the `JspContext` class, those classes provide access to all implicit objects (see chapter 5.4.4, p. 121) that are accessible from within a JSP page (such as `request`, `session`, and `application`) [ArBa04].

Listing 5-14 shows the simple tag handler `DateTag.java` that encapsulates the functionality to display a formatted date.

```
import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class DateTag extends SimpleTagSupport {

  private String format = "MMMM d, yyyy";

  public void setFormat(String format) {
    this.format = format;
  }

  public void doTag() throws JspException, IOException {
    DateFormat formatter = new SimpleDateFormat(format);
    getJspContext().getOut().print(formatter.format(new Date()));
  }
}
```

Listing 5-14: Simple tag handler "DateTag.java".

If a custom tag needs attributes the developer has to provide “a Bean-style setter method in the tag handler class for each attribute. If the tag invocation includes attributes, the [JSP] container invokes [the according] setter method for each attribute” [cf. BaSi04, 511].

As the custom tag `date` can include one attribute called `format`, the tag handler declares the private member variable format and the corresponding `setFormat()` method. In the event that no attribute is included, the string `format` is initialized with a default pattern.

The method `doTag()` contains the tag’s functionality. In order to format the current date, an object of type `DateFormat` is instantiated, passing the string format as argument to the constructor. Next, a `JspContext` reference is obtained by calling `getJspContext()`. The `JspContext` object provides access to the implicit object `JspWriter` by calling `getOut()`. Finally, the formatted date is printed to the servlet’s (i.e. the translated JSP) output-stream.

5.5.4.2 Tag Library Descriptor (TLD)

In order to be able to implement the custom tag, the tag handler must be declared in a tag library descriptor (TLD). As pointed out in chapter 5.5.2 (p. 130), a TLD is an XML document that “includes documentation on the tag library as a

whole and on its individual tags” [cf. PeRo03, 1-160]. The custom tag's TLD “is used by the JSP container to interpret pages that include `taglib` directives referring to that tag library [cf. PeRo03, 1-160].

The TLD-files names must have the extension `.tld`. If a tag library is deployed inside a JAR-file, the TLD must be in the `META-INF` directory or a subdirectory of it. The tag library can also be directly deployed to the Web application. Please see chapter 5.5.2.3 (p. 132) for details [ArBa04].

A TLD-file must begin with the root element `taglib` that specifies the XML Schema and the required JSP version:

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
  jsptaglibrary_2_0.xsd"
  version="2.0">
```

Table 5-11 lists the subelements[61] of the `taglib` element [ArBa04].

Element	Description
`tlib-version`	Indicates the tag library's version.
`uri`	This is the URI that “uniquely identifies the tag library” [ArBa04]. It is used by the `taglib` directive’s attribute `uri` to refer to a specific tag library.
`tag`	For each custom tag in the library there must be a `tag` element (which contains other subelements) declaring the custom tag.

Table 5-11: Subelements of the taglib element.

Each custom tag must be declared in the TLD with a `tag` element. The tag element provides the tag name, the class of its tag handler and information on the tag's attributes.

[61] Only those elements needed to implement a simple custom tag will be discussed. For all possible subelements please refer to the JSP Specification 2.0 available at [PeRo03].

Table 5-12 lists the tag element's subelements[62] [ArBa04].

Element	Description
`description`	Indicates "a description of the tag" [ArBa04] (optional).
`name`	The unique tag name that is used to reference the tag from within the JSP page (e.g. `<simple:date>`).
`tag-class`	Indicates "the fully qualified name of the tag handler class" [ArBa04].
`body-content`	Used "to specify the type of body that is valid for a tag" [ArBa04]. There are three possible values: `tagdependent`, `empty`, and `scriptless`.
`attribute`	For each attribute that a custom tag can/must have, there must be an `attribute` element (which contains other subelements) declaring the attribute.

Table 5-12: The tag element's subelements.

Each attribute that a custom tag may or must include has to be declared with the `attribute` element (that is nested in the `tag` element). For every attribute, the custom tag developer must specify the attribute's name, whether the attribute is required and whether the value can be determined by a runtime value.

Table 5-13 lists the `attribute` element's subelements [ArBa04], [cf. BaSi04, 470].

[62] Only those elements needed to implement a simple custom tag will be discussed. For all possible subelements please refer to the JSP Specification 2.0 available at [PeRo03].

Element	Description
`description`	Indicates “a description of the attribute” [ArBa04] (optional).
`name`	Indicates the attribute’s unique name.
`required`	Indicates “whether the attribute is required” [ArBa04] (optional, the default value is `false`).
`rtexprvalue`[63]	This element is optional. It defines whether the attribute's value is evaluated at translation or runtime. The default value is `false`. If the value is false, then only a string literal as that attribute’s value is possible. If it is true, EL expressions and scripting expressions can be used for the attribute’s value.

Table 5-13: The attribute element's subelements.

Listing 5-15 shows the TLD-file `simple.tld` that describes the tag library `simple` and the custom tag `date`.

[63] The abbriviation rtexprvalue stands for RunTime EXPRession value [cf. BaSi04, 467].

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-
    jsptaglibrary_2_0.xsd"
  version="2.0">

  <tlib-version>1.2</tlib-version>
  <uri>simpleTags</uri>
  <tag>
    <description>demonstration how to use Simple Tags</description>
    <name>date</name>
    <tag-class>cc.heinisch.thesis.tags.DateTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <description>
        The value has to comply with the Java date and time
        pattern strings.
      </description>
      <name>format</name>
      <required>false</required>
      <rtexprvalue>false</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Listing 5-15: TLD "simple.tld".

The TLD-file `simple.tld` describes one tag: the tag `date` must not have a body and can include the attribute `format` whose value must not be a runtime expression.

5.5.4.3 Using the Custom Tag

After deploying the tag library (i.e. the tag handler `DateTag.class` and the TLD `simple.tld`) to a Web application (see chapter 5.5.2.3, p. 132) the custom tag can be used in any JSP.

Listing 5-16 shows a JSP that uses the custom tag date to display the current date.

```
<%@ taglib prefix="simple" uri="simpleTags" %>
<html>
  <head>
    <title>Current Date</title>
  </head>
  <body>
    Today's date: <simple:date format="d/MM/yyyy" />
  </body>
</html>
```

Listing 5-16: "simpleTag.jsp" using the custom tag date.

The JSP declares a `taglib` directive. As the `uri` attribute's value exactly matches the `uri` element's value in `simple.tld`, the prefix simple will reference tags from this TLD. As the tag does not include any attribute, it will display the current date formatted with the default pattern as shown in figure 5-6.

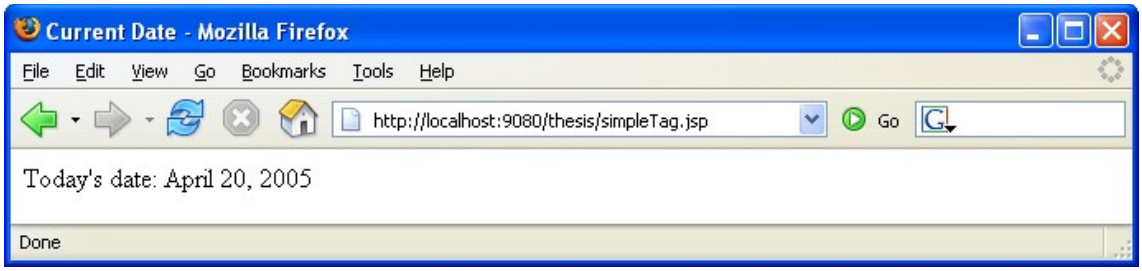

Figure 5-6: "simpleTag.jsp" using a custom tag (default pattern).

If the custom tag in `simpleTag.jsp` includes an attribute as

```
Today's date: <simple:date format="d/MM/yyyy" />
```

the date is formatted according to the defined pattern as shown in figure 5-7.

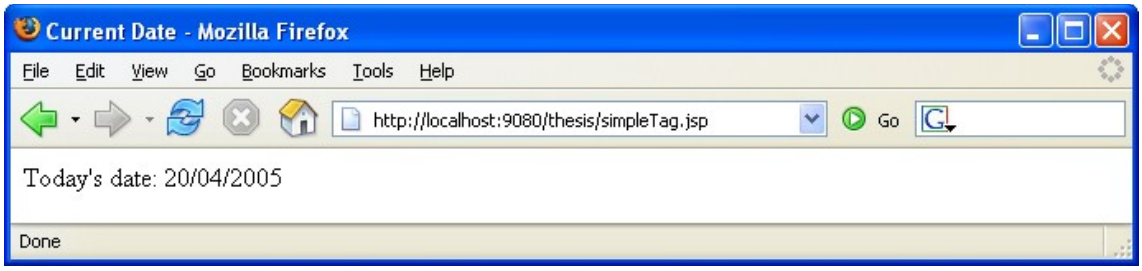

Figure 5-7: "simpleTag.jsp" using a custom tag with an attribute.

It is evident that the custom tag date provides a convenient way to display the current date in any format without the need to use scripting elements.

6 Roundup and Outlook

The main goal of the thesis was to provide an introduction to XML, Servlets and JavaServer Pages™. By now, the reader should be familiar with the basics of these technologies.

In the first part, the mother tongue of all markup languages, SGML, was introduced. Before the discussion of XML began, the benefits as well as the drawbacks of HTML were explained to highlight the need for an extensible markup language. The discussion about XML progressed to a look at its design and development. As XHTML is intended to be HTML's successor, it was only discussed briefly in the context of possible XML applications. By the means of a sample XML file, the structure of XML documents and the basic rules of how to create well-formed XML documents were presented. In order to familiarize the reader with the concept of document type definitions and valid XML documents, the discussion was led on to DTDs and XML schema. With the help of simple examples, it was demonstrated how to validate XML documents with these two technologies. As XML itself does not carry information of how to display its content, the discussion of XML concluded with the demonstration of how to use Cascading Style Sheets (CSS) and the Extensible Stylesheet Language (XSL) to format and display XML documents in a Web browser.

The second part intended to introduce the reader to the concepts of the Hypertext Transfer Protocol (HTTP). It was shown that resources on the Internet are generally addressed by the use of Uniform Resource Identifiers (URI). As all examples used in this thesis are accessed by HTTP, the Uniform Resource Locator (URL) was discussed in more detail. It was shown that the HTTP protocol is based on a request/response paradigm where a client sends an HTTP message to an HTTP server which in return sends back an HTTP message according to the request the client made. The reader was presented with both the HTTP request's format and the HTTP response's format, accompanied by HTTP request and HTTP response examples.

The third section concentrated on the discussion about servlets which are server-side software components written in Java. Subsequently, the servlet's appli-

cation programming interface (API) was presented. Then, the servlet's basic structure was explained to the reader before its life-cycle was elaborated in detail. The provided theory was clarified by the use of two examples: the first example demonstrated how to dynamically build a Web page that displays the current date in a Web browser. The second example servlet showed how to manipulate a picture dynamically and send the modified picture as binary data to the client that made the request. In order to understand how a servlet has to be deployed to a servlet container, the reader was introduced to the concept of Java Web applications. Finally, as servlets are similar to the Common Gateway Interface (CGI), but all together more efficient, the chapter closes with a comparison of these two technologies.

The last part was dedicated to the discussion of JavaServer Pages™ (JSP). Technically, a JSP is translated into a servlet by the JSP container. This process of transformation, as well as the phase of servicing requests, was explained comprehensively. To further enhance understanding of a JSP life-cycle, the automatically generated servlet Java-file resulting from an example JSP was presented and explained. Then, the reader was given an overview of the elements a JSP may include. Due to the limits of this thesis, not every element could be discussed in detail. As the Bean Scripting Framework provides the possibility to include non-Java scripts within a JSP, an excursus was provided on how to execute scripts written in Javascript and in ooRexx within a JSP. The last section of this chapter introduced the idea of script-free pages by the means of custom tags. Therefore, the JSP expression language was presented as an entry point to this topic before the discussion continued of how to include tag libraries in a Web application and how to reference and use custom tags in a JSP. Subsequently, a standardized set of custom tags, the JSP Standard Tag Library (JSTL), was introduced briefly. At the end, an example was presented of how to develop ones own custom tags, for the case that the JSTL or any other tag library did not provide for a specific functionality needed in a JSP.

In the appendix, the Web application MusicStore developed by the thesis' author was presented. The MusicStore is a simple Web shop that was developed to demonstrate how XML, Servlets and JavaServer Pages can be used together for building powerful Web applications. As the MusicStore application is based

upon the model-view-controller (MVC) pattern, the reader was given a basic introduction on the MVC pattern that is needed to understand Struts. Struts is an implementation of the MVC pattern that was chosen as the underlying application-framework for the MusicStore application. After a short explanation of Struts, the components of the MusicStore architecture were discussed. As not every file that is part of the application can be listed in the appendix due to lack of space, the MusicStore application can be downloaded from the author's Web site at [Hein06]. Furthermore, the reader was given detailed instruction of how to install the MusicStore Web application for both the IBM Websphere Application Server as well as the Apache Tomcat Servlet/JSP Container.

As a piece of work within the boundaries of a thesis, the subjects of XML, Servlets and JavaServer Pages™ have merely been touched upon. However, the numerous fields of discussion on these subjects far exceed the scope of this thesis. Nevertheless, as the reader should now have a fundamental knowledge of these subjects, this thesis may serve as an overture to further research into this particularly extensive area.

7 Appendix: Music Store

The primary goal of this thesis was to introduce the reader to XML, Servlets and to JavaServer Pages™.

In order to demonstrate how to build powerful Web applications by combining XML, Servlets and JavaServer Pages, a simple shopping cart Web application (titled *MusicStore*) that uses these three technologies extensively was developed by the thesis' author.

The MusicStore application is a simple Web shop where music instruments can be bought. The user can browse through the offered products, add them to the shopping cart and order the products. For simplicity's sake, the user does not have to create an account. To place an order, the user simply provides personal data via an HTML-form.

Before moving on to the discussion of the Web application's architecture, directory structure and installation, the next sections provide the theory needed to understand the Web application's underlying architecture.

7.1 Model-View-Controller (MVC) Pattern

The JSP specification provides two approaches for building Web applications using JSP pages: JSP Model 1 and Model 2 architectures. Model 2 (which is today most commonly referred to as "Model-View-Controller") provides a "clear separation of application responsibilities" [cf. Holm04, 6], as figure 7-1 illustrates.

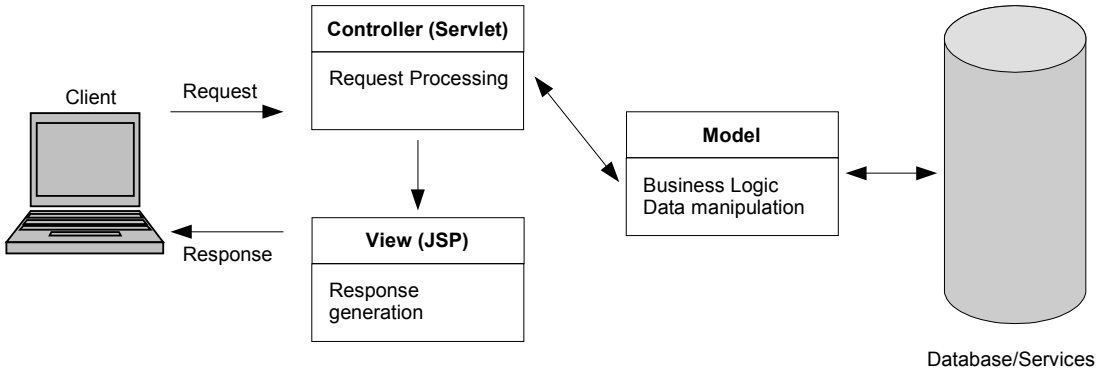

Figure 7-1: Model-View-Controller Architecture.

A central servlet (which is usually known as the controller) “receives the requests for the application” [cf. Holm04, 6]. It processes the requests and works with the model to prepare any data requested by the client. The controller forwards the data to the view (which is usually a JSP). Then the JSP uses the data to generate a response to the client.

As figure 7-1 illustrates, the MVC architecture breaks a Web application into three basic tiers [cf. Holm04, 7]:

- **Model Components**

  “Model components provide an interface to the data” [cf. Holm04, 7] (or services) needed by an application. They provide “the business logic to the application” [cf. Holm04, 7]. Consequently, there is no need for the controller components to “embed code for manipulating an application’s data” [cf. Holm04, 7]. The controller components “communicate with the model components that perform the data access and manipulation” [cf. Holm04, 7].

- **View Components**

  “View components are used to generate the response” [cf. Holm04, 7] to the client. In other words, a view component provides what the user sees. In the case of the MusicStore application, the view components are comprised of JSPs. However, any other view technology (such as WML or a Swing application) could be used “without impacting the model” [cf. Holm04, 7] tier of the application.

- **Controller Components**

  Controller components are “the core of the MVC architecture” [cf. Holm04, 7]. The controller usually is “a servlet that receives requests for the application and manages the flow of data between the model [...] and the view” [cf. Holm04, 7].

The MVC pattern is a powerful architecture for building Web applications. “The code for each screen in the application consists of a model and a view. Neither

of these components has explicit knowledge of the other's existence" as "they are decoupled via the controller" [cf. CaHa04, 42]. "This clean decoupling of the business and presentation logic" [cf. CaHa04, 42] facilitates Web development enormously since new functionality can be added to the application "by writing a model [component] and a view [component] and then registering these items to the controller of the application" [cf. CaHa04, 42].

As pointed out, a MVC-based architecture offers a very "flexible mechanism for building Web applications" [ShMa03]. However, developing the MVC infrastructure framework for ones Web application requires a lot of time and effort. Fortunately, today various implementations of a MVC-framework exist, that a developer can choose from. For the underlying Web application, the Struts development framework [Strut05] was chosen. Struts "is a full-blown implementation of the MVC pattern" [cf. CaHa04, 43] as the underlying framework for the Music-Store Web application.

7.2 Struts

The Struts development framework was initially designed "by Craig R. McClanahan and then donated to the Jakarta project of the Apache Software Foundation (ASF) in 2000" [cf. Holm04, 8]. With Craig's donation, Struts became open source software. Since then, many developers have contributed to the project and "Struts has flourished" [cf. Holm04, 8]. By now, "Struts has become the de facto standard for building Web applications in Java and has been embraced throughout the Java community" [cf. Holm04, 8].

7.2.1 Basic Components of Struts

"The Struts framework is a rich collection of Java libraries and can be broken down into the following major pieces" [cf. Holm04, 9]:

- **Base framework**

  "The base framework provides the core MVC functionality" [cf. Holm04, 9]. The controller servlet is at the core of this framework: `ActionServlet`. "The rest of the base framework is comprised of base classes [that the de-

veloper will extend] and several utility classes" [cf. Holm04, 9]. The most important base classes are the `Action` and the `ActionForm` classes. Action classes are used by the `ActionServlet` to process incoming requests. `ActionForm` classes "are used to capture data from HTML forms and to be a conduit of data" [cf. Holm04, 9] that is sent back to the view tier for page generation.

- **JSP tag libraries**

  Struts provides "several JSP tag libraries[64] for assisting with programming the view logic in JSPs" [cf. Holm04, 9].

- **Tiles plugin**

  "Tiles is a rich JSP templating framework that facilitates the reuse of presentation [i.e. HTML] code".

- **Validator plugin**

  The "Validator plugin provides a rich framework for performing data validation on both the server side and client side [...]. Each validation is configured in an [...] XML file so that validations can easily be added to and removed from the Web application declaratively" [cf. Holm04, 9].

7.2.2 Struts Workflow

Above the basics of the MVC pattern on which the Struts framework is based were discussed. Now the workflow that occurs when a user makes a request to a Struts-based Web application will be explained. Figure 7-2 illustrates that workflow [cf. CaHa04, 44].

[64] For a listing of these tag libraries please refer to [Stru05].

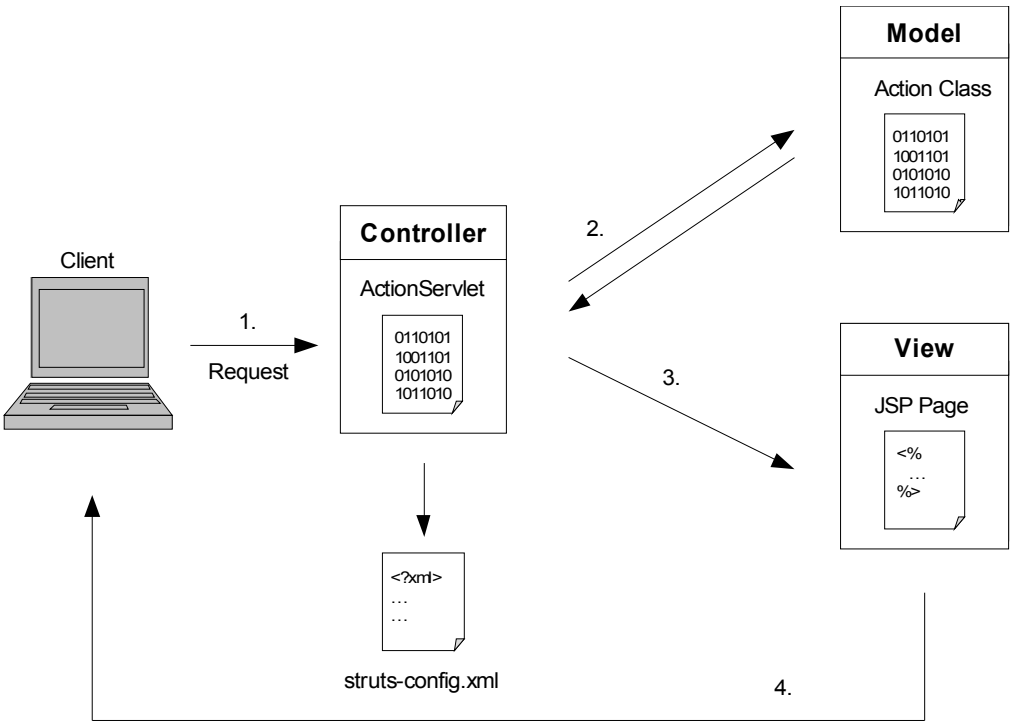

Figure 7-2: Struts workflow.

A Web page can usually contain a “variety of actions that the user may ask the [Web] application to undertake” [cf. CaHa04, 45]. Such actions may include clicking a hyperlink or an image that takes the user to another page, or submitting an HTML-form “that is to be processed by the application. All actions that are to be processed by the Struts framework will have a unique [...] file extension” [cf. CaHa04, 45] which is `*.do` (see the MusicStore’s deployment descriptor in listing 7-2, p. 172). This file extension “is used by the servlet container to map all the requests over to the Struts `ActionServlet`” [cf. CaHa04, 45].

The Struts `ActionServlet` “will take the incoming user request [see step one in figure 7-2] and map it to an action mapping defined in the `struts-config.xml` file” [cf. CaHa04, 45]. The `struts-config.xml` file (see listing 7-3, p. 174) contains the entire configuration “needed by the Struts framework to process a user’s request” [cf. CaHa04, 45]. Action mappings are declared with the `<action>` tag that provides the `ActionServlet` with the following information:

- "The `Action` class that is going to undertake the [...] user's request" [cf. CaHa04, 45]. An `Action` class is a Struts class that the application developer extends. The `Action` class's primary task is "to contain all of the logic that is necessary to process an [...] user's request" [cf. CaHa04, 45] (see listing 7-4, p. 175).

- An `ActionForm` class contains "any form data that is submitted by the [...] user" [cf. CaHa04, 45]. As with the `Action` class, it is also extended by the developer. Please note "that not every action in a Struts application requires an `ActionForm` class" [cf. CaHa04, 45]. An `ActionForm` class will have getter- and setter-methods "to retrieve each of the pieces of the form data" [cf. CaHa04, 45] (see listing 7-5, p. 176).

- "Where the users have to be forwarded to after their request has been processed by the `Action` class" [cf. CaHa04, 45]. Since there "can be multiple outcomes from an [...] user's request [...] an action mapping can contain multiple forward paths" [cf. CaHa04, 45]. A forward path is indicated by the `<forward>` tag and "is used by the Struts `ActionServlet` to direct the user to a specific JSP page or to another action mapping in the `struts-config.xml` file" [cf. CaHa04, 45] (see listing 7-3, p. 174).

As soon as the controller has fetched the information from the corresponding "`<action>` element for the request, it will process the [...] user's request" [cf. CaHa04, 45]. Therefore, "the `ActionServlet` will forward the user's request to the `Action` class defined by the action mapping" [cf. CaHa04, 46] (see step 2 in figure 7-2). The Action class provides three public methods and several protected methods. For the purpose of the MusicStore application, only the the `execute()` method of the `Action` class will be discussed. This method has to be overridden by the application developer. It "contains the entire business logic necessary to carry out the [...] user's request" [cf. CaHa04, 46] (see listing 7-4, p. 175).

When the `Action` has completed processing the request, "it will indicate to the `ActionServlet` where the user is to be forwarded" [cf. CaHa04, 46]. Usually, the user will be forwarded to a JSP that will display the request's result (see

step 3 in figure 7-2). “The JSP will render the data returned from the model as an HTML page that is displayed to the [...] user” [cf. CaHa04, 46] (see step 4 in figure 7-2).

To conclude the discussion on Struts, a typical Web screen that is based on the Struts development framework consists of the following:

- “An action that represents the code that will be executed when the user’s request will be processed. Each action in the Web page will map to exactly one `<action>` element that is defined in the `struts-config.xml` file” [cf. CaHa04, 46] (see listing 7-3, p. 174). Such an action that is requested by a user is embedded in a JSP as a hyperlink or “as an action `attribute` inside a `<form>` tag” [cf. CaHa04, 46].

- An `<action>` element that defines which `Action` class “will be used to process the [...] user’s request” [cf. CaHa04, 46].

- A JSP that will be used to “render a response to the user’s request” [cf. CaHa04, 46]. Inside the `<action>` element there is a `<forward>` element defined that is used to tell the `ActionServlet` which JSP will be used to render the response (see listing 7-3, p. 174).

7.3 Architecture of the MusicStore Web Application

This section illustrates and explains the various layers, patterns and technologies which together form the architecture of the MusicStore Web application.

Figure 7-3 illustrates the Web application’s architecture.

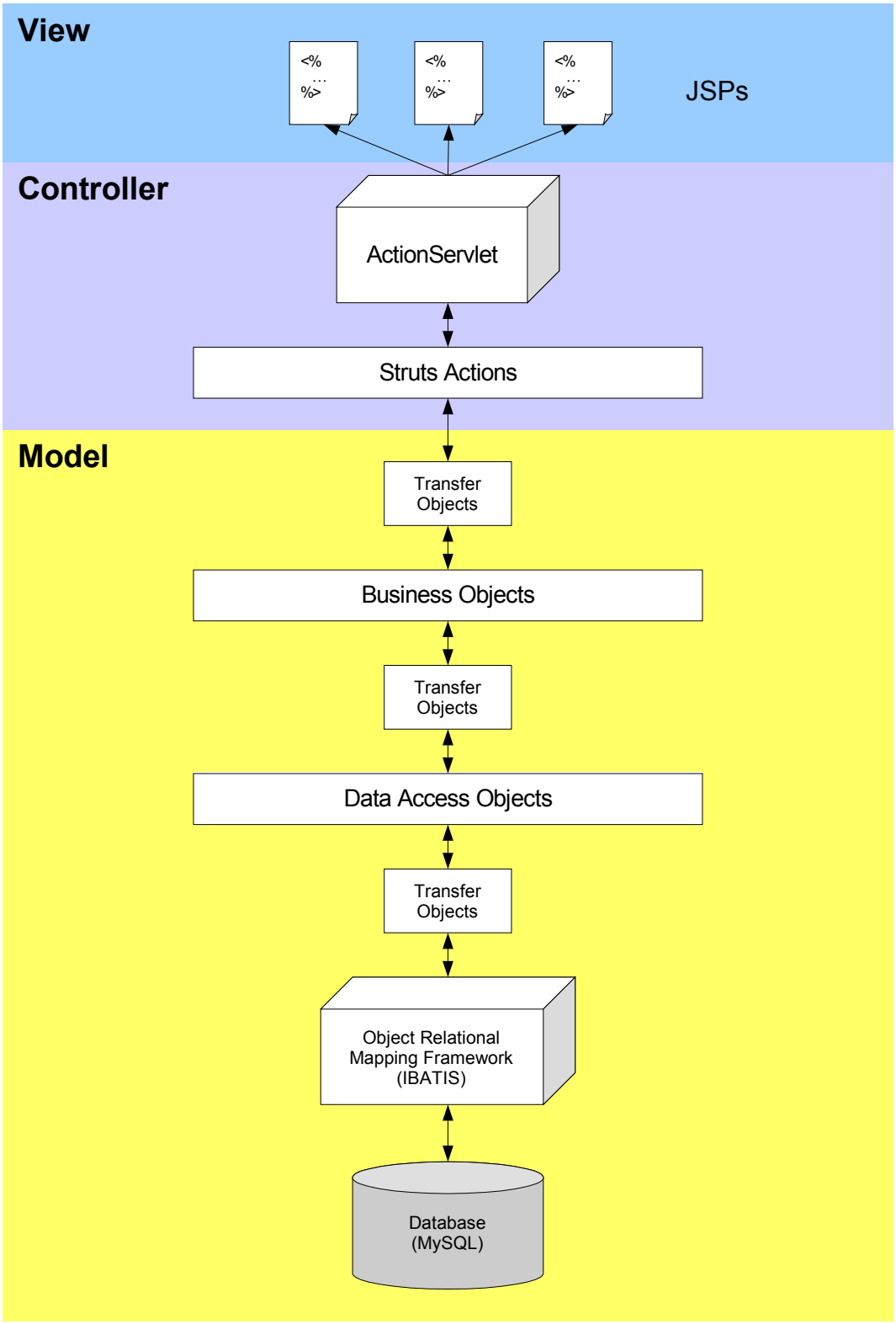

Figure 7-3: MusicStore architecture.

As mentioned above, the Struts `Action` class contains all of the logic that is necessary to process the user's request. Nevertheless, any concrete business logic should not be coded within the `Action` class (to enhance code re-use) [Stru06]. As a result, the challenge was to build the MusicStore application in

such a way that the business logic for the application becomes independent of the actual Struts framework. The `Action` classes of a Struts application shall only be a “plug-in point” for the business logic [cf. CaHa04, 161].

Therefore, the model-tier is divided into three layers:

- Business Logic Layer,
- Data Access Layer,
- Persistent Data Store Layer.

7.3.1 Business Logic Layer

The business logic layer was implemented with Business Objects. Business Objects (BO) “encapsulate and manage business data, behavior and persistence” [cf. AlCr03, 375]. For the MusicStore application, the Business Objects were implemented as Plain Old Java Objects (POJOs) which each represents a single domain model object.

In the MusicStore application there are following domains:

- Category: contains the logic to assemble the category tree to browse through products.
- Manufacturer: contains the logic to set up the manufacturer drop down list.
- Order: contains all the logic needed to process an order.
- Product: contains the logic to list and display products.
- Shoppingcart: contains the logic to add products to the shopping cart, delete products or update the shopping cart.

7.3.2 Data Access Layer

All interactions with the MusicStore database will be done through a set of Data Access Objects (DAOs). “DAO is a core J2EE design pattern that completely

abstracts the Create, Retrieve, Update and Delete (CRUD) logic" [cf. CaHa04, 216] that is needed to retrieve and manipulate the data.

The MusicStore database is a relational database which is row oriented and does not "map well into an object-oriented environment as Java" [cf. CaHa04, 217] is. Even with the use of DAOs, there is still the problem of passing "row-oriented Java objects, such as the `ResultSet` class, back and forth" [cf. CaHa04, 217] between the application's layers.

The solution to this is to use the Data Transfer Object (DTO) pattern to map the data that are "retrieved and sent to the relational database to a set of Java classes" [cf. CaHa04, 217]. DTOs "wrap the retrieved data behind simple `get()` and `set()` methods and minimize the exposure of the physical implementation details of the underlying database tables" [cf. CaHa04, 217]. Consequently, "the underlying database structure can be changed or even moved to an entirely different platform with a very small risk of breaking any applications consuming the data" [cf. CaHa04, 217].

The DAOs never communicate directly with the MusicStore database. Alternatively, all the database access is done through an Object Relational Mapping tool. The usage of such a tool "is significantly time saving" as it enables the developer "to define declaratively, rather than programmatically, how data is to be mapped to and from [the DTOs] in the application" [cf. CaHa04, 218]. In using an Object Relational Mapping tool it was not necessary to write JDBC code to retrieve the MusicStore data.

There is a huge offer on open source Object Relational Mapping tools. It was decided to use Ibatis SQL Maps framework [Ibat04] which seemed to much easier to use and integrate compared to other tools such as Hibernate [Hibe04] or ObjectRelationalBridge (OJB) [OJB04]. Additionally, Ibatis also offers a DAO Framework which neatly integrates its SQL Map software.

7.3.3 Persistent Data Store Layer

As the design of a data model for a shopping cart software is far beyond the scope of this thesis, the data model of the open source e-commerce solution osCommerce [Osco04], which uses MySQL as its database server, was used.

7.4 MusicStore Directory Structure

Figure 7-4 illustrates the directory layout of the MusicStore Web application.

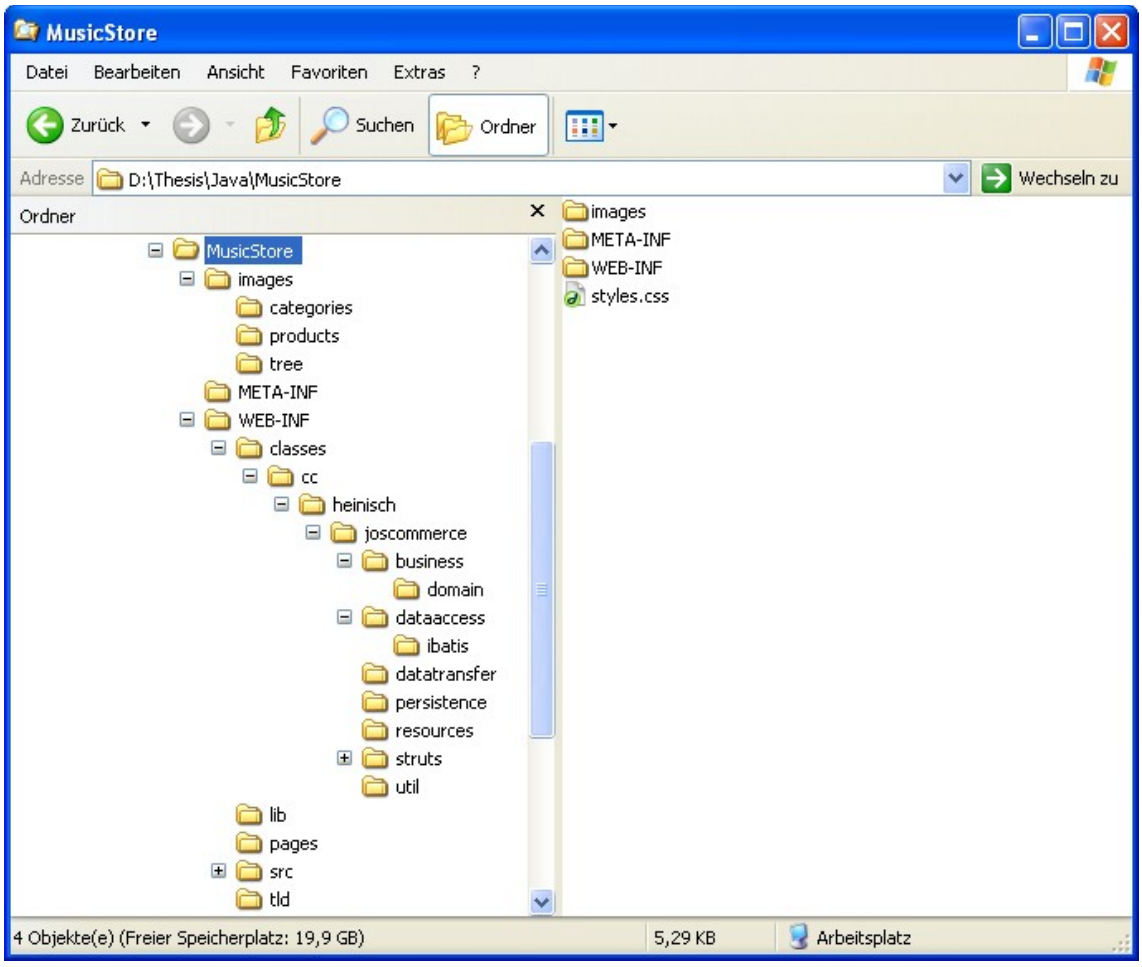

Figure 7-4: The MusicStore directory layout.

Please note that the `src`-directory's subdirectories was not expanded as its sub-directory-tree is exactly the same as the `classes`' subdirectory-tree.

Table 7-1 lists each directory's file(s). The files of the `src`-directory (and its sub-directories) were not listed as these files are the corresponding Java source files to the Java classes listed in `classes` and its subdirectories.

Directory	Files
MusicStore	styles.css
MusicStore/images	note.gif
MusicStore/images/categories	acustic_basses.jpg acustic_guitars.jpg basses.jpg drums.jpg electric_basses.jpg electric_guitars.jpg guitars.jpg keyboards.jpg piano.jpg synthesizer.jpg synthesizer_76.jpg synthesizer_88.jpg
MusicStore/images/products	AEB10.jpg bg29.jpg cg7_natural.jpg D-18.jpg deluxe_active_jazz_bassV.jpg fantom-x8.jpg FP-5.jpg les_paul_classic.jpg ludwig_maple.jpg motif8.jpg motifes7.jpg p250.jpg PA1XPRO.jpg pearl_forum.jpg s90.jpg sg_standard.jpg SP300.jpg sr500.jpg sr900.jpg srx750.jpg standard_jazz_bass.jpg standard_precision_bass.jpg standard_strat_hh.jpg standard_stratocaster.jpg standard_telecaster.jpg SWOMGT.jpg tama_superstar_custom.jpg triton_le.jpg x-plorer.jpg
MusicStore/images/tree	blankSpace.gif collapsedLastNode.gif collapsedMidNode.gif expandedLastNode.gif expandedMidNode.gif noChildrenLastNode.gif noChildrenMidNode.gif verticalLine.gif
MusicStore/META-INF	MANIFEST.MF
MusicStore/WEB-INF	struts-config.xml validation.xml validator-rules.xml web.xml

Directory	Files
MusicStore/WEB-INF/classes	log4j.properties
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/business/domain	CategoryBO.class ManufacturerBO.class OrderBO.class ProductBO.class ShoppingCartBO.class
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/dataaccess	dao.xml DaoConfig.class ICategoryDao.class IManufacturerDao.class IOrderDao.class IProductDao.class
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/dataaccess/ibatis	BaseDao.class CategoryDao.class ManufacturerDao.class OrderDao.class ProductDao.class
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/datatransfer	CategoryDTO.class CountryDTO.class CustomerDTO.class ManufacturerDTO.class OrderDTO.class OrderProductsDTO.class ProductDTO.class ShoppingCartDTO.class ShoppingCartItemDTO.class
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/persistence	CategorySQL.xml database.properties ManufacturerSQL.xml OrderSQL.xml ProductSQL.xml sql-map-config.xml
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/resources	ApplicationResources.properties
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/struts/action	CategoriesAction.class ConfirmOrderAction.class ManufacturerAction.class OrderAction.class ProductAction.class ProductDetailAction.class ReviewOrderAction.class ShoppingCartAction.class ShoppingCartEditAction.class
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/struts/bean	CustomerForm.class ManufacturerForm.class ShoppingCartForm.class
MusicStore/WEB-INF/classes/cc/heinisch/joscommerce/util	FormatPrice.class ResizeImageServlet.class
MusicStore/WEB-INF/lib	commons-beanutils.jar commons-collections.jar commons-digester.jar commons-fileupload.jar commons-lang.jar commons-logging.jar

Directory	Files
	commons-validator.jar ibatis-common-2.jar ibatis-dao-2.jar ibatis-sqlmap-2.jar jakarta-oro.jar jenkov-prizetags-bin-v.2.1.5.jar jstl.jar log4j-1.2.8.jar mysql-connector-java-3.0.15-ga-bin.jar standard.jar struts.jar struts-el.jar
MusicStore/WEB-INF/pages	cart.jsp categories.jsp confirmation.jsp footer.jsp header.jsp index.jsp manufacturer.jsp order.jsp product.jsp products.jsp review.jsp sessionTimedOut.jsp shoppingCart.jsp
MusicStore/WEB-INF/tld	c.tld struts-bean.tld struts-html.tld struts-html-el.tld struts-logic.tld treetag.tld

Table 7-1: Files contained in the MusicStore application.

7.5 Installing the Web Application

The MusicStore Web application is designed to run in a Servlet/JSP Container and to use MySQL as the back-end database.

The MusicStore Web application was deployed to both the IBM Websphere Application Server 6.0 and to the Apache Jakarta Tomcat Servlet/JSP container. The Web application was tested on both servers.

In this section it will be explained how to install the MusicStore database and how to deploy the Web application to the IBM Websphere Application Server and to the Apache Tomcat Server. The explanation of how to install the database server, the IBM Websphere Application Server and the Apache Tomcat Server is outwith the scope of this paper. However, it will be explicated

where the software can be downloaded. For any details on the respective installation the reader is asked to refer to the software's documentation.

Additionally, you need to have a current JDK installed. The MusicStore Web application relies on JVM version 1.4 or higher, so please make sure that the JDK is compatible.

All files regarding the MusicStore Web application can be downloaded from the authors Web site at [Hein06].

7.5.1 Installing the MusicStore Database

The MusicStore application uses MySQL as the data store. For the MusicStore application the MySQL Database Server 4.1 was used. If you do not already have the MySQL database server, then you need to obtain the version applicable to your platform which is available at [Mysl04a].

The next two steps are to install the MusicStore database and to create a MySQL user. The necessary sql-scripts can be downloaded from [Hein06] packed in a zip file name `MusicStore.zip`.

The zip-file contains two sql-scripts:

- `musicStore.sql`,
- `user_setup.sql`.

Please unpack the zip-file and store the two sql-files to your local hard disk.

1. Installing the MusicStore database

To install the MusicStore database you have to run the `musicStore.sql` script by using the command prompt (alternatively, you can use the MySQL Administrator [Mysl04b]). The script will automatically create the database (called `oscommerce`) and all the tables as well as populating the tables with data[65]:

Log in as root by typing in

[65] This is the syntax to login if a password for the user root was set. If there is no password, you can login with the command: `mysql --user=root`

```
mysql --user=root -p
```

MySQL will prompt you for the password.

Next, type in the following command using the correct script path for your environment:

```
\. d:\Thesis\Java\MusicStore\musicStore.sql
```

The script will create the MusicStore database, the tables in it and the data for those tables.

2. Creating the MySQL user

To create a MySQL user, type

```
\. d:\Thesis\Java\MusicStore\user_setup.sql
```

This script will create the user “oscommerce” with the password “oscommerce” used by the MusicStore application to connect to MySQL.

To verify that the scripts succeeded, swap to the MusicStore database which is named “oscommerce” by using the following command:

```
use oscommerce;
```

Now you can verify that the tables were created using the following command:

```
show tables;
```

This should show the 46 tables from listing 7-1[66].

[66] Please note that not all tables are used by the MusicStore web application. In order to not break the data model the unused tables were not removed.

```
address_book
address_format
banners
banners_history
categories
categories_description
configuration
configuration_group
counter
counter_history
countries
currencies
customers
customers_basket
customers_basket_attributes
customers_info
geo_zones
languages
manufacturers
manufacturers_info
newsletters
orders
orders_products
orders_products_attributes
orders_products_download
orders_status
orders_status_history
orders_total
products
products_attributes
products_attributes_download
products_description
products_notifications
products_options
products_options_values
products_options_values_to_products_options
products_to_categories
reviews
reviews_description
sessions
specials
tax_class
tax_rates
whos_online
zones
zones_to_geo_zones
```

Listing 7-1: SQL tables in the MusicStore database.

To make sure that the MusicStore user account was created in MySQL, run the following commands:

```
use mysql;

select user from user where user = 'oscommerce';
```

If this command returns zero results, then the user account is missing and you should run the `user_setup.sql` script again.

7.5.2 Deploying MusicStore to IBM WAS

If you do not already have the IBM Webphere Application Server 6.0 installed, you can download a trial version at [IBM05b]. If you do not have an account at www.ibm.com you will need to register to be able to download the trial version.

Once the IBM WAS is running, you can deploy the MusicStore Web application to it. Therefore, download the file `MusicStore.ear` from [Hein06].

Please follow the next steps to deploy the `MusicStore.ear` file to the application server:

1. Log in to the Websphere Administrative Console. Please make sure that your browser accepts cookies. Otherwise the login will fail. The default link for the console is

   `http://localhost:9060/ibm/console/`

2. In the menu, click on *Applications* and then on *Install New Application*.

3. In the field *Specify path*, select the `MusicStore.ear` file you downloaded from your local hard disk. The field *Context root* field is left empty. Then click the button *Next* (see figure 7-5).

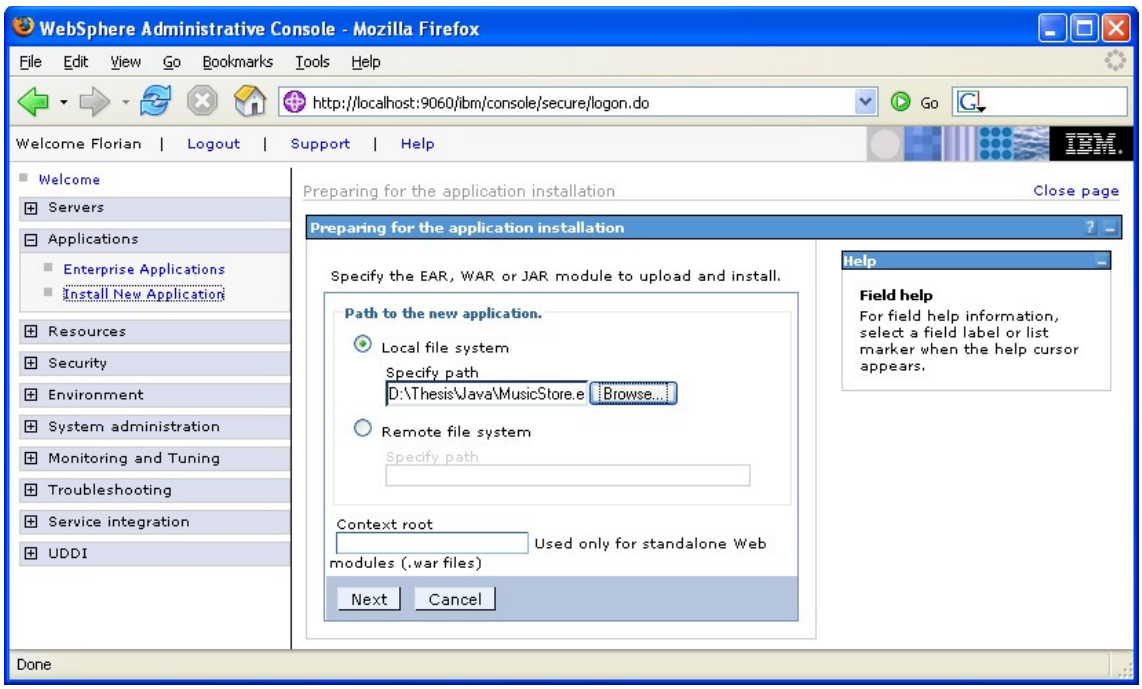

Figure 7-5: Deploying to IBM WAS: Step 3.

4. On this page you should just have to hit the button *Next* as by default the settings should be set as needed. Please verify that the checkbox *Generate Default Bindings* is not checked, so that in the form *Override* the radio box *Do not override existing bindings* is selected and that in the form *Virtual Host* none of the radio boxes is selected. The field *Specifiy binding file* is left empty (see figure 7-6).

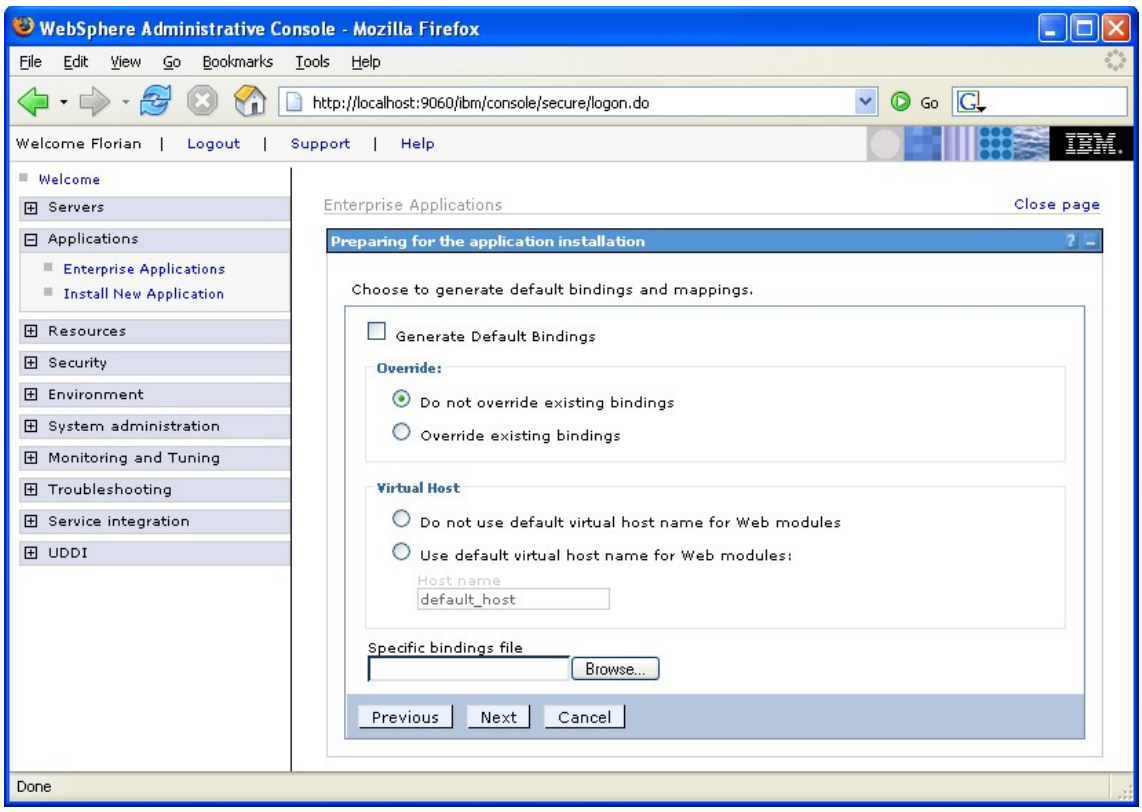

Figure 7-6: Deploying to IBM WAS: Step 4.

5. On this page you just have to change the entry in the field *Application name* from *MusicStoreEAR* to *MusicStore*. Then click next (see figure 7-7).

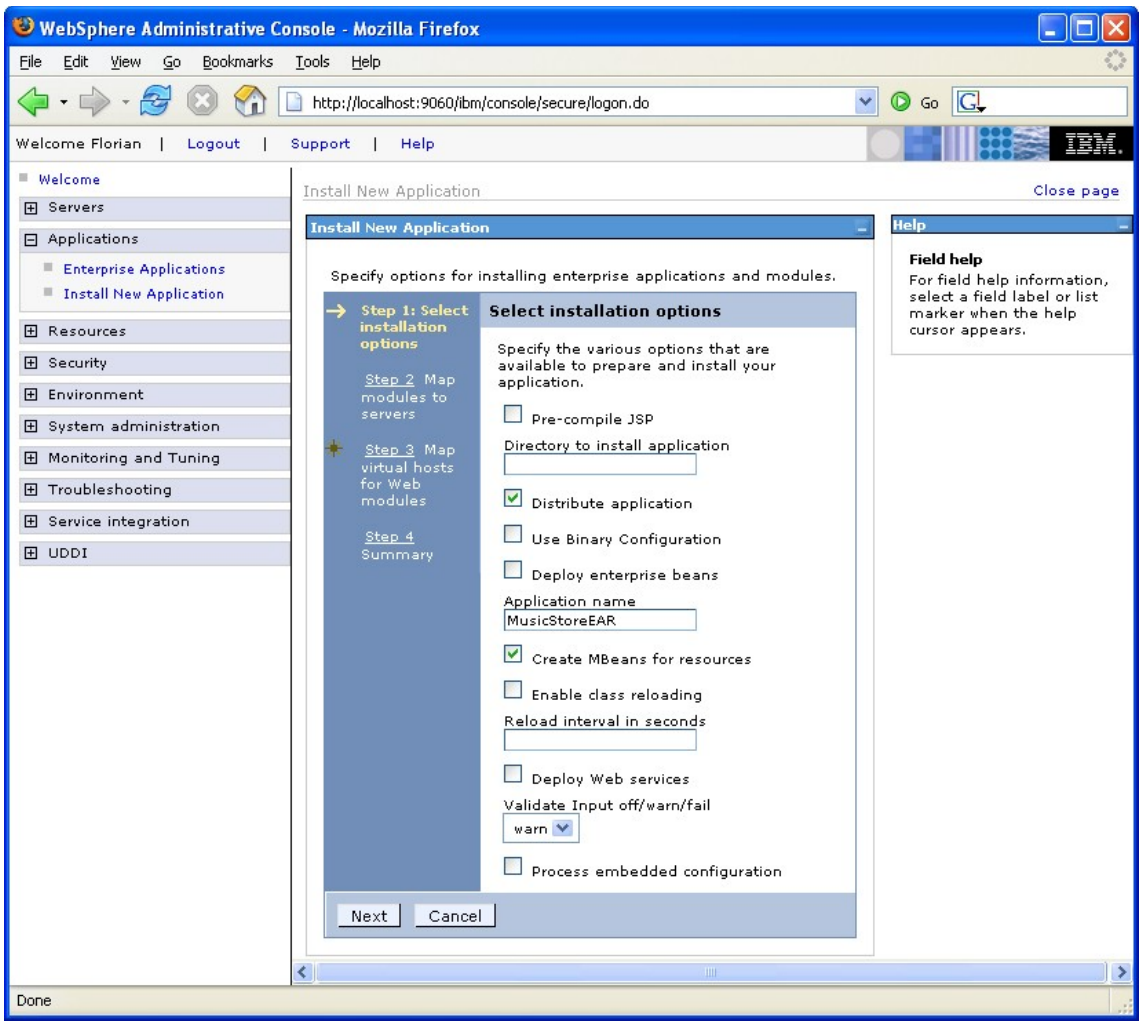

Figure 7-7: Deploying to IBM WAS: Step 5.

6. On the next page *Map modules* to servers, just click the button *Next*.

7. On the next page *Map virtual hosts for Web modules*, just click the button *Next*.

8. This page shows you the summary. Please click the button *Finish*. The Server now installs the Web application which should only take a few seconds. If the installation succeeded, the server displays *Application MusicStoreEAR installed successfully*. Next, click on the link *Save to Master Configuration*. The server will display a new page where you simply have to click the button *Save*.

9. Finally, you just have to start the application. To do so, click on the link *Enterprise Applications* in the menu on the left hand side of the page. The server will display the list of installed Web applications where you

should see the MusicStore application. Check the MusicStore's check-box and click the button *Start*.

Now, the MusicStore Web application can be invoked in the browser by the URL: `http://localhost:9080/MusicStore/`

which should display the a screen shown in figure 7-8.

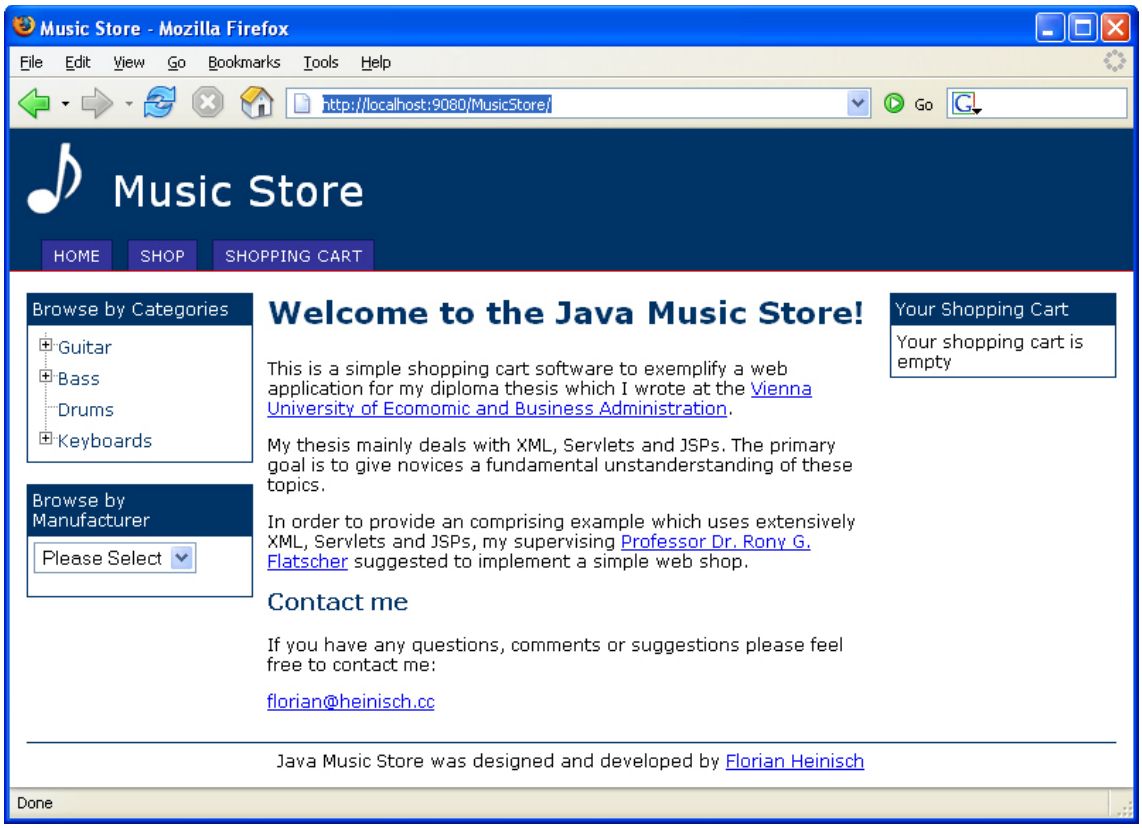

Figure 7-8: The MusicStore Web application displayed in a browser.

In order to browse through the MusicStore Web application your browser must accept cookies as the IBM Websphere Application Server does not enable URL rewriting by default.

7.5.3 Deploying MusicStore to Apache Tomcat

The MusicStore Web application was tested on the Apache Tomcat Servlet/JSP Container version 5.28. If you do not already have Tomcat installed, you can download the software at [Tomc04].

Once the Tomcat Container is installed on your machine, you can deploy the Web application to it. Compared to the deployment on the IBM Websphere Application Server, the deployment to the Tomcat Container is rather simple and straightforward.

Please download the file `MusicStore.war` from [Hein06].

Next, you just have to copy the file `MusicStore.war` into the Apache Tomcat's `webapps` directory which is located at

```
\your_local_path\tomcat_root\webapps
```

If the Tomcat Container has not been started already, start it now. The MusicStore Web application will be accessible by the URL

```
http://localhost:8080/MusicStore/
```

which should produce the same result as shown in figure 7-8 (p. 170).

7.6 Listings

Due to lack of space the content of each file contained in the MusicStore Web application will not be listed. Even if only the Java and JSP files (which are 51 files) were listed, this paper would become far too large. Therefore, only the files that are referenced from within the appendix are listed. To view all files, the reader is asked to download the entire MusicStore application from the author's Web site at [Hein06].

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>MusicStore</display-name>
  <!-- Standard Action Servlet Configuration (with debugging) -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
  </servlet>
  <!-- ImageResizeServlet Configuration -->
  <servlet>
    <servlet-name>ImageResizeServlet</servlet-name>
    <servlet-class>cc.heinisch.joscommerce.util.ResizeImageServlet</servlet-class>
  </servlet>
  <!-- Standard Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
  <!-- ImageResizeServlet Mapping -->
  <servlet-mapping>
    <servlet-name>ImageResizeServlet</servlet-name>
    <url-pattern>/resizeImage.do</url-pattern>
  </servlet-mapping>
  <!-- Session configuration -->
  <session-config>
    <session-timeout>20</session-timeout>
  </session-config>
  <!-- The Welcome File configuration -->
  <welcome-file-list>
    <welcome-file>/WEB-INF/pages/index.jsp</welcome-file>
  </welcome-file-list>
  <!-- Tag Library Descriptors -->
  <taglib>
    <taglib-uri>/tags/struts-bean</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-bean.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-html</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-html.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-logic</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-logic.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/struts-html-el</taglib-uri>
    <taglib-location>/WEB-INF/tld/struts-html-el.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/jenkov-treetag</taglib-uri>
    <taglib-location>/WEB-INF/tld/treetag.tld</taglib-location>
  </taglib>
  <taglib>
    <taglib-uri>/tags/jstl-c</taglib-uri>
    <taglib-location>/WEB-INF/tld/c.tld</taglib-location>
  </taglib>
</web-app>
```

Listing 7-2: "web.xml".

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>
  <!-- Form Bean Definitions -->
  <form-beans>
    <form-bean name="manufacturerForm"
               type="cc.heinisch.joscommerce.struts.bean.ManufacturerForm" />
    <form-bean name="customerForm"
               type="cc.heinisch.joscommerce.struts.bean.CustomerForm" />
    <form-bean name="shoppingCartForm"
               type="cc.heinisch.joscommerce.struts.bean.ShoppingCartForm" />
  </form-beans>
  <!-- Global Forward Definitions -->
  <global-forwards>
    <forward name="sessionTimedOut" path="/WEB-INF/pages/sessionTimedOut.jsp" />
  </global-forwards>
  <!-- Action Mapping Definitions -->
  <action-mappings>
    <action path="/index"
            type="org.apache.struts.actions.ForwardAction"
            name="manufacturerForm"
            validate="false"
            parameter="/WEB-INF/pages/index.jsp" />
    <action path="/categories"
            type="cc.heinisch.joscommerce.struts.action.CategoriesAction"
            name="manufacturerForm"
            validate="false"
            scope="request">
      <forward name="showCategories" path="/WEB-INF/pages/categories.jsp" />
    </action>
    <action path="/checkout"
            type="cc.heinisch.joscommerce.struts.action.OrderAction"
            name="customerForm"
            validate="false"
            scope="request">
      <forward name="success" path="/WEB-INF/pages/order.jsp" />
    </action>
    <action path="/confirmOrder"
            type="cc.heinisch.joscommerce.struts.action.ConfirmOrderAction">
      <forward name="success" path="/WEB-INF/pages/confirmation.jsp" />
    </action>
    <action path="/products"
            type="cc.heinisch.joscommerce.struts.action.ProductAction"
            name="manufacturerForm"
            validate="false"
            scope="request">
      <forward name="products" path="/WEB-INF/pages/products.jsp" />
    </action>
    <action path="/productDetail"
            type="cc.heinisch.joscommerce.struts.action.ProductDetailAction">
      <forward name="success" path="/WEB-INF/pages/product.jsp" />
    </action>
    <action path="/productsByManufacturer"
            type="cc.heinisch.joscommerce.struts.action.ManufacturerAction"
            name="manufacturerForm"
            validate="false"
            scope="request">
      <forward name="products" path="/WEB-INF/pages/products.jsp" />
    </action>
    <action path="/review"
            type="cc.heinisch.joscommerce.struts.action.ReviewOrderAction"
            name="customerForm"
            input="/checkout.do"
            validate="true"
            scope="request">
      <forward name="success" path="/WEB-INF/pages/review.jsp" />
    </action>
    <action path="/shoppingCart"
            type="cc.heinisch.joscommerce.struts.action.ShoppingCartAction"
            name="shoppingCartForm"
            validate="false" scope="request">
      <forward name="success" path="/WEB-INF/pages/shoppingCart.jsp"/>
    </action>
```

```
    <action path="/shoppingCartEdit"
            type="cc.heinisch.joscommerce.struts.action.ShoppingCartEditAction"
            name="shoppingCartForm"
            parameter="function"
            scope="request">
      <forward name="shoppingCart" path="/shoppingCart.do" redirect="true" />
    </action>
  </action-mappings>
  <!-- Message Resources Definitions -->
  <message-resources
    parameter="cc.heinisch.joscommerce.resources.ApplicationResources"/>
  <!-- Plug Ins Configuration -->
  <!-- Validator PlugiIn Configuration -->
  <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
                  value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
  </plug-in>
</struts-config>
```

Listing 7-3: "struts-config.xml".

```
package cc.heinisch.joscommerce.struts.action;

import java.util.List;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import cc.heinisch.joscommerce.business.domain.OrderBO;

public class OrderAction extends Action {

  public static Log log = LogFactory.getLog(OrderAction.class);

  public ActionForward execute(
      ActionMapping mapping,
      ActionForm form,
      HttpServletRequest request,
      HttpServletResponse response)
      throws Exception {

    ActionErrors errors = new ActionErrors();
    ActionForward forward = new ActionForward(); // return value
    HttpSession session = request.getSession();

    // Check whether Object intance initialized exists in session scope
    // If not, user is redirected to a specific error page
    if(session.getAttribute("initialized") == null) {
      return mapping.findForward("sessionTimedOut");
    }

    try {
      OrderBO orderBO = OrderBO.getInstance();
      List countryList = orderBO.getCountriesList();
      request.setAttribute("countries", countryList);
    }
    catch (Exception e) {
      // Report the error using the appropriate name and ID.
      errors.add("name", new ActionError("id"));
      log.debug("Some error occured: " + e);
    }

    // Finish with
    return (mapping.findForward("success"));
  }
}
```

Listing 7-4: "OrderAction.java".

```
package cc.heinisch.joscommerce.struts.bean;

import org.apache.struts.action.ActionForm;

public class ManufacturerForm extends ActionForm {

  private String manufacturerId;

  public String getManufacturerId() {
    return manufacturerId;
  }

  public void setManufacturerId(String string) {
    manufacturerId = string;
  }

}
```

Listing 7-5: "ManufacturerForm.java".

8 References

[AbPr98] Abrahamson, David; Price, Roger: User's Guide to ISO/IEC 15445:1998 HyperText Markup Language (HTML). http://xml.coverpages.org/cd15445UG.html, 1998, request on 2005-11-30.

[AdBe97] Adler, Sharon; Berglund, Anders et al.: A Proposal for XSL. http://www.w3.org/TR/NOTE-XSL.html, 1997-08-27, requested on 2006-03-10.

[AdBe01] Adler, Sharon; Berglund, Anders et al.: Extensible Stylesheet Language (XSL) Version 1.0. http://www.w3.org/TR/xsl/, 2001, requested on 2002-05-12.

[AlCr03] Alur, Deepak; Crupi, John et al.: Core J2EE Patterns – Best Practices and Design Strategies. Second Edition, Prentice Hall, Upper Saddle River 2004.

[Alle01] Allen, Mark: Document Type Definition (DTD) Introduction. http://ctdp.tripod.com/independent/web/dtd/, 2001-02-11, requested on 2001-10-02.

[Apac02] The Apache Jakarta Project: Application Developer's Guide. http://jakarta.apache.org/tomcat/tomcat-4.1-doc/appdev/deployment.html, 2002, requested on 2005-03-11.

[ArBa04] Armstrong, Eric; Ball, Jennifer et al.: The J2EE™ 1.4 Tutorial. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html, 2004, requested on 2005-03-10.

[AuPe06] Austin, Daniel; Peruvemba, Subramanian et al.: XHTML™ Modularization 1.1. http://www.w3.org/TR/2006/PR-xhtml-modularization-20060213/, 2006-02-13, requested on 2006-03-02.

[AvAy00] Avedal, Karl; Ayers, Danny et al.: Professional JSP: Using JavaServer Pages, Servlets, EJB, JNDI, JDBC, XML, XSLT, and WML. http://stardeveloper.com:8080/asp_bk_projsp_1.asp, 2000,

requested on 2002-07-06.

[BaSi04] Basham, Bryan; Sierra, Kathy et al.: Head First Servlets & JSP™ – Passing the Sun Certified Web Component Developer Exam. First Edition, O'Reilly, Sebastopol 2004.

[Bea04] Bea Systems: BEAWebLogic Server™: Assembling and Configuring Web Applications. http://e-docs.bea.com/wls/docs70/pdf/webapp.pdf, 2004, requested on 2005-03-11.

[Bea06] Bea Systems: Download of the Weblogic Server 9.1. http://commerce.bea.com/index.jsp, requested on 2006-03-22.

[Bean05] Sun Microsystems Web site on JavaBeans. http://java.sun.com/products/javabeans/, requested on 2005-05-06.

[BeFi96] Berners-Lee, Tim; Fielding Roy et al.: Hypertext Transfer Protocol -- HTTP/1.1. http://www.w3c.org/Protocols/HTTP/1.1/spec.html, 1996, requested on 2005-02-26.

[BeFi99] Berners-Lee, Tim; Fieldung Roy et al.: Hypertext Transfer Protocol -- HTTP/1.1. http://www.w3c.org/Protocols/rfc2616/rfc2616.html, 1999, requested on 2005-02-28.

[BeFi05] Berners-Lee, Tim; Fielding, Roy et al.: RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. http://www.ietf.org/rfc/rfc3986.txt, 2005-01, requested on 2005-02-27.

[BeMa94] Berners-Lee, Tim; Masinter, Larry et al.: RFC 1738: Uniform Resource Locators (URL). http://www.ietf.org/rfc/rfc1738.txt, 1994-12, requested on 2006-03-14.

[Berg02] Bergsten, Hans: JavaServer Pages, 2nd Edition. http://www.oreilly.com/catalog/jserverpages2/chapter/index.html, 2002, requested on 2005-03-21.

[Bern96] Berners-Lee, Tim: The World Wide Web: Past, Present and Future. http://www.w3c.org/People/Berners-Lee/1996/ppf.html, 1996-08, requested on 2001-09-13.

[BlBo02] Bloch, Cynthia; Bodoff, Stephanie: Overview of Servlets. http://java.sun.com/docs/books/tutorial/servlets/overview/architecture.html, requested on 2002-06-07.

[BiMa04] Biron, Paul V.; Malhotra, Ashok: XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/xmlschema-2/, 2004, requested on 2005-03-29.

[BoÇe05] Bos, Bert; Çelik, Tantek et al.: Cascading Style Sheets, level 2 revision 1 - CSS 2.1 Specification. http://www.w3.org/TR/CSS21/, 2005-06-13, requested on 2006-03-07.

[Bodo00b] Bodoff, Stephanie: Custom Tags in JSP Pages. http://java.sun.com/webservices/docs/ea1/tutorial/doc/JSPTags.html, 2000, requested on 2002-07-16.

[Bodo02a] Bodoff, Stephanie: JavaServer Pages Technology. http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPIntro.html, requested on 2002-07-06.

[Bodo02b] Bodoff, Stephanie: Java Servlet Technology. http://java.sun.com/webservices/docs/ea2/tutorial/doc/Servlets.html, requested on 2002-06-07.

[BoLi98] Bos, Bert; Lie, Håkon Wium et al.: Cascading Style Sheets, level 2 - CSS2 Specification. http://www.w3.org/TR/CSS2/, 1998-05-12, requested on 2006-03-07.

[Bos99] Bos, Bert: CSS & XSL. http://www.w3.org/Style/CSS-vs-XSL, 1999-07-22, requested on 2006-03-07.

[Bos00] Bos, Bert: How to add style to XML. http://www.w3.org/Style/styling-XML, 2000-02-29, requested on 2006-03-07.

[Bos01] Bos, Bert: XML in 10 points. http://www.w3.org/XML/1999/XML-in-10-points, 2001-07-02, requested on 2001-09-14.

[Bos06a] Bos, Bert: Web Style Sheets home page. http://www.w3.org/Style/, 2006-02-09, requested on 2006-03-07.

[Bos06b] Bos, Bert: Cascading Style Sheets home page. http://www.w3.org/Style/CSS/, 2006-02-09, requested on 2006-03-07.

[Bosa97] Bosak, Jon: XML, Java, and the future of the Web. http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.html, 1997-03-10, requested on 2001-09-13.

[Bour05] Bourret, Ronald: XML Namespaces FAQ. http://www.rpbourret.com/xml/NamespacesFAQ.htm, 2005, requested on 2006-03-04.

[Brad89] Braden, R.: RFC 1123: Requirements for Internet Hosts -- Application and Support. http://www.ietf.org/rfc/rfc1123.txt, 1989-10, requested on 2006-03-14.

[Bray98] Bray, Tim: The Annotated XML Specification. http://www.xml.com/axml/testaxml.htm, 1998-02-10, requested on 2006-02-22.

[BrDu00] Britt, James; Duynstee, Teun.: Doing XPath and XSLT with Style!. http://www.topxml.com/xsl/articles/xpath_xsl_style/, 2000, requested on 2002-05-12.

[BrHo99] Bray, Tim; Hollander, Dave et al.: Namespace in XML. http://www.w3.org/TR/REC-xml-names/, 1999, requested on 2005-03-28.

[Brow06] Brown, Mike J.: The skew.org XML Tutorial. http://skew.org/xml/tutorial/, 2006-01-21, requested on 2006-02-21.

[BrPa00] Bray, Tim; Paoli, Jean et al.: Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation. http://www.w3.org/TR/REC-xml, 2000-10-06, requested on 2001-09-13.

[BrPa04a] Bray, Tim; Paoli, Jean et al.: Extensible Markup Language (XML) 1.0 (Third Edition), W3C Recommandation. http://www.w3.org/TR/2004/REC-xml11-20040204/, 2004-02-04, requested on 2005-12-19.

[BrPa04b] Bray, Tim; Paoli, Jean et al.: Extensible Markup Language (XML) 1.1, W3C Recommendation. http://www.w3.org/TR/2004/REC-xml11-20040204/, 2004-04-15, requested on 2005-12-19.

[BSF05a] Bean Scripting Framework. http://jakarta.apache.org/bsf/, requested on 2006-03-20.

[BSF05b] BSF Tag Library. http://jakarta.apache.org/taglibs/doc/bsf-doc/intro.html, request on 2005-09-17.

[BSF406] Current version of BSF4Rexx. http://wi.wu-wien.ac.at/rgf/rexx/bsf4rexx/current/, requested on 2006-03-21.

[BuZu05] Bulterman, Dick; Zucker, Daniel: Synchronized Multimedia Integration Language (SMIL 2.1). http://www.w3.org/TR/2005/REC-SMIL2-20051213/, 2005-12-13, requested on 2006-03-14.

[Cagl00] Cagle, K.: Beginning XML. http://www.vbip.com/books/1861003412/pub.asp, 2000-06, requested on 2001-09-18.

[CaHa04] Carnell, John; Harrop, Rob: Pro Jakarta Struts. Second Edition, Apress, New York 2004.

[CaIo03] Carlisle, David; Ion, Patrick et al.: Mathematical Markup Language (MathML) Version 2.0 (Second Edition), http://www.w3.org/TR/MathML2/, 2003-10-21, requested on 2006-03-14.

[Cera01] Cerami, Ethan: Servlets: The Servelt Life Cycle. http://www.ecerami.com/applied_fall_2001/, 2001, requested on 2002-06-17.

[Chas03] Chase, Nicholas: Validating XML. http://www-106.ibm.com/developerworks/edu/x-dw-xvalid-i.html, 2003, requested on 2005-03-28.

[Clar97] Clark, James: Comparison of SGML and XML. http://www.w3.org/TR/NOTE-sgml-xml-971215, 1997-12-15, requested on 2006-02-27.

[Clar99b] Clark, James: XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, 1999-11-16, requested on 2006-03-12.

[Clar99a] Clark, James: Associating Style Sheets with XML documents Version 1.0. http://www.w3.org/TR/xml-stylesheet/, 1999-06-29, requested on 2006-03-07.

[Claß01a] Claßen, Michael: XML, the better HTML?. http://www.webreference.com/xml/column1/ 2001, requested on 2001-09-13.

[Claß01b] Claßen, Michael: XML, what for?. http://www.webreference.com/xml/column19/, 2001, requested on 2005-03-23.

[ClDe99] Clark, James; DeRose, Steve: XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 1999-11-16, requested on 2006-03-10.

[CoCo01] Coates, Tony; Connolly, Dan et al.: URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/uri-clarification/, 2001, requested on 2005-02-27.

[Conn99] Connolly, Dan: CGI: Common Gateway Interface. http://www.w3.org/CGI/, 1999, requested on 2006-03-16.

[Cost01] Costello, Roger L.: XML Schemas. http://www.xfront.com/xml-schema.html, 2001, requested on 2001-10-17.

[Cove01] Cover, Robin: XML Schemas. http://xml.coverpages.org/schemas.html, 2001-10-08, requested on 2001-10-16.

[Cowa01] Coward, Danny: Java™ Servlet Specification Version 2.3. http://java.sun.com/products/servlet/download.html, 2001, requested on 2001-06-18.

[CoYo03] Coward, Danny; Yoshida, Yutaka: Java™ Servlet Specification Version 2.4. http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html, 2003, requested on 2005-03-10.

[CSS02] Cascading Style Sheets. http://www.w3.org/Style/CSS/, requested on 2002-05-06.

[Culs97] Culshaw, Stuart: Solving the problem of publishing online documents. http://xml.coverpages.org/culshawSunserverXML.html, requested on 2005-12-18.

[Deli03] Delisle, Pierre: JavaServer Pages™ Standard Tag Library Version 1.1. http://jcp.org/aboutJava/communityprocess/final/jsr052/index2.html, 2003, requested on 2005-04-09.

[Day00] Day, Don: Hands-on XSL. http://www-106.ibm.com/developerworks/library/hands-on-xsl/, 2000, requested on 2002-05-10.

[DuCh01] DuCharme, Bob: XSLT Quickly. http://www.topxml.com/xsl/articles/xsltquickly/, 2001, requested on 2002-05-11.

[Duck01] Duckett, Jon.: Professional XML Schemas. http://www.topxml.com/schema/articles/xmlschemas/default.asp,

2001-07, requested on 2001-10-17.

[Extr01] Extropia: Introduction to XML for Web Developers. http://www.extropia.com/tutorials/xml/comments.html, requested on 2001-09-18.

[Fall01] Fallside, David C.: XML Schema Part 0: Primer, http://www.w3.org/TR/xmlschema-0/, 2001-05-02, requested on 2001-10-16.

[FaWa04] Fallside, David; Walmsley, Priscilla: XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/, 2004, requested on 2005-03-29.

[FeJa03] Ferraiolo, Jon; Jackson, Dean et al.: Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG11/, 2003-01-14, requested on 2006-03-14.

[FiGe99] Fielding, R.; Gettys, J. et al.: RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt, 1999-06, requested on 2006-03-15.

[Fisc01] Fischer, Stefan: Verteilte Systeme – Kapitel 5: Standard-Internetanwendungen. http://www.ibr.cs.tu-bs.de/lehre/ws0102/vs/VS-0102-Kap05-InternetAnwendungen-1S.pdf, 2001, requested on 2005-02-26.

[Flat03] Flatscher, Rony G.: The Augsburg Version of BSF4REXX. http://wi.wu-wien.ac.at/rgf/rexx/orx14/orx14_bsf4rexx-av.pdf, 2003, requested on 2005-09-13.

[Flat04] Flatscher, Rony G.: Camouflaging Java as Object Rexx. http://wi.wu-wien.ac.at/rgf/rexx/orx15/2004_orx15_bsf-orx-layer.pdf, 2004, requested on 2005-09-15.

[Flat06] The Website for BSF4Rexx. http://wi.wu-wien.ac.at/rgf/rexx/bsf4rexx/, requested on 2006-03-20.

[Flyn02] Flynn, Peter: The XML FAQ (v. 2.1). http://xml.coverpages.org/FAQv21-200201.html, 2002-01-01, requested on 2005-12-19.

[Flyn06] Flynn, Peter: The XML FAQ (v. 4.51). http://xml.silmaril.ie/, 2006-02-28, requested on 2006-02-28.

[FrBo96a] Freed, N.; Borenstein, N.: RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. http://www.ietf.org/rfc/rfc2045.txt, 1996-10, requested on 2006-03-15.

[FrBo96b] Freed, N.; Borenstein, N.: RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. http://www.ietf.org/rfc/rfc2046.txt, 1996-10, requested on 2006-03-15.

[FrBo96c] Freed, N.; Borenstein, N.: RFC 2047: Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text. http://www.ietf.org/rfc/rfc2047.txt, 1996-10, requested on 2006-03-15.

[FrBo96d] Freed, N.; Borenstein, N.: RFC 2048: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. http://www.ietf.org/rfc/rfc2048.txt, 1996-10, requested on 2006-03-15.

[FrBo96e] Freed, N.; Borenstein, N.: RFC 2049: Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. http://www.ietf.org/rfc/rfc2049.txt, 1996-10, requested on 2006-03-15.

[Fret98a] Freter, Todd: XML: Mastering Information on the Web. http://www.sun.com/980310/xml/ 1998-03-10, requested on 2001-09-13.

[Fret98b] Freter, Todd: XML: Document and Information Management. http://www.sun.com/980908/xml/, 1998-09-08, requested on 2001-

09-17.

[Gabh03a] Gabhart, Kyle: J2EE pathfinder: Create and manage stateful Web apps. http://www-106.ibm.com/developerworks/java/library/j-pj2ee6.html, 2003, requested on 2005-03-10.

[Gabh03b] Gabhart, Kyle: J2EE pathfinder: The many uses of implicit objects. http://www-106.ibm.com/developerworks/java/library/j-pj2ee7.html, 2003, requested on 2005-04-04.

[Gabh03c] Gabhart, Kyle: Implement JSP custom tags in five easy steps. http://www-128.ibm.com/developerworks/java/library/j-pj2ee8/index.html, 2003, requested on 2005-04-14.

[Gibb00] Gibbon, Cleveland: Servlet Tutorial. http://www.acknowledge.co.uk/java/tutorial/servlet_tutorial/servlets/index.html, 2000, requested on 2002-07-05.

[Gold96] Goldfarb, Charles F.: The Roots of SGML -- A Personal Recollection. http://www.sgmlsource.com/history/roots.htm, requested on 2005-11-29.

[Gold00] Goldfarb, Charles F.: Just enough XML. http://www.xmltimes.com/knowledge/xmlhandbook/2/, 2000, requested on 2001-09-17.

[Goog06] Google Internet Search Engine. http://www.google.com, requested on 2006-03-20.

[Gorm98] Gorman, Trisha: 20 Questions on XML. http://builder.cnet.com/webbuilding/pages/Authoring/Xml20/ss05.html, 1998-10-03, requested on 2001-10-02.

[GoPr05] Goldfarb, Charles F.; Prescod, Paul: XML Handbook™, Fifth Edition. http://www.xmlhandbook.com/04c-real.pdf, requested on 2006-02-20.

[Hall00a] Hall, Marty: Core Servlets and JavaServer Pages™. First Edition,

Prentice Hall, Santa Clara 2000.

[Hall00b] Hall, Marty: JavaServer Pages (JSP) 1.0. http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/Servlet-Tutorial-JSP.html, 1999, requested on 2002-07-14.

[HaNe02] Hansen, Hans Robert; Neumann, Gustaf: Wirtschaftsinformatik I. Eigth Edition, Lucius & Lucius, Stuttgart 2002.

[Hein06] The author's Website for this thesis. http://www.heinisch.cc/thesis/, 2006, requested on 2006-03-21.

[Heth97] Hethmon, Paul S.: Illustrated Guide to HTTP. http://www.manning-source.com/books/hethmon/hethmon_ch02.zip, 1997, requested on 2006-03-15.

[Hibe04] Download of Hibernate Object Relational Mapping software. http://www.hibernate.org/, requested on 2004-09-07.

[High04] Hightower, Rick: Mastering JSP custom tags. http://www-128.ibm.com/developerworks/edu/j-dw-java-custom-i.html, 2004, requested on 2005-04-14.

[Holm04] Holmes, James: Struts: The Complete Reference. First Edition, McGraw-Hill/Osborne, Emeryville 2004.

[Holz00] Holzner, Steven: Inside XML. http://www.topxml.com/xsl/articles/xsl_transformations/default.asp, 2000, requested on 2002-05-11.

[HuCr98] Hunter, Jason; Crawford, William: Java Servlet Programming. http://www.oreilly.com/catalog/jservlet/chapter/ch03.html, 1998, requested on 2002-07-02.

[Holz05] Holzschlag, Molly: XHTML 1.0: Marking up a new dawn. http://www-128.ibm.com/developerworks/web/library/w-xhtml.html, 2005-02-09, requested on 2006-03-14.

[HTML99a] SGML declaration for HTML 4.01. http://www.w3.org/TR/html4/sgml/sgmldecl.html, requested on

2005-12-18.

[HTML99b] HTML 4.01 document type definitions. http://www.w3.org/TR/html4/struct/global.html#h-7.2, requested on 2005-12-18.

[Ibat04] Download of Ibatis Object Relational Mapping software. http://www.ibatis.com/, requested on 2004-09-17.

[IBM99] Website of IBM's alphaWorks BSF project. http://www.alphaworks.ibm.com/tech/bsf, 1999, requested on 2005-09-13.

[IBM05b] Download of IBM Websphere Application Server. http://www-128.ibm.com/developerworks/downloads/ws/was/index.html, requested on 2005-03-16.

[Idri00] Idris, Nazmul: Introduction to threads. http://developerlife.com/lessons/threadsintro/default.htm, 2000, requested on 2002-06-10.

[J2EE05] JavaTM 2 Platform Enterprise Edition, v 1.4 API Specification. http://java.sun.com/j2ee/1.4/docs/api/index.html, requested on 2005-05-06.

[JaGu02] Jacobs, Ian; Gunderson, Jon et al.: User Agent Accessibility Guidelines 1.0. http://www.w3.org/TR/WAI-USERAGENT/, 2002-12-17, requested on 2006-03-14.

[Jell00] Jelliffle, Rick: The XML Schema Specification in Context. http://www.ascc.net/~ricko/XMLSchemaInContext.html, 2000-02-24, requested on 2001-10-16.

[John99] Johnson, Mark: XML for the absolute beginner. http://www.javaworld.com/javaworld/jw-04-1999/jw-04-xml.html, requested on 2001-09-13.

[John00] Johnson, Mark: Script JavaBeans with the Bean Scripting Framework - Add scripts to your JavaBeans or JavaBeans to your

scripts. http://www.javaworld.com/javaworld/jw-03-2000/jw-03-beans.html, 2000, requested on 2005-09-13.

[John01] Johnson, Mark: Elementary Entity Terminology. http://www.itworld.com/nl/xml_prac/04122001/, 2001, requested on 2005-03-24.

[Kamt00] Kamthan, Pankaj: XML Entities and their Applications. http://tech.irt.org/articles/js212/, 2000-05-21, requested on 2001-09-17.

[Kay01] Kay, Michael: XSLT Programmer's Reference. http://www.topxml.com/xsl/articles/xslt_what_about/, 2001, requested on 2002-05-12.

[Kay02] Kay, Michael: XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/, 2002, requested on 2002-05-12.

[Kay04] Kay, Michael: XSLT 2.0 Programmer's Reference, 3rd Edition. http://media.wiley.com/product_data/excerpt/90/07645690/0764569090.pdf, 2004-08, requested on 2006-03-12.

[Kay05a] Kay, Michael: XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/, 2005-11-03, requested on 2006-03-12.

[Kay05b] Kay, Michael: What kind of language is XSLT?. http://www-128.ibm.com/developerworks/library/x-xslt/, 2005-04-20, requested on 2006-03-13.

[Kolb03] Kolb, Marc A.: A JSTL primer, Part 2: Getting down to the core. http://www-128.ibm.com/developerworks/java/library/j-jstl0318/index.html, 2003, requested on 2005-04-18.

[Koo01] Koo, J.: Extensible Markup Language, XML. http://www.stanford.edu/class/msande234/HWsubmission/xml/ie275/, 2001-04-20 requested on 2001-10-17.

[Kyrn00a] Kyrnin, Jennifer: DTDs and Markup Languages.

http://html.about.com/library/weekly/aa092500a.htm, 2000-09-25, requested on 2001-10-08.

[Kyrn00b] Kyrnin, Jennifer: What is a DTD. http://html.about.com/library/weekly/aa101700a.htm. 2000-10-17, requested on 2001-10-08.

[LiBo98] Lie, Håkon; Bos, Bert: Using XSL and CSS together. http://www.w3.org/TR/NOTE-XSL-and-CSS, 1998-09-11, requested on 2006-03-07.

[Live05] Download of LiveHTTPHeaders. http://livehttpheaders.mozdev.org/, requested on 2005-02-25.

[McLa03] McLaughlin, Brett: JSP best practices: Intro to taglibs. http://www-106.ibm.com/developerworks/java/library/j-jsp07233.html, 2003, requested on 2005-04-14.

[Mage99] Fundamentals of Java Servlets Introduction. http://developer.java.sun.com/developer/onlineTraining/Servlets/Fundamentals/servlets.html, 1999, requested on 2002-07-05.

[Mahm01] Mahmoud, Qusay: Web Application Development with JSPTM and XML: Part III Developing JSP Custom Tags. http://java.sun.com/developer/technicalArticles/xml/WebAppDev3/, 2001, requested on 2002-07-14.

[Mari04] Marinacci, Joshua: Java Sketchbook: Getting Started With Scripting. http://today.java.net/pub/a/today/2004/09/20/javascript.html, 2004, requested on 2005-09-13.

[Mars97] Marshall, James: HTTP Made Really Easy. http://www.jmarshall.com/easy/http/, 1997-08-17, requested on 2005-02-26.

[McGr02] McGrath, Sean: XML IN PRACTICE. http://www.propylon.com/news/ctoarticles/XML_is_Just_a_Tree_N

ot20020117.html, 2002, requested on 2005-03-24.

[Megg98] Megginson, David: The SGML FAQ. http://math.albany.edu:8800/hm/sgml/cts-faq.html, 1998-09-16, requested on 2006-02-28.

[Meye02] Meyer, A.: Servlets - Eine Einführung. http://www.independent-web.de/advanced/java/servlets.html, requested on 2002-06-06.

[Micro06] Microsoft Corporation: Microsoft XML Parser 4.0 B2 and Windows XP. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmsxml/html/dnmsxmlfusion.asp, requested on 2006-003-12.

[Mock87a] Mockapetris, Paul: RFC 1034: Domain Names – Concepts and Facilities. http://www.ietf.org/rfc/rfc1034.txt, 1987-11, requested on 2006-03-14.

[Mock87b] Mockapetris, Paul: RFC 1035: Domain Names – Implementation and Specification. http://www.ietf.org/rfc/rfc1035.txt, 1987-11, requested on 2006-03-15.

[Morg01] Morgenthal, JP.: What is XML Schema? http://www.xml.gov/xmlschemapart1/index.htm, 2001-03-14, requested on 2001-10-17.

[MoRo00] Monson-Haefel, Richard; Rohaly, Tim: Enterprise JavaBeans™(EJB) Technology Fundamentals. http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html, 2000, requested on 2002-07-05.

[Mozi05a] Download of Mozilla Firefox web browser. http://www.mozilla.org/products/firefox/, requested on 2004-09-01.

[Mozi05b] Mozilla's Website for Rhino JavaScript for Java. http://www.mozilla.org/rhino/, requested on 2005-09-13.

[Mozi06] Mozilla: XSL Transformations (XSLT) in Mozilla. http://www.mozilla.org/projects/xslt/, requested on 2006-03-12.

[Münz98] Münz, Stefan: Selfhtml (7.0 Edition).

http://www.teamone.de/selfhtml/tbad.htm, 1998-04-27, requested on 2001-09-13.

[Münz01a] Münz, Stefan: CSS Stylesheets und HTML. http://selfhtml.teamone.de/css/intro.htm, 2001, requested on 2002-05-06.

[Münz01b] Münz, Stefan: XML-Darstellung mit CSS Stylesheets. http://selfhtml.teamone.de/xml/darstellung/css.htm, 2001, requested on 2002-05-08.

[Münz01c] Münz, Stefan: Grundlagen von XLS/XSLT. http://selfhtml.teamone.de/xml/darstellung/xslgrundlagen.htm, 2001, requested on 2002-05-11.

[Muny00] Munyati, M.: XML Toolkit Overview. http://www.oneworld.org/thinktank/iktools/xmltutorial/index.html, 2000-02, requested on 2001-09-18.

[Mysl04a] Download of MySQL Database Server. http://www.mysql.com/products/mysql/, requested on 2005-01-10.

[Mysl04b] Download of MySQL Administrator. http://www.mysql.com/products/administrator/, requested on 2005-01-10.

[Naja00] Najafi, Larry: From XML content to HTML display. http://www-106.ibm.com/developerworks/web/library/web-xml/?dwzone=web, 2000, requested on 2002-05-10.

[NCSA98] National Center for Supercomputing Applications' Web site on CGI: The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html, 1998, requested on 2006-03-16.

[Ogbu04] Ogbuji, Uche: Tip: Always use an XML declaration. http://www-128.ibm.com/developerworks/webservices/library/x-tipdecl.html, 2004-4-30, requested on 2006-02-20.

[OJB04] Download of ObJectRelationalBridge Object Relational Mapping software. http://db.apache.org/ojb/, requested on 2004-09-03.

[ooRe06] Website of Object Rexx. http://www.oorexx.org/, requested on 2006-03-20.

[Osco04] osCommerce: Open Source E-Commerce Solutions. http://www.oscommerce.com/, requested on 2004-09-22.

[PaCh05] Padmanaban, Hari Vignesh; Chopra, Pradeep: IBM XML certification success, Part 1. http://www-106.ibm.com/developerworks/edu/x-dw-x-cert1-i.html, 2005, requested on 2005-03-27.

[PeAu02] Pemberton, Steven; Austin, Daniel et al.: XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). http://www.w3.org/TR/xhtml1/, 2002-08-01, requested on 2006-03-14.

[Pele01] Pelegrí-Llopart, Eduardo: JavaServer Pages™ Specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr053/, 2001, requested on 2006-03-18.

[Pemb04] Pemberton, Steven: HTML and XHTML Frequently Answered Questions. http://www.w3.org/MarkUp/2004/xhtml-faq, 2004-07-21, requested on 2006-03-14.

[Pemb06] Pemberton, Steven: HyperText Markup Language (HTML) Home Page. http://www.w3.org/MarkUp/, 2006-02-26, requested on 2006-03-14.

[PeRo03] Pelegrí-Llopart, Eduardo; Roth, Mark: JavaServer Pages™ Specification Version 2.0. http://jcp.org/aboutJava/communityprocess/final/jsr152/index.html, 2003, requested on 2005-03-11.

[Quin05] Quin, Liam: The Extensible Stylesheet Language Family (XSL). http://www.w3.org/Style/XSL/, 2005, requested on 2005-03-30.

[RaLe99] Raggett, Dave; Le Hors, Arnaud et al.: HTML 4.01 Specification - W3C Recommendation 24 December 1999. http://www.w3.org/TR/REC-html40/cover.html, 1999, requested on 2005-12-13.

[Ray01] Ray, Erik T.: Learning XML – (Guide to) Creating Self-Describing Data. http://www.oreilly.com/catalog/learnxml/chapter/ch02.html, 2001, requested on 2006-02-18.

[ReKi01] Reif, Gerald; Kirda, Egin: Hypertext Transfer Protocol (HTTP) Tutorial. http://www.dslab.tuwien.ac.at/Task_Description/http.html, 2001, requested on 2005-02-26.

[Reum04] Struttin' with Struts. http://www.reumann.net/struts/main.do, requested on 2005-03-8.

[Roy02] Roy, Tracey: XSLT & Xpath Tutorial. http://www.topxml.com/xsl/tutorials/intro/default.asp, requested on 2002-05-12.

[ScVa02] Schmelzer, Ron; Vandersypen, Travis et al.: XML and Web Services Unleashed, Chapter 3: Validating XML with the Document Type Definition (DTD). http://www.worldofdotnet.net/internalcontent/0672323419/0672323419_ch03.html, 2002, requested on 2005-03-27.

[Sesh00] Seshadri, Govind: JavaServer Pages Fundamentals. http://developer.java.sun.com/developer/onlineTraining/JSPIntro/, 2000, requested on 2001-09-01.

[SGML90] SGML Users' Group: A Brief History of the Development of SGML. http://www.sgmlsource.com/history/sgmlhist.htm, 1990, requested on 2005-11-29.

[Shan03] Shannon, Bill: Java™ 2 Platform Enterprise Edition Specification, v1.4. http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf, 2003, requested on 2005-03-04.

[ShMa03] Shenoy, Srikanth; Mallya, Nithin: Integrating Struts, Tiles, and

JavaServer Faces. http://www-128.ibm.com/developerworks/library/j-integrate/, 2003-09-23, requested on 2006-03-19.

[Smit01] Smith, Donald: Understanding W3C Schema Complex Types. http://www.xml.com/pub/a/2001/08/22/easyschema.html, 2001-08-22, requested on 2006-03-02.

[SpBu94] Sperberg-McQueen, C.M.; Burnard, Lou: A Gentle Introduction to SGML. http://www.tei-c.org/Papers/gentleguide.pdf, 1994, requested on 2005-11-28.

[SpBu04] Sperberg-McQueen, C.M.; Burnard, Lou: A Gentle Introduction to XML. http://www.tei-c.org/P4X/SG.html, 2004, requested on 2005-12-19.

[Sper05] Sperberg-McQueen, C.M.: How schema-validity is different from being married, 2005, requested on 2006-03-01.

[Spiel01] Spielman, Sue: Designing JSP Custom Tag Libraries. http://www.onjava.com/pub/a/onjava/2000/12/15/jsp_custom_tags.html, 2001, requested on 2005-04-18.

[Star02] Introduction to Servlets. http://www.stardeveloper.com:8080/articles/061701-1.shtml, requested on 2002-06-09.

[Stear00] Stears, Beth: JavaBeans™ 101, Part I. http://developer.java.sun.com/developer/onlineTraining/Beans/bean01/index.html, 2000, requested on 2002-07-14.

[Stru05] Struts. http://struts.apache.org/, requested on 2005-05-11.

[Stru06] Struts: Action Framework User Guide. http://struts.apache.org/struts-action/userGuide/index.html, 2006-03-11, requested on 2006-03-19.

[StSa00] St. Laurent, Simon; Sall, Ken et al.: DTDs vs. XML Schemas for Data-centric Java™ Technology-baded Applications.

http://www.cen.com/ng-html/xml/schema/DTD-vs-Schema-NASA-TEAS.pdf, 2000-06-20, requested on 2001-10-17.

[Sun02a] Sun Microsystems, Inc.: Java Servlet Technology: White Paper. http://java.sun.com/products/servlet/whitepaper.html, 2002, requested on 2002-06-07.

[Sun02b] Sun Microsystems, Inc.: Glossary of Java™ technology-related terms. http://java.sun.com/docs/glossary.html, 2002, requested on 2002-06-07.

[Sun05a] Sun Microsystems, Inc.: Java Servlet Technology Industry Momentum. http://java.sun.com/products/servlet/industry.html, requested on 2005-05-06.

[Sun05b] Sun Microsystems, Inc.: Trail: JAR Files. http://java.sun.com/docs/books/tutorial/jar/, requested on 2005-05-06.

[Sun05c] Sun Microsystems, Inc: JSPs Standard Tag Library 1.1 Tag Reference. http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/index.html, requested on 2005-05-06.

[Sun05d] Sun Microsystems, Inc: JavaServer Pages™ Technology. http://java.sun.com/products/jsp/, requested on 2005-05-06.

[Sun06a] Sun's XML Glossary. http://java.sun.com/webservices/jaxp/dist/1.1/docs/tutorial/glossary.html, requested on 2006-02-23.

[Sun06b] Sun's J2EE 1.4 Glossary. http://java.sun.com/j2ee/1.4/docs/glossary.html, requested on 2006-03-17.

[ThBe04] Thompson, Henry S.; Beech, David et al.: XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/xmlschema-1/, 2004-10-28, requested on 2006-03-01.

[Thom05] Thomas, Dilip: Understanding the New Features of JSP 2.0. http://www.oracle.com/technology/sample_code/tutorials/jsp20/toc.html, 2005, requested on 2005-04-19.

[Tomc05] Download of Apache Jakarta Servlet/JSP Container. http://jakarta.apache.org/tomcat/, requested on 2005-09-21

[TuGo99] Tun, Zar Zar; Goodchild, Andrew et al.: Introduction to XML Schema. http://www.dstc.edu.au/Research/Projects/Titanium/papers/XMLSchema/Intro-to-XMLSchema.html, 1999-10-04, requested on 2001-10-17.

[Vali06] Validome HTML / XHTML / WML / XML Validator. http://www.validome.org/validate, requested on 2006-02-28.

[Vlis01] van der Vlist, Eric: Using W3C XML Schema. http://www.xml.com/pub/a/2000/11/29/schemas/part1.html?page=1, 2001-10-17, requested on 2001-10-16.

[W3C95] W3C: Logging Control In W3C httpd. http://www.w3.org/Daemon/User/Config/Logging.html, 1995-07, requested on 2006-03-15.

[W3sc01a] W3 Schools: Introduction to XML. http://www.w3schools.com/xml/xml_whatis.asp, requested on 2001-09-13.

[W3sc01b] W3 Schools: Introduction to XML. http://www.w3schools.com/xml/xml_usedfor.asp, requested on 2001-09-14.

[W3sc01c] W3 Schools: Introduction to XML. http://www.w3schools.com/xml/xml_syntax.asp, 2001, requested on 2001-09-18.

[W3sc01d] W3 Schools: Introduction to DTD. http://www.w3schools.com/dtd/dtd_intro.asp, 2001, requested on 2001-10-08.

[WaFi00] Wahli, Ueli; Fielding, Mitch et al.: Servlet and JSP Programming with IBM WebSphere Studio and Visual Age for Java. http://ibm.com/redbooks, 2001, requested on 2001-06-19.

[Wals98] Walsh, Norman: A Technical Introduction to XML. http://www.xml.com/pub/a/axml/axmlintro.html, 1998-10-03, requested on 2001-09-13.

[Watt03] Wattle Software: The XML Guide. http://xmlwriter.net/resources/xml_guide.shtml, 2003-01-15, requested on 2006-02-24.

[WuWa00] Wu, Amanda W.; Wang, Haibo et al.: Performance Comparison of Alternative Solutions For Web-To-Database Applications. http://citeseer.nj.nec.com/cache/papers/cs/20879/http:zSzzSzrain.vislab.olemiss.eduzSz~ww1zSzSlide_Show_ImageszSzSCC_AmandazSzSCC_Amanda.pdf/performance-comparison-of-alternative.pdf, 2000, requested on 2002-06-10.

[Zeig99] Zeiger, Stefan: Servlet Essentials. http://www.novocode.com/doc/servlet-essentials/index.html, 1999, requested on 2002-06-06.

9 Resources and Utilities

[Ecli04] Download of Eclipse IDE. http://www.eclipse.org/, requested on 2004-08-10.

[IBM04] Download of IBM WebSphere Studio Application Developer - Download. http://www-306.ibm.com/software/awdtools/studioappdev/, requested on 2004-09-06.

[IBM05a] Download of IBM Rational Application Developer for WebSphere Software. http://www-128.ibm.com/developerworks/rational/products/rad/, requested on 2005-04-06.

[IBM06] Download of IBM Rational Web Developer for Websphere Software. http://www-128.ibm.com/developerworks/rational/products/rad/, requested on 2006-03-20.

[Java06] JavaRanch Big Moose Saloon – a forum for all sort of Java questions. http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi, requested on 2006-02-28.

[Jenk04] Download of JSP Tree Tag from Jenkov Development. http://www.jenkov.com/index.tmpl, requested on 2004-10-25.

[JSTL04] Download of JSP Standard Tag Library. http://jakarta.apache.org/site/downloads/downloads_taglibs-standard.cgi, requested on 2004-10-25.

[Leo06] LEO. Link everything online – English/German and German/English dictionary. An Online Service by Informatik der Technischen Universität München. http://dict.leo.org/, requested on 2006-03-21.

[Mysl04c] Download of MySQL Query Browser. http://www.mysql.com/products/query-browser/, requested on

2005-01-10.