

The logo for Dublin City University (DCU) features a stylized, grey, curved graphic above the letters "DCU" in a bold, black, sans-serif font.

**DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING**

**The YBox – A Front-End Processing Engine for
Web Community based Applications**

Liam Frawley
December 2003

**MASTER OF ENGINEERING
IN
ELECTRONIC SYSTEMS**

Supervised by Dr. D. Molloy

Acknowledgements

I would like to thank my supervisor Dr. Derek Molloy for his guidance, enthusiasm and commitment to this project. I would also like to thank my family for their patience throughout the entire project. Finally I would like to express my deep appreciation to Fiona for her support and understanding over the past two and a half years.

Declaration

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: Lin Frank.....

Date: 21-02-03.....

Abstract

This document describes the YBox framework that enables web application developers to rapidly develop web applications for the Java 2 Enterprise Edition (J2EE) platform. The YBox is a fully implemented and tested framework that provides a “front-end” for Servlet Containers and contains functionality that all user-based web applications for virtual communities require. The YBox extends the functionality of the Servlet Container and the Servlet 2.3 API and is implemented in a platform independent manner, which means the YBox will run on any operating system or any Servlet Container. As a result of extra functionality added by the YBox, the presentation logic is clearly separated from the business logic in a web application. This also enables the web application to provide content to more client types and automatically perform form validation on all web forms submitted from clients. The YBox framework also allows the web application designer to control access to protected content from one central resource. Finally, the YBox framework enables session persistence across multiple sessions, therefore, user information is not lost between sessions.

Table of Contents

ACKNOWLEDGEMENTS	II
DECLARATION.....	III
ABSTRACT.....	IV
TABLE OF CONTENTS	V
TABLE OF FIGURES.....	VIII
CHAPTER 1 - INTRODUCTION.....	1
1.1. OVERVIEW OF WEB APPLICATIONS.....	2
1.2. HISTORY OF WEB APPLICATIONS	4
1.3. THE J2EE FRAMEWORK.....	5
1.4. THE .NET FRAMEWORK	6
1.5. OVERVIEW OF YBOX	8
1.5.1. <i>Content Presentation</i>	9
1.5.2. <i>Form Validation</i>	10
1.5.3. <i>Session Management</i>	11
1.5.4. <i>Security</i>	11
1.6. CONTRIBUTIONS	12
1.7. ORGANISATION	13
CHAPTER 2 - BACKGROUND TECHNOLOGY REVIEW	14
2.1. SECURITY.....	15
2.1.1. <i>Security with Servlet 2.2 API</i>	15
2.1.2. <i>Security with Servlet 2.3 API</i>	17
2.2. MULTIPLE CLIENTS AND CONTENT PRESENTATION	19
2.2.1. <i>Separate content for different clients</i>	20
2.2.2. <i>Cocoon from Apache</i>	21
2.3. FORM VALIDATION	24
2.3.1. <i>Client Side Validation</i>	25
2.3.2. <i>Java based Server Side Validation</i>	26
2.3.3. <i>Form Validation using the Form Processing API</i>	30
2.4. HTTP SESSION MANAGEMENT.....	31

2.4.1.	<i>Overview of Sessions</i>	31
2.4.2.	<i>Example of a Session</i>	32
2.4.3.	<i>Problems associated with Sessions</i>	34
2.4.4.	<i>Servlet Container Implementation of Session Persistence</i>	35
2.5.	SUMMARY	35
CHAPTER 3 - DESIGN OF THE YBOX		37
3.1.	REQUIREMENTS OF THE YBOX	38
3.1.1.	<i>Content Presentation</i>	38
3.1.2.	<i>Security Requirement</i>	39
3.1.3.	<i>Form Validation Requirement</i>	39
3.1.4.	<i>Session Management Requirement</i>	40
3.2.	ANALYSIS OF THE YBOX DESIGN	40
3.2.1.	<i>Analysis of Content Presentation</i>	41
3.2.2.	<i>Analysis of Security in the YBox</i>	44
3.2.3.	<i>Analysis of Form Validation</i>	48
3.2.4.	<i>Analysis of Session Management</i>	53
3.3.	SUMMARY	55
CHAPTER 4 - IMPLEMENTATION OF THE YBOX		56
4.1.	THE CONFIGURATION FILE.....	56
4.1.1.	<i>Accessing the Configuration File</i>	57
4.1.2.	<i>The Structure of the Configuration File</i>	59
4.1.3.	<i>Loading the Configuration File</i>	61
4.1.4.	<i>Storing the Configuration Information</i>	66
4.2.	CONTENT PRESENTATION	68
4.2.1.	<i>Static Content - Flat Files</i>	69
4.2.2.	<i>Dynamic Content - Servlets and JSPs</i>	73
4.3.	SECURITY.....	78
4.3.1.	<i>URL of the requested resource</i>	79
4.3.2.	<i>The User making the request</i>	79
4.3.3.	<i>Access permissions</i>	80
4.4.	FORM VALIDATION	84
4.4.1.	<i>Instantiating a requested Servlet</i>	84

4.4.2.	<i>Using Reflection to invoke methods</i>	85
4.4.3.	<i>Rules for Validation</i>	86
4.4.4.	<i>Redisplaying the resource with Errors</i>	97
4.4.5.	<i>Problems associated with form validation</i>	99
4.5.	SESSION MANAGEMENT.....	101
4.5.1.	<i>Using Session Listeners</i>	101
4.5.2.	<i>Registering Session Listeners</i>	103
4.5.3.	<i>Saving the Session Attributes</i>	103
4.5.4.	<i>Restoring the Session Attributes</i>	105
4.6.	SUMMARY.....	106
CHAPTER 5 - TESTING OF THE YBOX		107
5.1.	FUNCTIONAL TESTING.....	107
5.1.1.	<i>Testing Content Presentation</i>	108
5.1.2.	<i>Testing Form validation</i>	113
5.1.3.	<i>Testing Security</i>	123
5.1.4.	<i>Testing Session persistence</i>	125
5.2.	PERFORMANCE TESTING THE YBOX.....	130
5.2.1.	<i>HTML requests</i>	130
5.2.2.	<i>Single User, multiple requests</i>	131
5.2.3.	<i>Increasing number of users, fixed number of requests</i>	132
5.3.	ERROR HANDLING WITH THE YBOX.....	134
5.3.1.	<i>Configuration Errors</i>	135
5.3.2.	<i>Runtime Errors</i>	136
5.4.	SUMMARY.....	140
CHAPTER 6 - CONCLUSIONS AND FURTHER RESEARCH		142
6.1.	FUTURE RESEARCH.....	143
6.1.1.	<i>Performance of the YBox</i>	143
6.1.2.	<i>Reuse open-source frameworks</i>	144
6.1.3.	<i>New XML Schema</i>	145
REFERENCES		147
APPENDIX A – COMPLETE DIAGRAM OF THE YBOX		152
APPENDIX B – SOURCE CODE FOR SAMPLE APPLICATION		153

Table of Figures

<i>Figure 1.1. Physical representation of a 3-tiered web application</i>	3
<i>Figure 1.2. Logical representation of a 3-tiered web application</i>	4
<i>Figure 1.3. A J2EE Server</i>	6
<i>Figure 1.4. A J2EE Server with the YBox</i>	6
<i>Figure 1.5. Overview the .NET Framework</i>	7
<i>Figure 1.6. YBox introduces an extra tier</i>	9
<i>Figure 2.1. Request and Response using Servlet 2.2 API</i>	16
<i>Figure 2.2. Security implementation using the 2.2 API</i>	17
<i>Figure 2.3. Request and Response using Servlet 2.3 API</i>	19
<i>Figure 2.4. Duplication of content to support multiple clients</i>	21
<i>Figure 2.5. Cocoon supporting multiple clients</i>	22
<i>Figure 2.6. Cocoon example on (a) Internet Explorer, (b) a Nokia 6210 and (c) and PalmV PDA</i>	24
<i>Figure 2.7. Simple Form for validation</i>	27
<i>Figure 2.8. Simple form with errors</i>	28
<i>Figure 2.9. Flow Chart to validate a simple form</i>	29
<i>Figure 2.10. Class diagram for FPAPI</i>	30
<i>Figure 2.11. Clients making purchases in an online shop</i>	33
<i>Figure 2.12. HttpSession remembering the Items added to the shopping cart</i>	33
<i>Figure 3.1. YBox position in a Web Server</i>	37
<i>Figure 3.2. YBox dealing with Request and Response</i>	41
<i>Figure 3.3. Client accessing legacy HTML content</i>	42
<i>Figure 3.4. YBox producing content not based on connected device</i>	43
<i>Figure 3.5. HTTP Header with User-Agent</i>	43
<i>Figure 3.6. XSL Processor transforming XML using an XSL</i>	44
<i>Figure 3.7. File permissions on a Unix File System</i>	45
<i>Figure 3.8. Directory Structure of sample web application</i>	46
<i>Figure 3.9. The YBoxUser Abstract Class</i>	47
<i>Figure 3.10. Form Validation Failed when requesting a Servlet/JSP</i>	49
<i>Figure 3.11. YBox caches the users requested form</i>	49
<i>Figure 3.12. YBox sends back cached form to user (with error messages)</i>	50
<i>Figure 3.13. "Submit" button mapped to a method in a Servlet</i>	51

<i>Figure 3.14. Object Validation in the YBox</i>	53
<i>Figure 3.15. Class Diagram of the updated YBoxUser</i>	54
<i>Figure 4.1. The tree structure of the Configuration file</i>	61
<i>Figure 4.2. Xerces loading the XML Configuration File into memory</i>	62
<i>Figure 4.3. JAXB Compiler generating Java source files</i>	63
<i>Figure 4.4. UML representation of (a) the Access class, (b) the Group class and (c) the Person class</i>	65
<i>Figure 4.5. Sequence diagram for the init method of the YBoxFilter</i>	67
<i>Figure 4.6. Sequence diagram for the doFilter method of the YBoxFilter</i>	68
<i>Figure 4.7. YBoxFilter dealing with a request for a HTML file</i>	71
<i>Figure 4.8. Configuration file showing the browser/XSL style sheet mapping</i>	72
<i>Figure 4.9. YBoxServlet cannot modify the Response from a Servlet</i>	73
<i>Figure 4.10. Piped I/O Streams used to allow XSLT</i>	74
<i>Figure 4.11. The Class diagram for the YBox class</i>	76
<i>Figure 4.12. UML Class diagram for the FilterServletOutputStream class</i>	77
<i>Figure 4.13. Class diagram for the GenericResponseWrapper class</i>	77
<i>Figure 4.14. Dynamic Content manipulation using XSLT</i>	78
<i>Figure 4.15. UML class diagram of the YBoxUser</i>	79
<i>Figure 4.16. Security Control using the YBox</i>	81
<i>Figure 4.17. Class diagram of the resource types</i>	82
<i>Figure 4.18. YBox Configuration of the Security</i>	83
<i>Figure 4.19. Getting an instance of a Servlet</i>	84
<i>Figure 4.20. The LoginServlet Class</i>	86
<i>Figure 4.21. Simple form with one required input field</i>	88
<i>Figure 4.22. Validation on simple form failed</i>	88
<i>Figure 4.23. Simple form with one integer field</i>	89
<i>Figure 4.24. Failed to cast input to Integer</i>	90
<i>Figure 4.25. Form Validation using a Custom Class</i>	91
<i>Figure 4.26. Class diagram of the ParseException class</i>	92
<i>Figure 4.27. XML source for a form</i>	93
<i>Figure 4.28. Constructor of Shoe class</i>	93
<i>Figure 4.29. User fills out "Purchase Shoes" form</i>	94
<i>Figure 4.30. "Purchase Shoes" with error messages on manufacturer</i>	95
<i>Figure 4.31. User fills out "Purchase Shoes" form again</i>	96

<i>Figure 4.32. "Purchase Shoes" with error messages on size</i>	97
<i>Figure 4.33. Every response is stored in the Session</i>	98
<i>Figure 4.34. XML forms cached in Session</i>	100
<i>Figure 4.35. UML class diagram for the YBoxSessionListener</i>	103
<i>Figure 4.36. YBoxUser object saving the session attributes to persistent storage</i>	105
<i>Figure 4.37. YBoxUser restoring the session attributes from a flat file/database</i>	106
<i>Figure 5.1. Test Page on Internet Explorer (Windows 2000)</i>	109
<i>Figure 5.2. Test Page on Internet Explorer (Windows CE)</i>	109
<i>Figure 5.3. Test Page on a Palm V</i>	110
<i>Figure 5.4. Test Page on a WAP enabled Nokia mobile phone</i>	110
<i>Figure 5.5. Sample Document with an image</i>	111
<i>Figure 5.6. Sample Document with an image in PDF format</i>	112
<i>Figure 5.7. Sample form loaded for the first time</i>	114
<i>Figure 5.8. Completed for to show type validation</i>	116
<i>Figure 5.9. Type validation failure</i>	117
<i>Figure 5.10. Required field left blank in incomplete form</i>	118
<i>Figure 5.11. The required field fails validation</i>	119
<i>Figure 5.12. Custom class validation incorrectly filled out</i>	120
<i>Figure 5.13. Custom class validation failure</i>	121
<i>Figure 5.14. The correct input to the sample form</i>	122
<i>Figure 5.15. The resulting Servlet from the correct form</i>	122
<i>Figure 5.16. The login to the sample web application</i>	123
<i>Figure 5.17. The page user sees when he/she is denied access to a resource</i>	124
<i>Figure 5.18. Steps involved in session persistence</i>	126
<i>Figure 5.19. session.xml – A resource to test session persistence</i>	127
<i>Figure 5.20. User enters test data into the input fields</i>	128
<i>Figure 5.21. The session attribute retrieved from the users session</i>	128
<i>Figure 5.22. The session attribute not found in the session</i>	129
<i>Figure 5.23. YBox performance with a static HTML resource</i>	131
<i>Figure 5.24. A single user making 500 requests for a single resource</i>	132
<i>Figure 5.25. Web application performance with a changing number of users (static resource)</i>	133
<i>Figure 5.26. Web application performance with a changing number of users (dynamic resource)</i>	134

<i>Figure 5.27. Error message when web application is not loaded</i>	<i>136</i>
<i>Figure 5.28. User attempting to hack the web application using the URL</i>	<i>137</i>
<i>Figure 5.29. Error message associated with the wrong number a parameters</i>	<i>138</i>
<i>Figure 5.30. File not found error</i>	<i>138</i>
<i>Figure 5.31. Error message displayed when the method is not registered</i>	<i>139</i>
<i>Figure 5.32. Error message displayed when the footprint does not match</i>	<i>140</i>
<i>Figure 5.33. No method specified in the URL</i>	<i>140</i>
<i>Figure 6.1. A Web Application with the YBox and Struts combined</i>	<i>145</i>

Chapter 1 - Introduction

The YBox is a framework for aiding the design of user-based web applications for virtual communities using the Servlet 2.3 Application Programming Interface (API). The YBox framework is used by web application designers as it enables them to reduce the time taken to develop, test and deploy a web application.

The YBox gets its name from initial discussions about the implementation and where the framework should reside. From these discussions the framework was being treated as a “black-box” inside a web application. As XML is at the core of framework, the initial project name for the framework was the XBox (a combination of XML and black-box). Microsoft has a games console called the Xbox, therefore a new name was needed. It was decided to use the next letter of the alphabet, “Y”, hence the name, the YBox.

User-based web applications are very common throughout the Internet. Examples include: online banking, online shopping and web based email. User-based web applications are at the centre of e-commerce as they allow the user to interact with services and perform transactions without leaving their home.

Virtual communities allow individual users to be treated as a part of a larger group. Groups can be used to categorise people, as a result it is easier to provide specific services to groups of individuals. Examples of virtual communities include: The Virtual Community Project at DCU [1], photo.net [29] and the Nokia developers forum [2].

This chapter explains in detail what a web application is and how a web application can be represented as a series of tiers. The topic is discussed further to describe exactly what a user-based web application is. A history of web applications and how they were designed in the past is discussed. From this section it can be seen how web applications have evolved and how the YBox framework is taking that evolution one step further.

The J2EE [3] framework is described in relation to the YBox and the role the YBox plays in the development of enterprise applications is also examined. The .NET framework [4] is examined and how it is similar/different to the J2EE framework. This discussion forms the basis for using Java technology for the development of the YBox.

1.1. Overview of Web Applications

A web application is an extension of a web server that enables the server to produce dynamic content. While a web page is a simple static Hypertext Mark-up Language (HTML) file, a web application provides a more interactive experience for the user [5]. A web application usually contains dynamically generated content based on user input.

A web application enables end users to obtain information and access data in a well-defined fashion. The user is accessing the information across a network, therefore the user is known as a client. To obtain the information the user must connect to a server that is a physical device somewhere else on the network.

The client can connect to the server in several different ways. The client could be connected to the server via a wired Ethernet based network. The client could also be a wireless device such as a Wireless Application Protocol (WAP) enabled mobile phone. For this discussion it is not important how the client connects: as long as connection exists, the client is able to communicate with the server.

A web application is a generic name for an **n-tiered** server side application that handles the following:

- The presentation of the information to the client.
- The business logic.
- The storage of data.

This representation is typically called a 3-tiered web application, but each of the these tiers can be broken up into smaller tiers where appropriate. This 3-tiered model of a

web application is actually a logical representation of each tier. Each logical tier does not need to be on a physical device, but it can be. Figure 1.1 shows the physical and logical tiers in a 3-tiered web application.

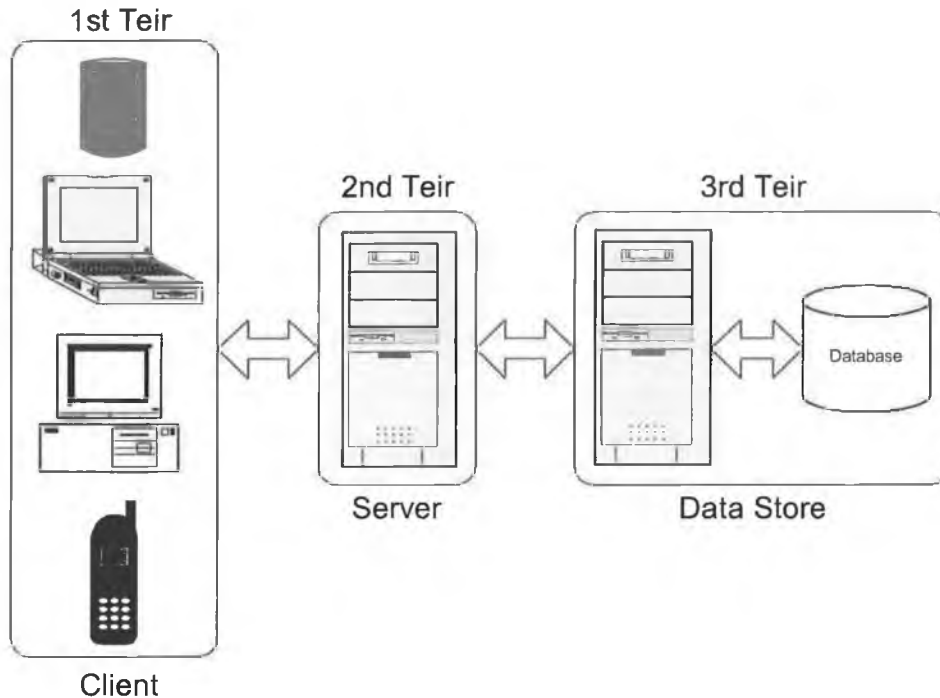


Figure 1.1. Physical representation of a 3-tiered web application

Each logical tier in Figure 1.1 has a physical device associated with it. The 1st tier (the client) encapsulates several different physical devices. The 1st tier could also represent several different client applications accessing the 2nd tier (the server). The 3rd tier (data store) is where the user information is stored. In Figure 1.1 the data store is shown as a separate physical tier.

Using this 3-tiered model, it is possible for the 3rd physical tier to be consolidated into the 2nd physical tier. In this case only one physical device is needed for the 2nd and 3rd logical tiers. Therefore, the web application is physically 2-tiered. The web application still retains a 3-tier logical structure. This can be seen in Figure 1.2.

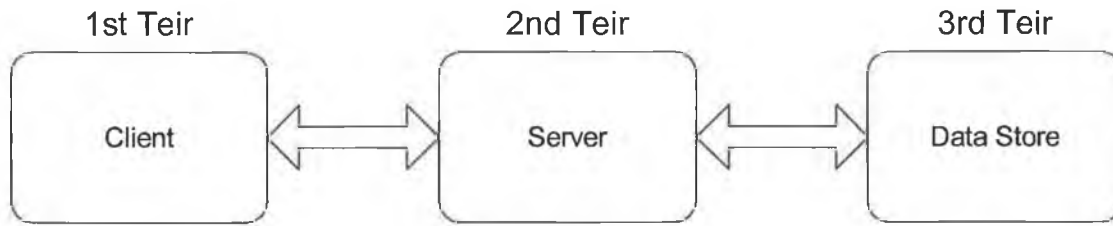


Figure 1.2. Logical representation of a 3-tiered web application

1.2. History of Web Applications

Common Gateway Interface (CGI) was one of the first methods of generating dynamic content [6]. With CGI, the web server passes certain requests to an external process. The output of this process is passed back to the server when it is complete. The server then returns the result to the client.

Perl is one of the most widely used languages for CGI programming [28] even though almost any language could be used. Perl has advanced text-processing abilities, which are of great benefit to CGI programming. The main disadvantage of Perl is a separate process must be created for every request. This results in a large load on the server for a busy web application. This also means the Perl script cannot easily access any of the features of the web server, such as; write messages to the servers log file because the script is running as a separate process and must deal with resource locking.

Java Servlets address this problem, because all requests are handled by separate threads inside the web server (Servlet Container) process. Therefore, Servlets are efficient and scalable. This also means the Servlet can access any resources available to the web server. Another advantage of Java Servlets is the fact that they are portable. Like all Java applications, they are portable across Operating Systems (OSs). Another important portability feature is they are portable across Servlet Containers allowing the web application to be deployed on a number of different Servlet Containers.

1.3. The J2EE Framework

The Servlet specification [7] is part of a much larger framework known as the J2EE framework. The YBox conforms to all of the J2EE specifications and does not affect the way the J2EE framework operates.

The J2EE framework is supported on any Operating System that supports Java. This makes the J2EE framework extremely portable. There are also several J2EE servers available at the moment; therefore the web application is not confined to one operating system and one J2EE server.

The framework is based on Enterprise JavaBeans (EJBs). There are three main kind of EJBs [8].

1. Session beans: represent the conversation between a client¹ and a server.
2. Entity beans: represent a persistent data object (usually data from a database).
3. Message beans: communicates with a Java Messaging Server.

It is not within the scope of this document to examine the J2EE framework in detail, as it is a rather large specification. Instead, the J2EE framework is looked at with respect to the YBox and where the YBox fits.

Figure 1.3 shows a J2EE Server. The J2EE Server has two main components; a Servlet Container and an EJB Container. The Servlet Container contains Servlet and Java Servlet Pages (JSPs) and deals directly with the client. The EJB Container contains EJBs (three main types of EJBs as mentioned above) and communicates with the Servlet Container and databases. It is possible for the J2EE Server to be distributed across several different physical Servers.

¹ The reference to “client” is not in the same context as mentioned through this document. The client in this context could actually be a Servlet Container. It is not a Web Browser.

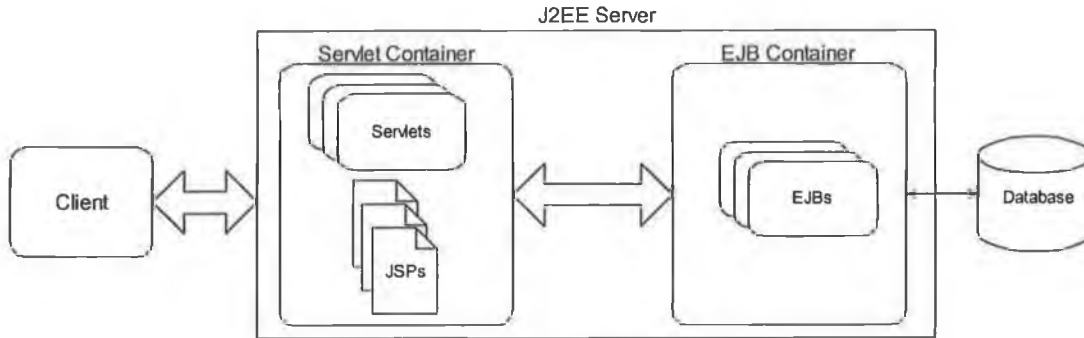


Figure 1.3. A J2EE Server

The YBox fits well into this model. It can be responsible for all communication with the client. Every client must access the J2EE Server through the YBox. Therefore, a client cannot access any Servlets, JSPs or EJBs without communicating with the YBox first. As the YBox is responsible for the security in a web application, all content and business logic is protected by a single authorisation mechanism.

Figure 1.4 shows where the YBox fits into the J2EE framework. The YBox resides towards the “front” of the Servlet Container inside the J2EE Server. The “front” means the part closest the client. If the requesting client is not a valid user, then the user will not have access to any resource beyond the YBox. The YBox must be positioned towards the front of the web application as it must have access to the request and response.

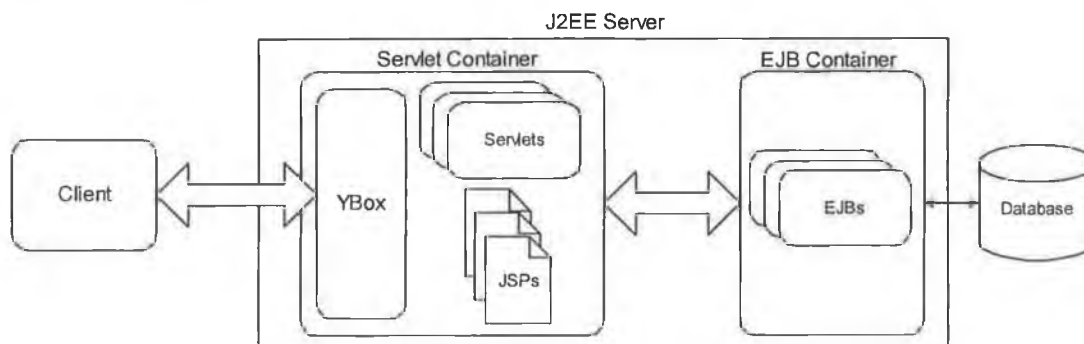


Figure 1.4. A J2EE Server with the YBox

1.4. The .NET Framework

The Microsoft .NET framework is used by developers for building, deploying and running web applications and XML based web services [9]. The lifecycle of the

framework (from development to deployment) is completely controlled by the Visual Studio.NET Integrated Development Environment (IDE).

The Common Language Runtime (CLR) [30] forms the base of the .NET framework, providing the code execution environment. The CLR allows code to be written in several languages and compiled for the .NET framework. Microsoft currently provides CLR compliant versions of Visual Basic, C#, C++, JScript and Java. It is important to note that this compiled code only executes on a Windows 2000 or Windows XP platform. Code compiled for the CLR is not OS independent like Java compiled byte code form Sun Microsystems [10].

Figure 1.5 shows the architecture overview of the .NET framework. From this diagram it can be seen how similar the high level architecture of the .NET and J2EE frameworks are. The .NET framework is broken up into a web tier, a business tier and a data tier.

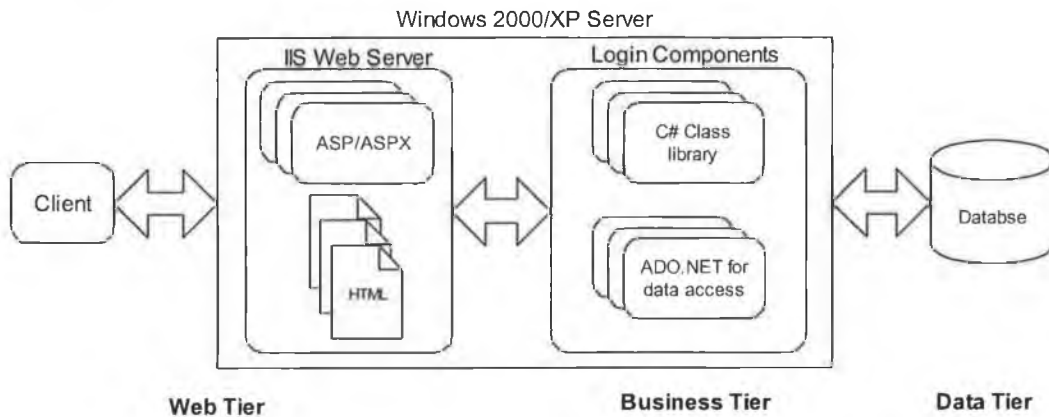


Figure 1.5. Overview the .NET Framework

The following examines each tier in more detail:

- **Web Tier:** The .NET framework builds and hosts web application under Microsoft’s Internet Information Server (IIS). Active Server Pages (ASP) .NET are used by Microsoft to provide dynamic content (the equivalent of JSPs and Servlets in J2EE).
- **Business Tier:** The Business tier can be designed using any language the CLR supports. The libraries supported by the CLR provide support for data management and XML manipulation.

- **Data Tier:** The .NET framework manages database interaction through a collection of classes known as ADO.NET.

When comparing J2EE to .NET, ASP.NET is similar to JSPs and Servlets. ASP.NET is the dynamic aspect of web content in the .NET framework. The major advantage of ASP.NET over JSP is ASP has built in form validation. This form validation can be client side, server side or both. These advanced form validation features were only introduced with the release of the .NET framework. They did not exist in the previous ASP specification. Therefore, when the design of the YBox began, automated form validation was not possible using ASP.

ASP.NET does not support the advanced security features implemented in the YBox. ASP.NET supports user authorisation, but not group authorisation. These security features must be set up manually when using ASP.NET. ASP.NET does not support multiple clients or content types. The recommendations from Microsoft [11] suggest that each resource should examine the user agent of the connecting client and modify the response based on the client. This is far from ideal. Finally, session persistence is not possible with the .NET framework. Once the session expires on IIS, the session information is lost.

1.5. Overview of YBox

The YBox is Operating System and Servlet Container independent; therefore it provides a solution that behaves in an identical manner on all platforms. The YBox does not have any dependencies on databases. As a result it does not depend on database drivers or connection issues (usernames, passwords, table structure, etc). The YBox must be implemented using the J2EE framework as the .NET framework is not OS independent. If the YBox was implemented on the .NET framework then the process of building, deploying and running a web application would be confined to the Windows OS, more specifically, Windows 2000 or Windows XP.

A web application designer uses the YBox at design time and the deployment stage of a web application. The YBox is a “front-end” to the 2nd tier, which is the **Server** tier.

This tier is actually a Servlet Container such as Tomcat from Apache [12]. The YBox is in control of all interaction between the web application and the client.

The YBox is actually another tier in a server side application. This tier can be seen in Figure 1.6. The **Server** tier is now split into two separate tiers; a presentation tier and a business tier.

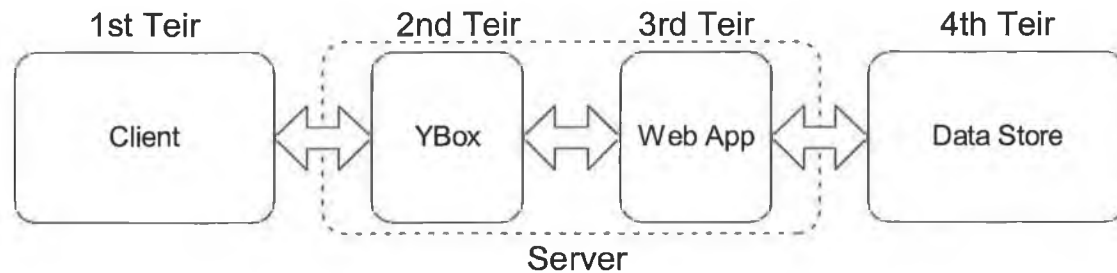


Figure 1.6. YBox introduces an extra tier

The 2nd tier (the YBox) is responsible for:

1. Content Presentation.
2. Form Validation.
3. Session Management.
4. Security.

These four points form the basis for every chapter in this document.

Points 1 and 2 above combine to form the presentation layer. All the presentation logic is now contained within the YBox tier of the web application. The 3rd tier in Figure 1.6 (the Web App) contains the business logic. This tier is responsible for connecting to the database and performing the business functionality. Using this representation of the web application it is possible that this layer could be based on the Enterprise Java Beans framework and split into more tiers if necessary.

1.5.1. Content Presentation

The YBox is responsible for simplifying the way content is managed in a web application. There must only be one source for all content in a web application and this source must be as “neutral” as possible.

Neutral source content has two meanings in this context. It means the YBox must be able to provide the content for any connecting client. This client can be:

- Any form of web browser on a PC (e.g. Netscape, Internet Explorer).
- A Personal Digital Assistant (PDA) or handheld computer (e.g. Palm, Ipaq).
- A WAP enabled mobile phone.

Neutral source content also means the YBox must be able to provide the content in several different formats:

- HTML for web browsers.
- Portable Document Format (PDF) for printing purposes.
- XML for clients that require raw XML (e.g. a speech synthesis tool).

1.5.2. Form Validation

Web forms are used for getting information from the end user to the web application. The user must enter information into a form and using the Hypertext Transfer Protocol (HTTP), this information is transferred to the Servlet Container. HTTP sends this information as strings. This protocol does not have any types (integers, floating point numbers, ... etc) associated with it. Therefore all information received from the client must be validated by the web application.

Traditionally, code associated with form validation is dispersed through the business logic code inside the web application. If the completed web form is invalid, the form must be redisplayed with error messages explaining in detail why the form is invalid. Therefore, the presentation logic has to have information about why the validation failed. The information can be linked with the business logic.

The YBox can validate web forms based on certain rules. The web application designer specifies these rules at design time and the YBox uses them during every request. These rules aid in separating the business logic from the presentation logic. The YBox can check the type of the input and compare it to the desired input type. It can also create an instance of a user defined business object and validate the form based on the successful creation of this object.

1.5.3. Session Management

A session is set of data objects stored in memory in the Servlet Container. A session is used to store user information that spans multiple client requests. Each client has a unique session ID associated with it. This is how the Servlet Container recognises each client's request. As the server does not have an infinite amount of memory, there is a way of controlling how long each session resides in memory.

The session can expire if the client associated with it does not make a request for a predefined period of time. The session can also be invalidated if the user logs out. Finally, the Servlet Container can invalidate the session if the web application or the Servlet Container is being restarted.

Session management in a web application means making user interaction over multiple sessions as seamless as possible. The end user must not lose information if they are in the middle of a transaction and for some reason their session is invalidated by the Servlet Container. This results in frustration for the end user.

The session data must be stored to a persistent storage device such as a flat file or a database. This allows the session data to be restored the next time the user returns to the web application. The end user is not aware of any of this and their experience using the web application is greatly improved.

Session management in the YBox makes it possible for the web application designer to implement a secure and platform independent solution. The YBox "listens" to events from the Servlet Container – in particular, session events. When the YBox detects a session is being invalidated, it calls a predefined storage procedure. The session data is then stored to a database or flat file. When the user returns to the web application, it is possible to restore the session information into the user's current session using the YBox.

1.5.4. Security

Security in a web application is the ability to discriminate against certain users. A security mechanism should allow the protection of information and make it

impossible for this information to be viewed by undesired users. The security mechanism must provide a way to protect all types of content, both static and dynamic.

Security always requires some form of login where the user supplies a username and password. The end user must be authenticated by the web application before being allowed to view certain information. If the user does not have the required privileges, then they should be shown an error explaining this.

As mentioned, the YBox aids the design of user-based web applications. When dealing with security, the YBox must know exactly what a user is and what privileges the user has. If the YBox detects the user does not have the required privileges, then they should not be allowed access to the requested resource.

The YBox can deal with virtual communities or groups of users. A user can be a member of one or more groups. Therefore, restrictions to certain resources can be applied to groups of users. The YBox must know what group(s) the user belongs to and base the decision to allow him/her view the resource on the group privileges.

1.6. Contributions

This document presents a framework for developing and deploying J2EE web applications that enables the developer to separate the content and business logic inside the application. This framework has been fully implemented and tested. The document begins by performing a review of existing implementations of web application frameworks. This review forms the basis for the requirements of the YBox as several shortcomings of existing implementations are discussed.

The immediate contribution of this work is an innovative approach to help solve some of the major web application development problems. This approach presents an innovative method of form validation (using user defined custom classes) which allows Object Orientated (OO) methodology to be used to validate a web form. This technique has not been implemented previously and is unique to the YBox

framework. The validity of this framework is examined through extensive testing on real life web applications.

1.7. Organisation

The organisation of this document is as follows: Chapter 2 introduces the problems encountered when designing a web application in greater detail. It presents a review of current technologies that solve these problems, why they are not suitable and why there is a need for the YBox. Chapter 3 discusses the requirements and architecture of the YBox framework. Chapter 4 describes the implementation of the YBox framework in detail. It discusses the technologies used and how the evolution of these technologies affected the implementation. Chapter 5 examines the testing of the YBox. The development of a sample application performs functional testing. The performance of the YBox is also documented in this chapter, as comparisons are made between the performance of a web application designed with the YBox and a web application designed without the YBox. Finally, Chapter 6 summarises the work and suggests future research directions.

Chapter 2 - Background Technology Review

When this project began, the release version of the Servlet API from Sun Microsystems was version 2.2 [13]. This version tackled some of the problems web application developers were having at the time, but still left a lot of them unresolved. Some of the thought process and motivation behind the YBox was based on version 2.2 of the Servlet specification. Version 2.3 of the specification has taken steps to solving some of these problems.

The current implementation of Servlet Containers and the Servlet 2.3 specification still has several shortcomings. The Servlet 2.3 API is excellent for designing a small web application that has only a few Servlets and JSPs. These shortcomings become a problem when the web application in question is large and complex.

There are several questions that have to be asked before designing such an application. Four major questions are listed below:

1. How is security going to be dealt with?
2. Will the application support multiple content types and multiple client types: PDA, PC, and Cell Phone?
3. How will user input be validated?
4. How will the user's session be dealt with?

A web application designer can design complex web applications with the current 2.3 API and a Servlet Container that supports it, but the chances of bugs and the security risks increase exponentially as the application grows. Dealing with these security risks results in much repetition of code (i.e. security check at each entry point), so the web application risks being insecure and being hacked!

The YBox is described in more detail later, but before this it is important to understand the way a web application is designed at present. The questions mentioned above are examined individually and examples given for each. From this discussion it becomes clear where the shortcomings of the Servlet specification are and why they need to be addressed.

2.1. Security

Security is one of the most important considerations when planning a web application. Protecting all information on a web server (static content, images, documents and dynamically generated content) can be a difficult task with version 2.2 of the Servlet API. Version 2.3 of the Servlet API has implemented “Container Managed Security” [14] which is a giant step forward from a security perspective, but it still has some way to go as it only implements security controls on Servlets and JSPs.

Security is only an issue when data on the server should only be accessible to one user or a group of users. There must be some authentication on the server to recognise exactly who is logged in and what resources they have access to. To understand this fully, an example is needed to show how security is dealt with inside a web application. This example is explained with respect to version 2.2 and 2.3 of the Servlet API.

2.1.1. Security with Servlet 2.2 API

Version 2.2 of the Servlet API does not have any support for Container Managed Security, therefore some of the Servlet Container vendors included a proprietary system for security management. This idea defeats one of the major advantages of Java and of Servlets – that is the Java compiled byte code is platform independent and the Servlets are container independent. By choosing to design a web application with a Servlet Container that has a proprietary security mechanism, the web application is bound to that Container and cannot be easily ported to any other container. If the designer decides not to use proprietary Container Managed Security (generally a good idea), they usually must include code inside each Servlet/JSP to ensure a desired user or group of users can only access the resource.

Putting security related code inside every Servlet and JSP leads to code repetition, and the risk of introducing a bug into the web application. It also results in presentation code getting mixed with business logic. This security code is essential but it increases the risk of the web application being hacked (more lines of code implies more testing).

From Figure 2.1 it can be seen that there need to be security checks inside *private.jsp*. This JSP needs to get the user information from the session (if the imformation exists) and validate it, before displaying *private.jsp*.

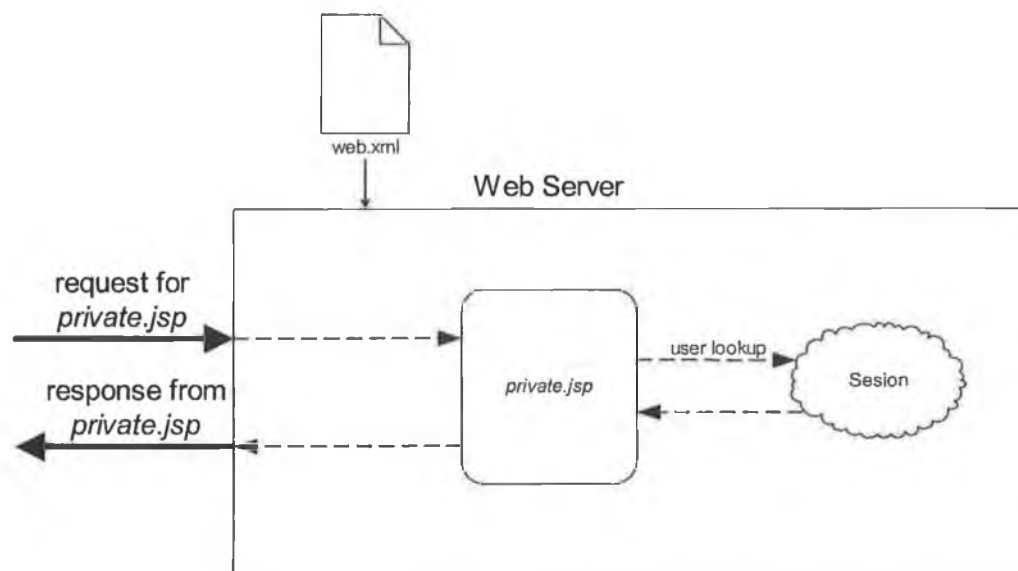


Figure 2.1. Request and Response using Servlet 2.2 API

Generally speaking, all security implementations using the 2.2 API are based on a similar idea:

1. The user requests a resource.
2. They get redirected to a login page if they are not already logged in.
3. If the login is successful and the user has adequate permissions, they are shown the requested resource.
4. Otherwise, a failure message is reported to the user.

An example of this can be seen in Figure 2.2.

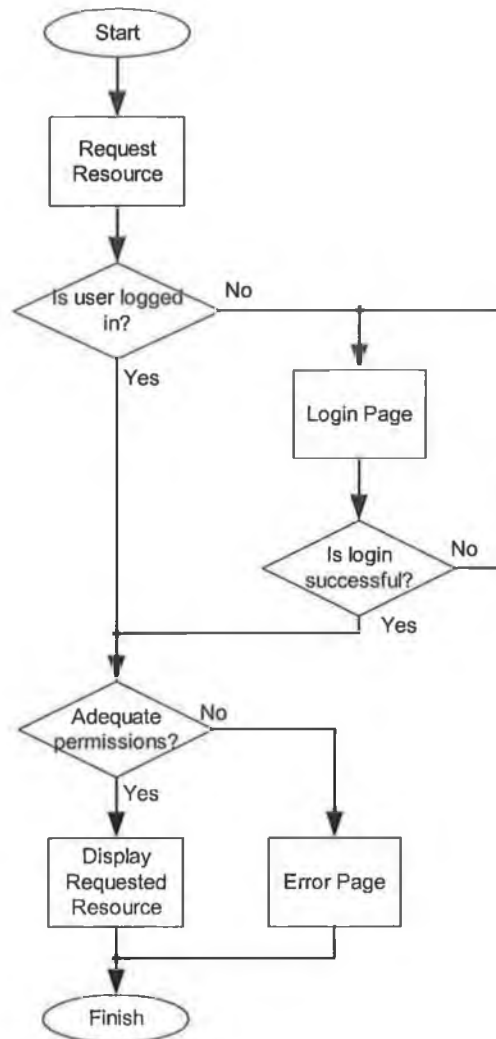


Figure 2.2. Security implementation using the 2.2 API

When the user is logged in successfully, the user name is stored in the Session. For every protected request, their username must be checked and compared with some set of access permission rules. These rules may be stored in a database.

This approach is a painful, code intensive way of checking security. As can be seen from Figure 2.2, if there is even the slightest bug in the security code, or the security code is not included in one Servlet/JSP, the whole web application becomes exposed to hackers.

2.1.2. Security with Servlet 2.3 API

Using version 2.3 of the Servlet API, there is a way to perform authentication that is platform and web server independent. This method is called Container Managed

Security. This means that the vendor who designed the Servlet Container had to conform to strict guidelines laid down by Sun Microsystems so that a web application will run inside any container without modifications. For example, security constraints that were designed and tested on Tomcat 4 from Apache [12] will behave the same on a BEA server [15] or any other Servlet Container for that matter.

The security information is contained within the deployment descriptor, *web.xml*. Every Servlet Container requires this file to initialise each web context it is going to host. The file contains a list of Servlet classes and mappings between the class name and the URL, a welcome file, mime mapping and an error page.

To understand how the Servlet Container controls security and authentication it is best to use an example. Below is a snippet from a *web.xml* file that shows a Servlet declared and the security constraints associated with that Servlet.

```
<Servlet>
  <Servlet-name>testServlet</Servlet-name>
  <Servlet-class>TestServlet</Servlet-class>
</Servlet>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected
    </web-resource-name>
    <url-pattern>/testServlet</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>student</role-name>
  </auth-constraint>
</security-constraint>
```

From this example it can be seen that there is an `<auth-constraint>` tag that specifies that only users who are a member of the student role are allowed access the `/testServlet` resource. The Servlet Container checks this role before the request is passed onto the Servlet. This can be seen in Figure 2.3.

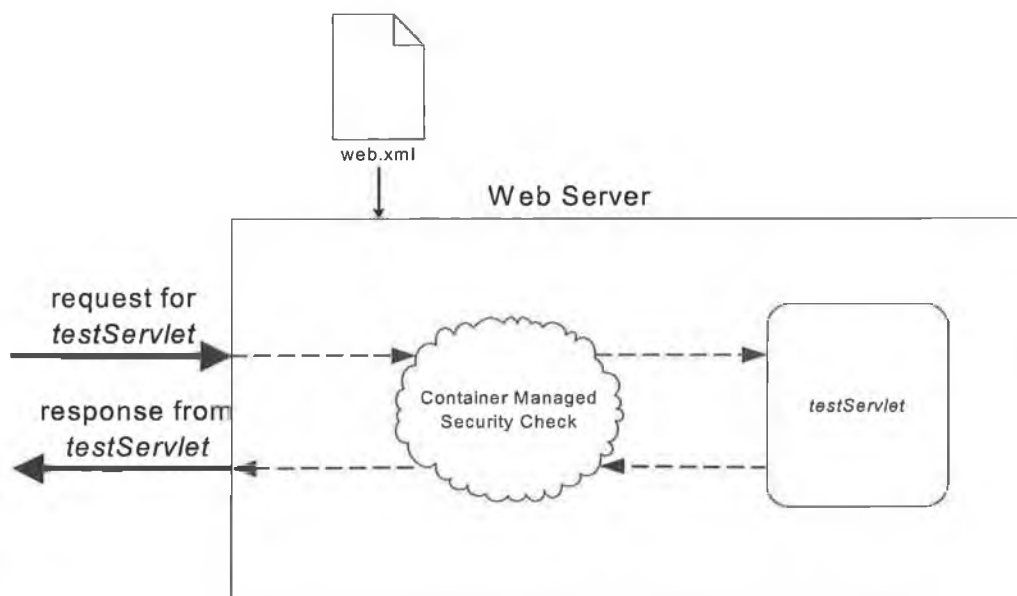


Figure 2.3. Request and Response using Servlet 2.3 API

The service method of *testServlet* does not get called unless the user has successfully logged in and had a role associated with him/her that matches the student role. This method of security does not apply to flat files², therefore a different security mechanism would have to be used. As a result, the security mechanism introduced in the Servlet 2.3 API is not suitable for use in the YBox.

2.2. Multiple Clients and Content Presentation

Content presentation is a consideration that is often overlooked when designing a web application. It can end up causing serious deployment problems, as it can be virtually impossible to write Graphical User Interface (GUI) code that is recognised by all client types. If content presentation is not taken into consideration during the design and testing phase, the web application may only be accessible to clients the web application has been tested on.

There are 2 existing solutions to this problem:

1. Use separate content for different client types (i.e. HTML, WML, XHTML³).

² A flat file is one that is not executed on the web server.

³ XHTML is well formed HTML and all tags conform to the W3C recommendations [19].

2. Use Cocoon from Apache [16], which transforms XML into client specific GUI code (HTML, WML, etc.) at run time.

Both solutions have their problems, but Cocoon is definitely a better implementation. Taking a closer look at these solutions exposes the weaknesses of both.

2.2.1. Separate content for different clients

This approach is a poor solution as it does not scale and as an application grows, the problem tends to grow also. Separate content for different clients involves having separate static files for the different clients and also having separate generators for dynamic content. This solution is recommended by Microsoft when providing content for Internet Explorer for the Pocket PC and Internet Explorer for a desktop PC [11].

For example, when accessing a web site from a PC using Internet Explorer, the URL for a particular resource is:

http://hostname/testPage.html

When accessing the same resource from a mobile phone using WAP, the URL is:

http://hostname/testPage.wml

The content within the page has to be duplicated in both static files. If the content ever changes, then both files must be updated. This problem spirals out of control the more clients the web application has to support. If the web application has to support the latest PDA on the market, then new files are needed such that the clients GUI is displayed properly.

Also if a printable version of the content is needed, then the web application designer needs to store a PDF or Word Document version of the content on the server. In Figure 2.4 it can be seen just how many copies of identical content needs to be stored on the server to serve different types of clients.

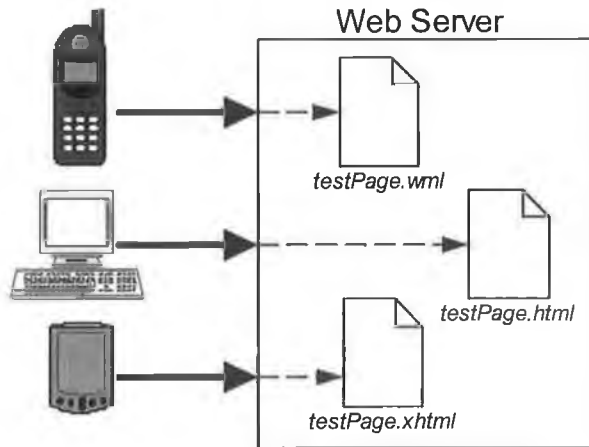


Figure 2.4. Duplication of content to support multiple clients

From Figure 2.4 it can be seen why this solution is not a realistic one when designing a large web application. Using this solution it would be very difficult to maintain all the different versions of content on the server and ensure they are all up to date. Testing such an application would be difficult as there are so many client types required.

2.2.2. Cocoon from Apache

Cocoon from Apache [16] is an advanced XML publishing framework that can be used in a Java enabled web server to serve multiple clients from a single XML source. When research first began on the YBox, Cocoon did not have as many features as it has right now and it was not as flexible. Cocoon fails to implement dynamic content generation in a manner that is acceptable to web application developers as XML processing instructions are used to trigger Java logic execution.

At the time of writing this document, Cocoon is on version 2.03, but when work on the YBox began, Cocoon was on version 1.02. There is not much point discussing what Cocoon could do back then because it has progressed enormously since. By looking at the way Cocoon deals with multiple clients and content presentation, it is possible to get an understanding just how important Cocoon is and where it fits into the development of a web application.

Cocoon “sits” in a very similar position to the YBox when looking at the big picture. It is a “front end” to a web application. From Figure 2.5 it can be seen where exactly Cocoon resides in the web application server.

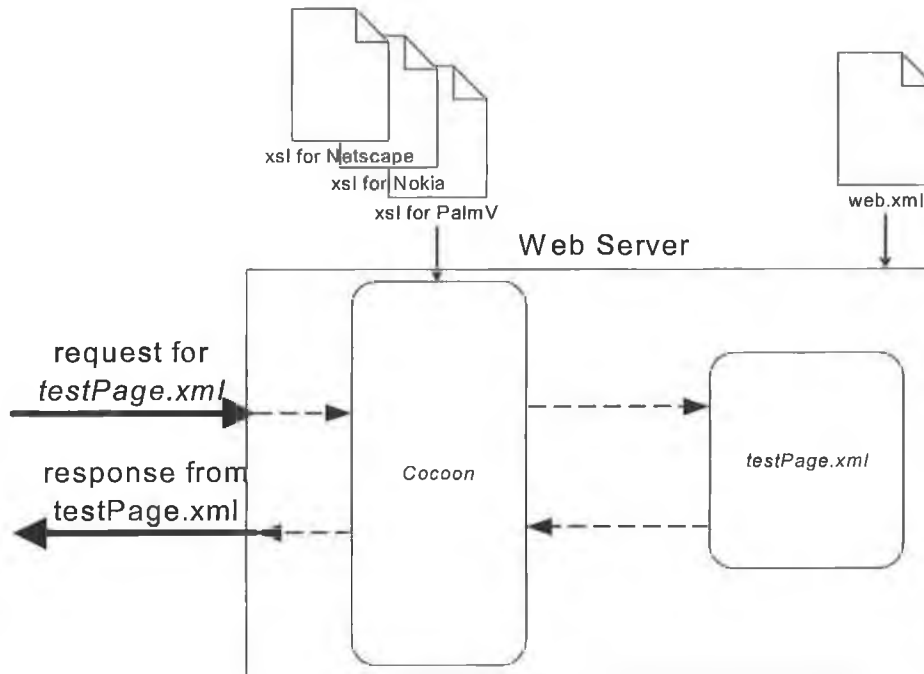


Figure 2.5. Cocoon supporting multiple clients

Cocoon examines every request before the request is passed onto the requested resource. Cocoon also examines every response before sending it back to the client. Cocoon may even modify the response, depending on the configuration.

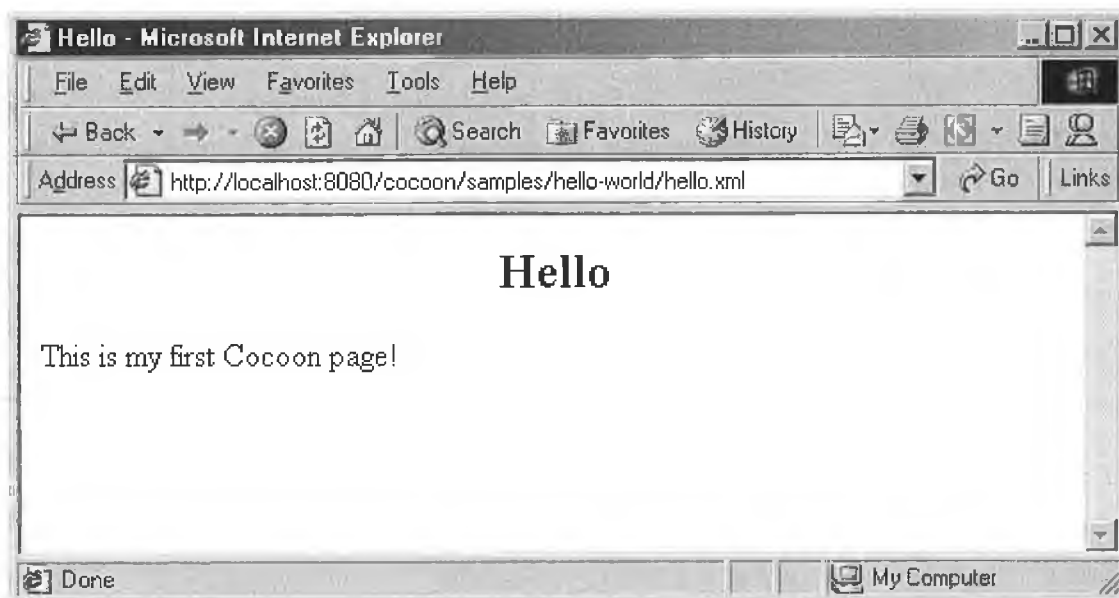
Each client supported has a different user agent declared in the HTTP header. The user agent identifies a browser and each browser has its own unique user agent. Cocoon extracts the user agent from the HTTP header as the request is being passed to the requested XML resource. This allows Cocoon to choose an appropriate XSL (eXtensible Style-sheet Language) [31] file that transforms the XML source into something the client understands. The transformation is performed using XSL Transformation (XSLT) [32].

The web application designer specifies these XSL files in the configuration file. Cocoon picks up this configuration file at initialisation and uses it to decide what types of clients to support at runtime. There is also a default XSL file for clients that

are not recognised (or not supported by the web application). This default XSL file should output well-formatted HTML that most client types can understand.

The next three figures show Cocoon in action. This is a live example that is included in the download bundle with Cocoon. All images show an identical URL. They are all requesting *hello.xml*. Cocoon then examines the user agent in the HTTP header when the page is requested and transforms it differently, depending on the client.

Figure 2.6(a) shows the page when viewed on Internet Explorer 6. The XML is transformed into HTML. There is nothing very complex about the page, but the capabilities of Cocoon become clearer when the same URL is viewed on a PDA or mobile phone. Figure 2.6(b) shows the page on a Nokia 6210. The XML source is actually transformed into WML by Cocoon based on the phones user agent. Figure 2.6(c) shows the page on a PalmV PDA.



(a)



Figure 2.6. Cocoon example on (a) Internet Explorer, (b) a Nokia 6210 and (c) and PalmV PDA

This simple example shows the uses of Cocoon. Write the content in XML and it can be displayed anywhere! Using speech synthesis tools it is even possible to convert the XML into speech. For printing purposes it is possible to convert the XML into PDF using the Formatting Object Processor (FOP) [20]. This is discussed further with respect to the YBox in section 4.2.1 as the YBox also has these same features.

When the YBox was being designed, Cocoon did not deal with content that needed to be generated dynamically from Servlets. Cocoon used a new technology called XSP, which involved embedding logic in the XML. The principal is identical to JSP, but the syntax is different, therefore the designer has to learn a new syntax structure.

2.3. Form Validation

Form validation is usually the part of the design web application developers dislike the most. It is repetitive, boring and tends to lead to difficult testing and trying to catch corner cases. The reason form validation is so complex is because the designer does not know how the end user will fill out the form, when they click submit, how

often they use the “Back” button on the browser or how long they take to fill out the form.

There are some products on the market that help in form validation, most notably Struts from Apache [17] and the Form Processing API [18] (both open source). Struts has some very advanced features and not just for Form Processing. It is based on the Model View Controller architecture and is intended for use with large web applications. Struts does not support multiple clients.

In the next three sections, client side validation, Java based server side validation and the Form Processing API are examined.

2.3.1. Client Side Validation

Client side validation was the first generation of user input validation technology for web applications and many developers still use it today. The validation logic is implemented in JavaScript embedded in the HTML that is sent to the client. JavaScript has the advantage of reducing the number of requests and responses needed to successfully fill out a form.

This approach to data validation has the following disadvantages:

- Successful validation relies heavily on client-side configuration. An end user may decide to have JavaScript disabled on his or her browser (e.g. to avoid popup advertising).
- Different browsers support different feature sets of JavaScript and not all browsers implement the feature set in the same manner. Therefore JavaScript that works on Internet Explorer may not work on Netscape and vice versa. It is often necessary to use different JavaScript in different browsers to achieve the same function.
- Some browsers do not support JavaScript at all. For example, Palmscape for the Palm OS does not support JavaScript. In that case, it is not possible to implement client side form validation.
- The data validation logic is an integral part of the business logic. By embedding the validation logic in HTML, it is removed from the business

layer and combined with the presentation layer. This violates the principles of three-tiered architecture, introducing unnecessary coupling that reduces flexibility.

2.3.2. Java based Server Side Validation

Java based server validation is a more robust way of handling data validation. The ability to code validation logic directly in Java provides flexibility and portability. However, for large forms this approach becomes cumbersome because it can require many lines of code to validate one input parameter. The coding process is repetitive and labour intensive and the final code is lengthy.

Java based server validation is also resource intensive and for the validation to be transparent to the user, the network that the web application is running on must have a fast response time. Every time the user submits a form, there is additional traffic on the network and additional processing on the server. If the network is slow (e.g. a 56K modem) then the response time from the server can take quite some time and the web application performance may not be acceptable to the end user.

The fatal weakness of this approach is the inability to customise the validation rules once the application is deployed. The entire software development process (code, debug, test, deploy) needs to be repeated to modify the Java code that implements the rule changes.

To understand how complex a simple task becomes with this method of validation, take the following example: the user is required to fill in their name, age and date of birth. An example of what the partially filled out form would look like is shown in Figure 2.7.

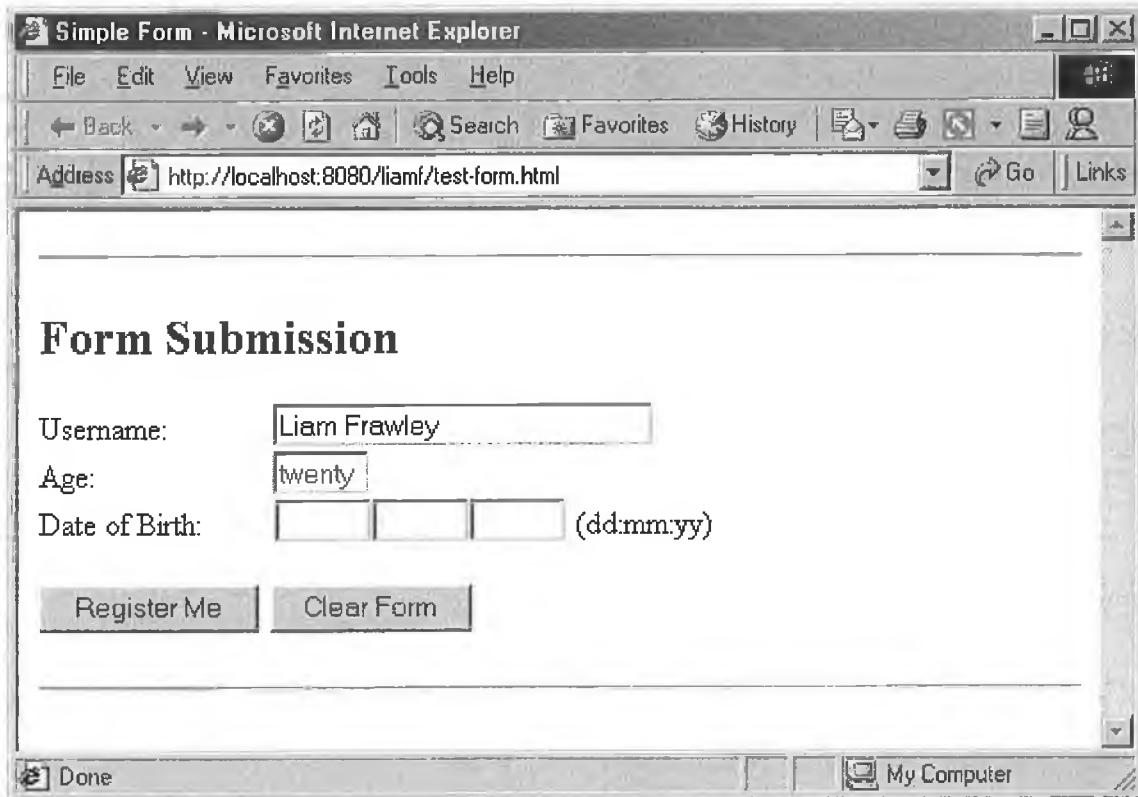


Figure 2.7. Simple Form for validation

The user has entered their name correctly. The age field appeared correct to the user while filling the form, but the Servlet expects the age to be an integer. Therefore this is going to cause an error. The Date of Birth field is also left blank – this also causes another error, as the data of birth of the user is required by the Servlet.

When the user clicks “Register Me”, the form gets posted to the server and the Servlet replies with the result in Figure 2.8.

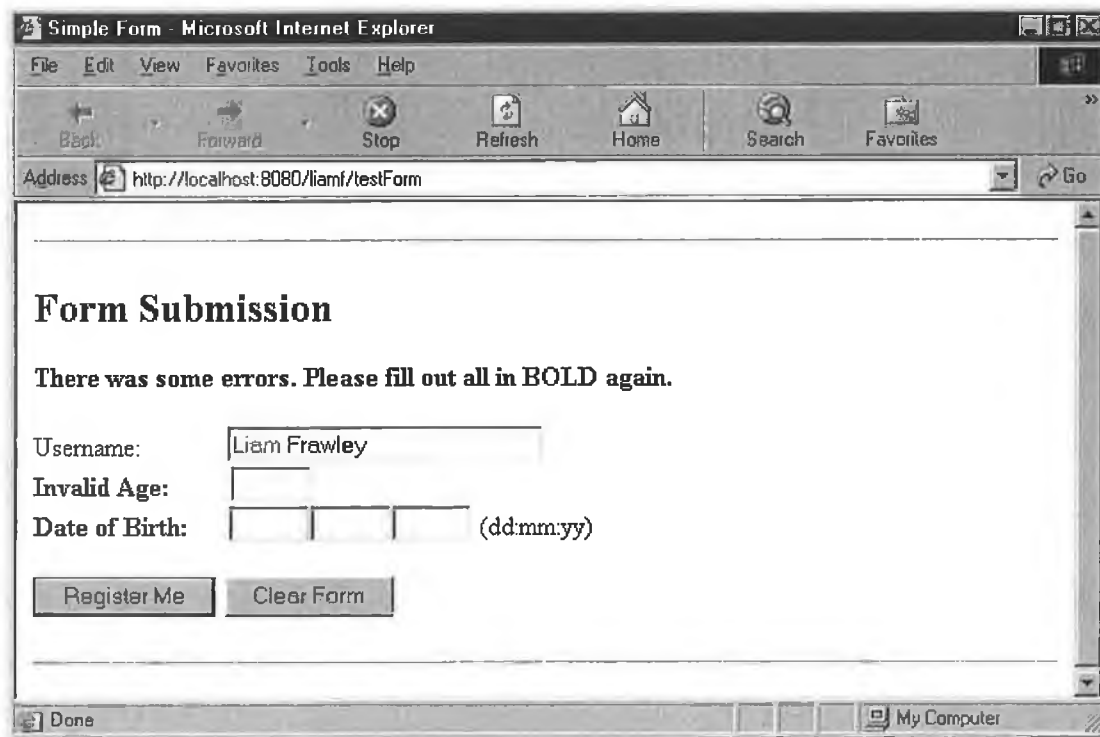


Figure 2.8. Simple form with errors

The resulting page is a typical response from a Servlet. The text in **bold** is the errors the user made in filling out the form. The flow chart for this Servlet can be seen in Figure 2.9. From this diagram it can be seen just how complicated simple form validation is. As forms grow in size, so too does the validation logic associated with the form.

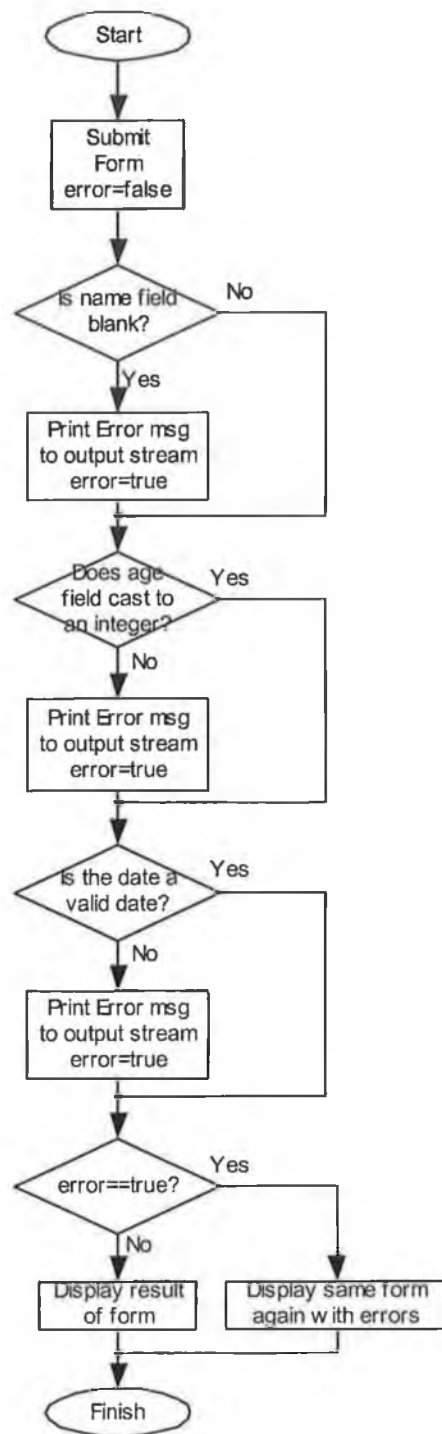


Figure 2.9. Flow Chart to validate a simple form

The flow chart in Figure 2.9 is complex, and if this is combined with the flow chart for security (see Figure 2.2) the number of lines of code in every Servlet is substantial, even before the presentation logic is implemented. To test the Servlet with this method of form validation can be difficult. One change to the business logic

results in a change to the presentation logic. The knock on effect results in changes to the validation logic and hence the Servlet needs to be tested again.

2.3.3. Form Validation using the Form Processing API

The Form Processing API (FPAPI) presents a more Object Oriented approach to validating a form. It is completely Java based, and works with version 2.2 or 2.3 of the Servlet API making it a very portable solution.

At the class level is consists of two main classes; FormElement and Form and two interfaces; GroupValidator and FieldValidator. Figure 2.10 shows how these classes are related.

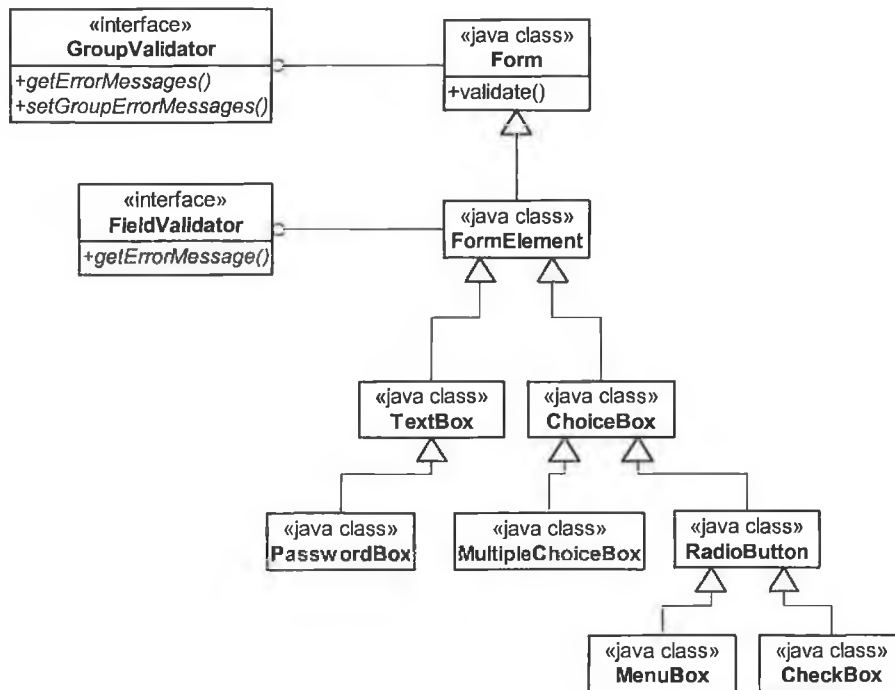


Figure 2.10. Class diagram for FPAPI

The FormElement class is the parent of all form input classes. Text fields, password fields, text areas, checkboxes, radio boxes, combo boxes and list boxes are all child classes of FormElement. These classes are all the available GUI elements available under HTML 4.0.

The FormElement class has four main attributes:

1. A name that identifies the field to the server (a string).
2. A value that represents the data that was entered in the form (a string).
3. A required flag that shows whether or not that particular field is required (boolean).
4. An errorMessage to be displayed to the client if the field fails validation (a string).

Each `FormElement` also has a `GroupValidator` and `FieldValidator` associated with it. Once a validator is registered with a `FormElement`, the validation is implemented by the FPAPI. The methods `getErrorMessage()` and `getErrorMessage()` still have to be implemented in the `FormElement` class (because both validators are interfaces). This is where all the validation logic sits and it is up to the designer to implement this logic. Now the business logic is clearly separated from the presentation logic.

One problem with FPAPI is that it does not support forms that are generated from flat HTML files. The form has to be implemented as a Servlet or JSP. The reason is that each input field has to be instantiated, as each field is as an object (instance of `FormElement`). For this reason it does not meet the needs of the YBox and FPAPI cannot be used.

2.4. HTTP Session Management

Before going into detail about the current problems with Session Management in web applications an explanation is presented on how sessions are implemented in the Servlet API.

2.4.1. Overview of Sessions

Sessions are important in web applications because it helps the designer to overcome the fact that HTTP is a stateless protocol. When a browser requests a web page, the browser establishes a Transfer Control Protocol (TCP) connection with the web server. This connection only exists until the requested page is retrieved after which the connection is broken.

This makes HTTP a very scalable protocol but it does cause a problem when trying to maintain state. This is where the Servlet API is very useful. The Servlet specification requires that the Servlet Container must be able to uniquely identify each client by inspecting the client's request.

The Servlet API has a `HttpServletRequest` class for handling client requests. This `HttpServletRequest` class has a `getSession()` method, which returns an instance of the `HttpSession` class. This object is stored in memory by the Servlet Container and can be accessed each time the client makes a request. This allows the web application designer to keep track of individuals.

The `HttpSession` has methods `getAttribute()` and `setAttribute()` that allow the designer to store information in the session that can be retrieved in subsequent requests. The best way to understand this concept and appreciate the benefits associated with it is to use an example.

2.4.2. Example of a Session

Take an online shop where the user can chose items from a catalogue and purchase them with a credit card.

- The user can chose one item and add it to a “shopping cart”.
- He/she can then go on to chose another item and add it also.
- When the user decides they have enough, they take their items to a “check-out” and pay for the goods.

Figure 2.11 illustrates this.

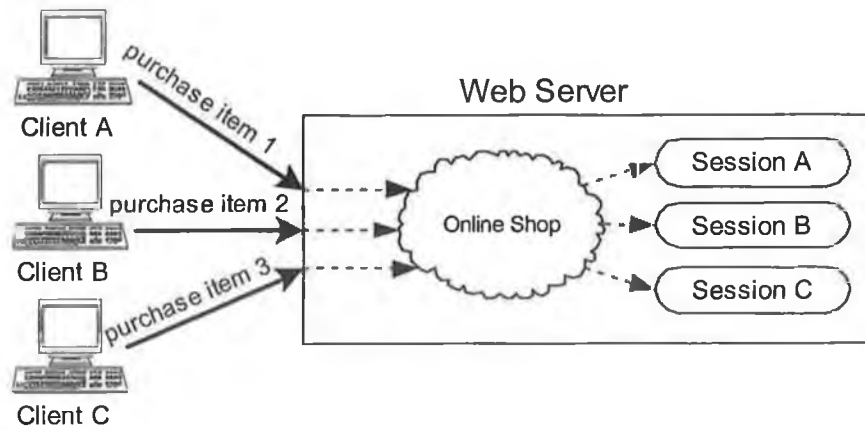


Figure 2.11. Clients making purchases in an online shop

This example could take several requests and responses to complete one sale. There is a need to “remember” what the user has in their shopping cart between requests. This is where `HttpSessions` are used. The web application designer uses the `setAttribute()` method to add items the user wishes to purchase to the `HttpSession`. The `HttpSession` object can almost be thought of as the shopping cart. As can be seen from Figure 2.12 the session for Client A “remembers” that Client A has Item 1 in his/her shopping cart, the session for Client B “remembers” Item 2 is in his/her shopping cart etc.

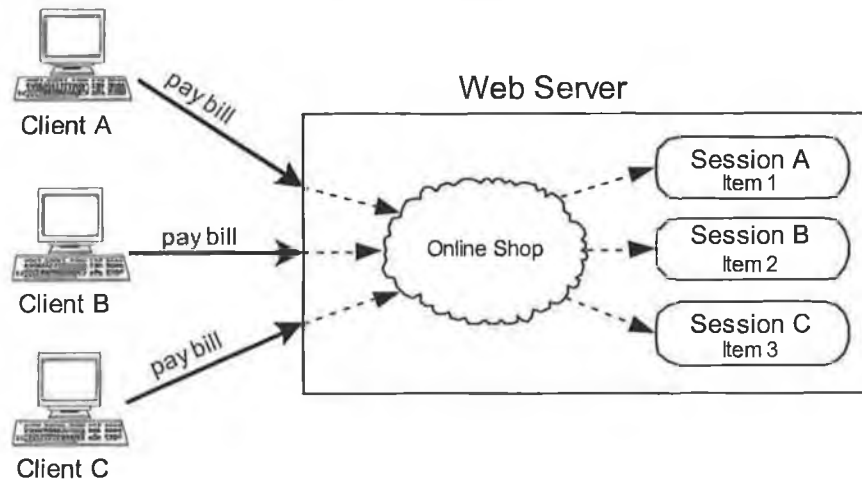


Figure 2.12. HttpSession remembering the Items added to the shopping cart

When it comes to paying the bill, each client only pays for items they added to their shopping cart.

2.4.3. Problems associated with Sessions

The problems occur when Client B leaves the online shop and decides he/she does not wish to purchase Item 2. Session B still exists in the Servlet Containers memory. Therefore, there needs to be a way to destroy Session B.

The `HttpSession` object lives in the Servlet Containers memory for as long as specified by the *web.xml* configuration file. The `<session-config>` tag shown in the *web.xml* extract specifies the number of minutes the session will reside inactive in memory before being destroyed.

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

From this piece of the configuration file, if the user does not access the web application for more than 30 minutes, then the Servlet Container unloads the `HttpSession` object out of memory.

This solution also has problems associated with it. During the purchase, Client C may go to lunch and come back 40 minutes later. When he/she returns and tries to pay for Item 3, he/she will not be “remembered” by the server and will have to start all over again. This does not make good business sense, as customers will not want to repeat the process again.

To overcome the problem with sessions being destroyed without trace, version 2.3 of the Servlet API allows listeners to be registered with the Servlet Container. This enables the Servlet Container to throw an event when it is about to destroy a session. These listeners must be registered in *web.xml* before the Servlet Container loads the web application.

To create a listener, the designer must write a Java class that implements the `javax.servlet.http.HttpSessionListener` interface. The Servlet Container calls the `sessionDestroyed(HttpSessionEvent)` method of this class every time a session is invalidated. The designer must then write code to save the session to the hard disk or to a database.

With this implementation, when Client C returns from lunch 40 minutes later, the designer reloads the HttpSession from hard disk/database and Client C is able to continue and pay for Item 3. This is quite some effort every time session persistence needs to be implemented in a web application.

2.4.4. Servlet Container Implementation of Session Persistence

Most Servlet Containers support Session Persistence in some form. Tomcat for example supports storing the session to a file or a database. WebLogic from BEA also supports the same but all Session Persistence methods are proprietary to each vendor. To set up Session Persistence in a Servlet Container involves editing the configuration file for the container and not *web.xml*, the deployment descriptor associated with each web application. This means that if a web application is designed with dependencies on the Servlet Container to handle Session Persistence, the web application will not be portable to other Servlet Containers.

2.5. Summary

The problems with the Servlet API and Servlet Containers have been clearly stated and discussed. Some of the more popular solutions to these problems and their impact on the design of a web application can be seen. Each major problem and solution was dealt with individually:

1. Security in a web application.
2. Managing content for multiple client types.
3. Form Validation.
4. Session persistence.

To make it clearer, examples were given for all problems discussed. The shortcomings of these solutions were pointed out. Where necessary, the differences between version 2.2 and 2.3 of the Servlet AIP were noted. This is because the design of the YBox spans the two versions of the API.

The next chapter discusses the requirements and architecture of the YBox. It becomes clear how the problems discussed in this chapter are tackled using a solution that is portable across all Operating Systems and Servlet Containers.

Chapter 3 - Design of the YBox

The YBox is designed with **user-based** web applications in mind at all times. The design of the YBox involves breaking down the central requirements into separate, manageable blocks. The requirements of the YBox are:

- Content Presentation.
- Security.
- Form Validation.
- Session Management.

The YBox must be portable across all Servlet Containers and J2EE Application Servers (version 2.3 and greater of the Servlet API). This means that the features required cannot be implemented in a manner that will only run on certain Servlet Containers. The YBox must be portable across Operating Systems. To ensure this, the design must use the Servlet API and not use any proprietary Servlet Container specific implementations.

The YBox cannot take the functionality of the J2EE framework away from the web application developer. The developer must have the flexibility of JSPs and Servlets available to them. They must still be able to connect to databases and perform any task they are able to perform without the YBox present. Figure 3.1 shows where the YBox fits into a web application.

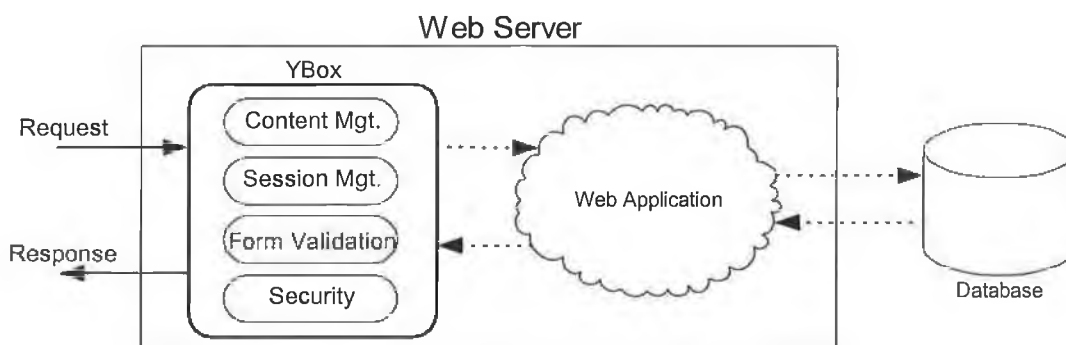


Figure 3.1. YBox position in a Web Server

3.1. Requirements of the YBox

The four requirements listed previously are the core functionality needed in the YBox. As these features are implemented in the YBox, web application design becomes a much easier task. Every user-based web application uses all of these features, so it should be possible to encapsulate all of them into one design and reuse this design across all user-based web applications.

Before attempting to design the YBox each element of the requirements needs to be discussed further to ensure the needs are correctly understood.

3.1.1. Content Presentation

In its simplest form managing content presentation means one source for all content. For static content, there should be no need to have one set of content represented in HTML for web browsers, one set of content represented in WML for WAP enabled mobile phones and so on. For dynamic content (JSPs/Servlets), there should be no need to have separate generators either.

When the web application developer has to maintain separate representations of the same content, they run the risk of providing information that is out of date. Using this technique it is cumbersome for the developer to update several content sources when only one piece of information changes. The YBox is required to have one source for all content and this source provides the information to all clients.

The YBox is also required to provide a framework to supply content to **everybody**, regardless of disability. This means that the same content source must be able to supply information to people with visual impairments or any disability for that matter. The content can be transformed into VoiceXML [33] for people with visual impairments. A speech synthesis tool such as Natural Voices from AT&T [34] can transform the VoiceXML into an audio stream.

Another requirement of content presentation is that content formats that do not conform to the above must also be supported. This is required as some web

applications may adopt the YBox and want to take advantage of its security features and not Content presentation. Legacy content (e.g. HTML format) may also need to be hosted using the YBox.

3.1.2. Security Requirement

The security needs of the YBox are quite extensive. A user-based web application being deployed in conjunction with the YBox as a front-end, must be totally secure from hackers. The content in the web application is only accessible to valid users. The security controls must apply to static and dynamic content of all forms (XML, HTML, PDF, doc, images, JSPs, Servlets ...etc.).

The security mechanism in the YBox is to be user-based. The web application designer must define the definition of a user at design time. The designer has full control over how a new user is created and how the user is authenticated. This allows the web application to connect to existing authentication mechanisms (e.g. LDAP). The designer must also have control over what resources each user has access to. This is specified at deployment.

The YBox must also support **groups** of users. A group is made up of one or more users. Access to resources can also be controlled by the group the user belongs to.

Take the following example: A student is part of a class called EE553. All the class (and only the class) should have access to the class notes available on the web. Using the YBox, the web application designer needs to specify that only members of the group EE553 are allowed access to the class notes. The web application designer can specify that the course lecturer also has access to this resource.

3.1.3. Form Validation Requirement

The YBox is required to validate user input from web forms. This validation is going to take away the responsibility of form checking from the web application designer. The YBox must achieve complete separation of presentation logic from business logic. Checks on user input are not contained in the same files as the presentation code. All of the above must apply to static and dynamic content.

3.1.4. Session Management Requirement

The YBox must be able deal with session storage independent of the Servlet Container used. This is not possible with the current implementation of the Servlet 2.3 API (as discussed in Chapter 2).

Session management in the YBox must be a flexible implementation giving control to the web application designer. The designer must be able to choose how he/she wishes to store the session (flat file or database). The implementation must enable the previous session to be restored if the user re-visits the web application again. This is required because the user may not have fully completed a task (e.g. purchase an item), and when they return, they should have the option to continue from where they left off. Using session management in the YBox it must be possible to access the web application from a PC, store the session state by logging out, and continue using the web application from a PDA and restore the saved state.

3.2. Analysis of the YBox Design

Now that the requirements of the YBox have been discussed, the design of the YBox can be analysed in detail. The core of the YBox revolves around the configuration file. The configuration file combines the four major features of the YBox. It contains information on content, security, form validation and session management. Each web application has its own associated configuration file.

The YBox loads the configuration file during initialisation. At this time some checks have to be performed on the structure of the file, to ensure it conforms to a set of rules so that it can be interpreted correctly. If the configuration file is not correctly structured then the web application associated with this configuration file will not accept any requests. The configuration file remains the source of most decisions for the duration of the life of the YBox.

The analysis of the YBox is examined in four sub blocks; content presentation, security, form validation and session management.

3.2.1. Analysis of Content Presentation

Content presentation in the YBox is based on XML content and XSLT to transform this content. All the source content for a web application must be written in XML for it to take full advantage of the YBox capabilities. If the source content is not written in XML, then the web application can only take advantage of the security and session management features of the YBox and not content presentation and form validation.

Firstly, the HTTP model has to be broken up into a request and response. The request is received by the Servlet Container from the client and the response is transmitted back to the client by the Servlet Container. For the content source to be client independent, the YBox has to process the request and the response.

Figure 3.2 shows how the YBox uses the request and response.

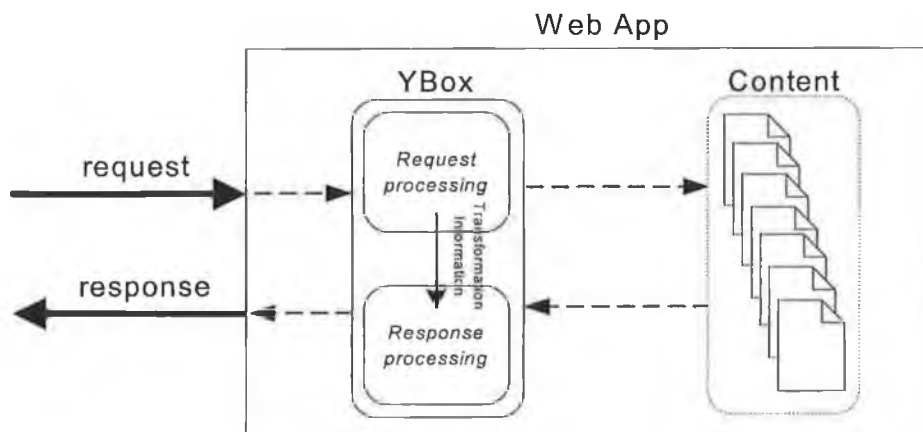


Figure 3.2. YBox dealing with Request and Response

Processing the Request URL

The YBox processes the request received from the client. It first examines the requested URL. There are two reasons for doing this:

1. Existing legacy content must be supported; therefore no transformation is performed on the content.
2. Special transformations that allow the YBox to produce content that is not based on the connecting device.

To understand these two points fully, it is best to take an example for each. For point one above, if the content is already written in HTML then processing on the response cannot take place. The HTML must be sent directly to the client whether it supports it or not. Part of the configuration file is dedicated to the file types that will not be transformed by the YBox. Figure 3.3 shows a client requesting legacy HTML content and bypassing the Response processing.

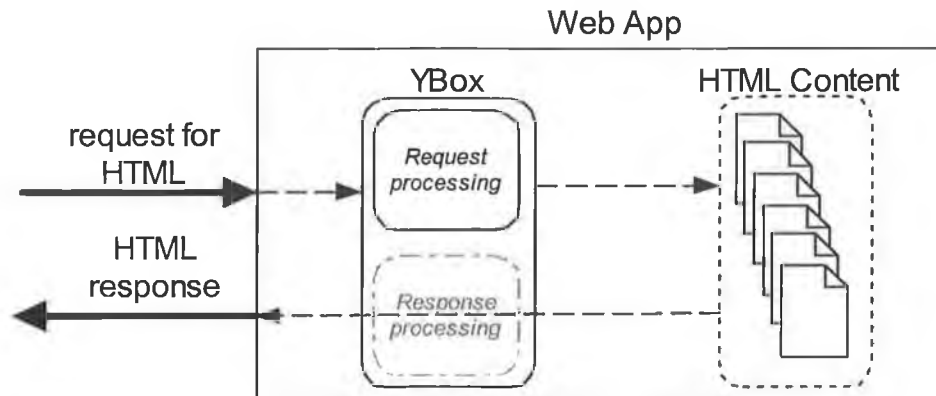


Figure 3.3. Client accessing legacy HTML content

For point two, special file types may be defined in the configuration file that will not be transformed using the standard transformation. Take the example of the user who wants to get a printable version of the content. In this case, the user may request the content in PDF format. When the Request processor detects that the user has requested PDF (from the requested URLs file extension), the Request processor notifies the Response processor and informs it to process the response differently. In this case, the Response processor uses a Formatting Object Processor (FOP) [20] to process the response. This can be seen in Figure 3.4.

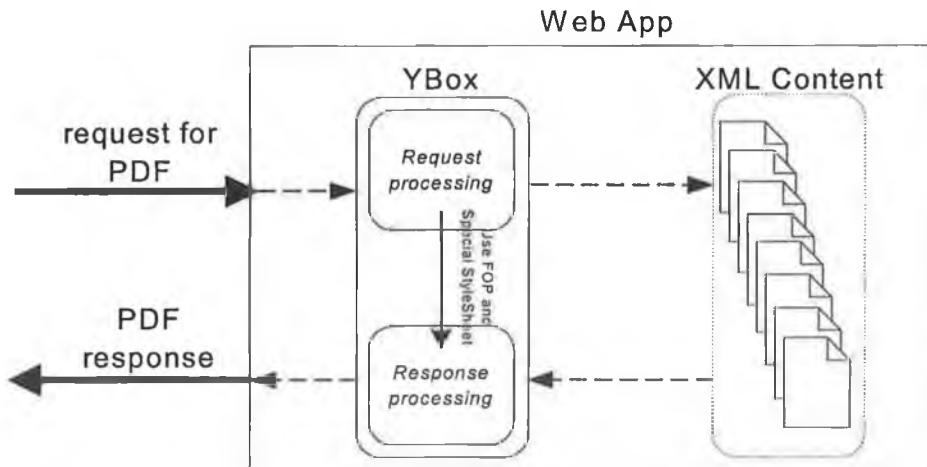


Figure 3.4. YBox producing content not based on connected device

Processing the Request HTTP Header

The YBox then processes the HTTP header in the request, as it needs to be able to recognise the type of client that is attached. This information can be extracted from the HTTP header (see Figure 3.5) sent with every request. This is called the **user agent**. An example of a user agent is:

Mozilla/4.0 (compatible; MSIE 6.0; Windows 98)

This is the user agent for Microsoft Internet Explorer 6.0 running on Windows 98.

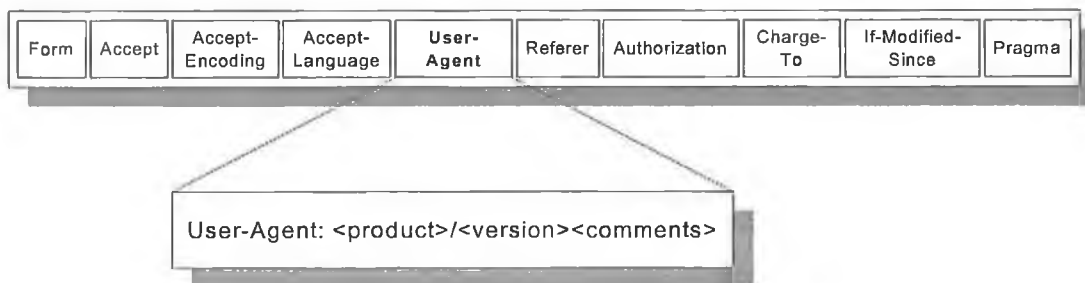


Figure 3.5. HTTP Header with User-Agent

After the user agent is known, a search of the configuration file is performed for a user agent match. If the user agent matches successfully, a set of transformation rules (specified by the configuration file) are loaded. If there is no match, then the default transformation rules are loaded. These rules should be as flexible and browser neutral as possible. It is recommended that these rules transform the XML into XHTML (well formed HTML).

The content source is written in XML. The content transformation rules are written in XSL. The XSL processor used in the YBox is Xalan from Apache [21]. Both the XML and XSL files are given to Xalan and the transformation rules are applied (see Figure 3.6). The resulting output from Xalan is sent back to the client (assuming the XML was transformed without any errors).

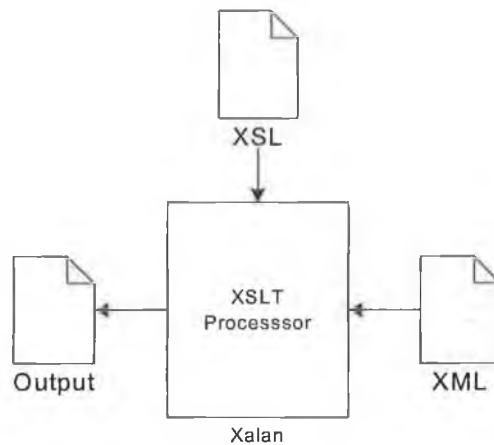


Figure 3.6. XSL Processor transforming XML using an XSL

3.2.2. Analysis of Security in the YBox

There are two central aspects to security in the YBox:

1. The Configuration file.
2. The YBox user.

The YBox cannot implement security correctly unless the web application designer includes these two requirements.

The Configuration File

The Security features rely heavily on the YBox configuration file and how the web application designer configures it at the deployment stage. The security features are not a design time feature, but rather a deployment feature. This means that the web application designer does not have to make any special changes to the source content for its access restrictions to be controlled.

The advantage of this is the web application designer can change the security features at deployment and this does not result in modifying any source content. In the case of dynamic content, no changes need to be made to the Java that generates the content.

To understand this feature, take the example given in section 3.1.2 again, where a group of students are part of the class EE553. If a new student joins the class mid way through the term, it is easy for the web application designer to add their name to the class list (using existing security such as LDAP). Hence, the new student has access to all the class notes without the web application designer modifying a single line of web application code.

The configuration file is written in XML and is validated against a Document Type Definition (DTD) during YBox initialisation. This configuration file has nested elements that contain information on all resources protected by the YBox.

The concept behind the protection system used in the YBox is similar to the Unix file system. In a Unix system each individual has a user name. Each individual is also part of a group (or groups) of users. Every file on a Unix system is protected by this mechanism; it is best to use an example to understand how.

A file named *index.html* exists on a Unix file system. Figure 3.7 shows what is returned when the command

```
ls -l index.html
```

is run on the file system. We are not interested in the file type, the write permissions or the execute permissions in a web application. We are only concerned with the read permissions (i.e. the user only ever wants to view the content).

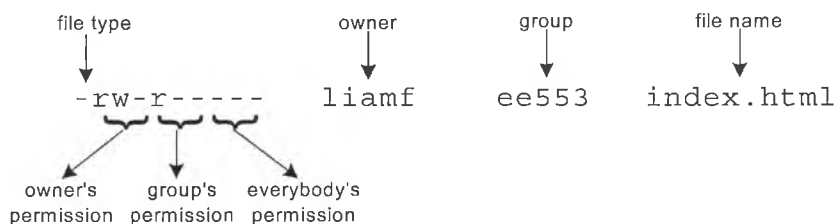


Figure 3.7. File permissions on a Unix File System

For this particular file, the owner (*liamf*) has read and write permissions but not execute permissions. Everybody in the group *ee553* has read only permissions. Everybody else (other users who are not *liamf* and not in the group *ee553*) cannot

view, modify or execute the file. This security system can also be enforced over complete directories.

The design of the YBox uses a novel way of implementing a security system on web applications similar to a Unix file system. The web application designer controls access to resources using the configuration file. They can control single resources (i.e. one XML file or one JSP file) or they can control groups of resources (whole directories in the web application).

The web application designer must also specify a login page that the user is redirected to if they do not have the required permissions to view a resource. This is specified in the configuration file. All users must have read access to the login page.

By extending the example of the EE553 Class, it can be demonstrated how the YBox controls access to particular resources inside a web application.

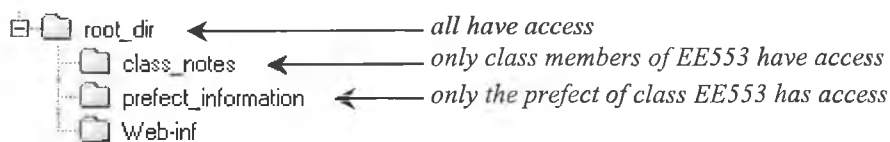


Figure 3.8. Directory Structure of sample web application

The directory structure for the class EE553 can be seen from Figure 3.8. All users have access to the resources contained within the *root_dir* directory. Only members of the class are allowed access the resources contained in the *class_notes* directory. Only the class prefect is allowed to access the resources in the *prefect_information* directory. Typically, the login page will be located in the *root_dir* directory as all have access to it.

Take the following scenarios:

1. What happens if a user who has not logged in tries to access a resource in the *class_notes* or *prefect_information* directory? The user gets redirected to the login page. If login is successful they are allowed access to the resource.
2. What happens if a user who has successfully logged on as a member of EE553 and is not the prefect, tries to access a resource in the *prefect_information*

directory? The user gets told he/she does not have access to the requested resource and gets redirected to the login page.

3. What happens if the prefect tries to access a resource in the *class_notes* directory? The user, who is the prefect in this case, is allowed access to the resource, as he/she is also a member of the EE553 class.

The YBox User

The other core aspect to the security implementation in the YBox is the concept of a YBox user. Because the YBox is designed to aid the development of user-based web applications, this is an important aspect of the design.

This feature tells the YBox exactly **who** is logged on. This information is stored in the users session and is kept there and examined during each request. Not only does it tell the YBox who is making the request, but it also tells the YBox what group(s) the user belongs to.

Figure 3.9 shows a UML class diagram of the YBoxUser. From this class diagram it can be seen that this class is abstract (Note: This is not the full class diagram. This topic is discussed in more detail in section 3.2.4). The abstract class has two private attributes; `userName`, which is a string, and `userGroups` which is an array of Strings (a user can be a member of more than one group). It also has two public abstract methods, `getUserName` and `getUserGroups`. The web application designer must implement both of these methods.

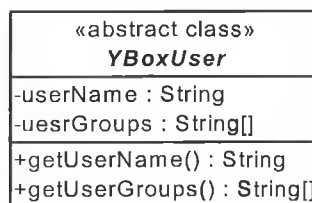


Figure 3.9. The YBoxUser Abstract Class

To use the security features in the YBox, the web application designer must extend the YBoxUser abstract class. In doing so, they must also implement all abstract methods. This means that the authorization is left to the web application designer to

implement. Therefore, they could connect to existing authorisation systems (e.g. LDAP) to secure a web application.

The YBoxUser **object** is stored in the users session. This means the YBoxUser object is accessible to the YBox and to the web application designer at run time. The first time the user logs on to the web application, a new instance of the YBoxUser is created and placed in the users Session. If the YBoxUser object does not exist in the session, then the YBox assumes the user has not logged in. The user will only have access to resources that *everybody* has access to (information from the configuration file).

3.2.3. Analysis of Form Validation

The purpose of form validation is to ensure the requested resource never gets invoked if the user incorrectly fills out a web form. This means that validation logic is not required in the presentation layer. When using the form validation feature of the YBox, the resource does not require code that performs validation mixed through code that performs presentation. All validation is server side as the YBox does not support client side form validation.

Form validation is only possible in the YBox when the web application designer writes the content in XML and also conforms to some YBox specific rules about form generation using XML. These rules are discussed in more detail in Chapter 4. Form Validation is **not** possible when the designer uses HTML, WML, XHTML or anything other than XML for the form design.

Form Validation is not applicable to static content as it is only concerned when the user requests a dynamic resource (Servlet or JSP). The request may come from a static source. Figure 3.10 shows a typical example to help understand this concept.

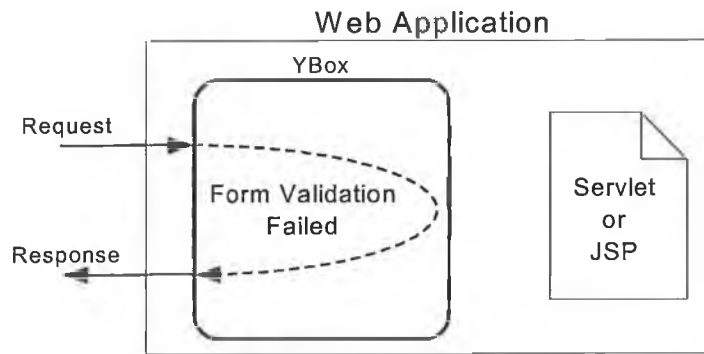


Figure 3.10. Form Validation Failed when requesting a Servlet/JSP

The only way form validation will work using the YBox is if the web application designer conforms to specific guidelines laid down when designing a form in XML. The reason these rules are required is so that the YBox can properly interpret the incoming form data and redisplay the page again (with errors messages) if necessary.

To understand the solution to this problem, see the Figure 3.11 and Figure 3.12. Figure 3.11 shows the user requesting *form.xml*. The YBox caches the XML file in the memory of the Servlet Container (actually as an object in the users session). The YBox does this so it can redisplay the same page again and again, until the user fills out the form correctly.

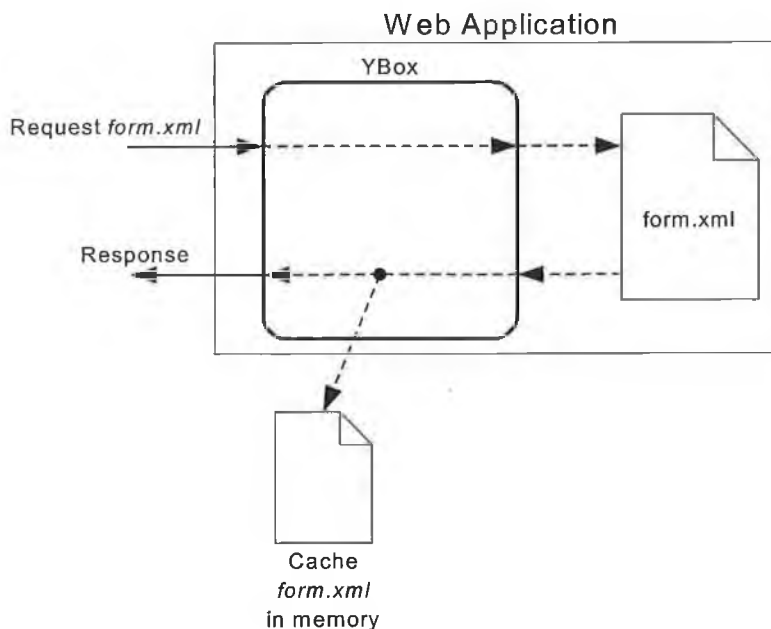


Figure 3.11. YBox caches the users requested form

The YBox examines the submitted data and decides if it is valid or not based on the validation rules. If the submitted data is valid, the YBox lets the user proceed – otherwise the YBox displays the same page again with error messages. For the purpose of this example, the user incorrectly fills out the form and submits it to the Servlet Container.

When the data that is submitted is invalid, the YBox gets the cached version of *form.xml* from the session. It scans through the file and expands all errors messages specified by the web application designer. The XML file with error messages is sent to the transformation stage and the result is sent back to the user. This is shown in Figure 3.12. The process is repeated until the user correctly fills out the form and is then allowed access to the Servlet/JSP.

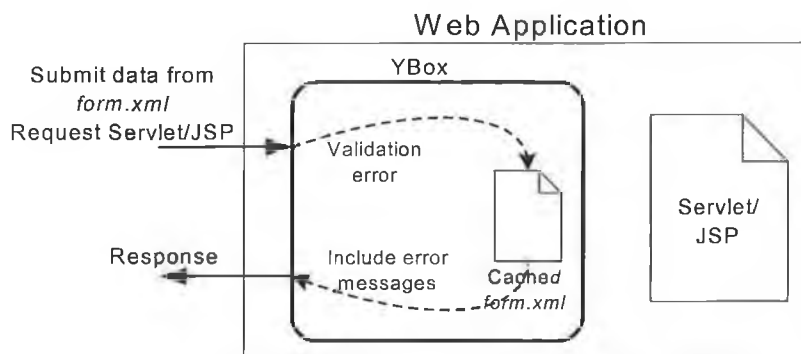


Figure 3.12. YBox sends back cached form to user (with error messages)

The web application designer must specify all the information required by the YBox to validate a form at design time. This means the form description in XML must have tags for input field, error messages, data types, and the resource being requested (Servlet/JSP). Using the current Servlet API (version 2.3) and HTML 4.0, all fields received from a web form will be a string (`java.lang.String`) object.

Using the YBox it is possible to specify the following Java data types for validation:

- String
- int
- double
- Any custom class

It is also possible to specify whether the field is **required** or not. This means that the field need not be filled out by the user and the form will still validate correctly. The YBox attempts to cast the input fields into the required types. If it succeeds, then the user is shown the requested Servlet/JSP.

Version 2.3 of the Servlet API also directs all requests to the `service` method of the Servlet. The footprint for this method looks like:

```
public void service(HttpServletRequest req,  
    HttpServletResponse res);
```

Using the YBox, each “Submit” button on a web form is mapped directly to a method of the Servlet. Therefore, if the user clicks a “Submit” button on a web form, the YBox calls the appropriate method of the Servlet, specified by the underlying XML.

Figure 3.13 shows a HTML form viewed in Internet Explorer. When the user clicks “Submit” the browser sends the form information to the Servlet Container. The YBox validates the information entered. The “Username” and “Password” are both required. From Figure 3.13 is can be seen that both of these fields contain information, and therefore validates correctly. The “Submit” button is mapped to the `login` method of the requested Servlet and input parameters to this method are the validated fields of the form.

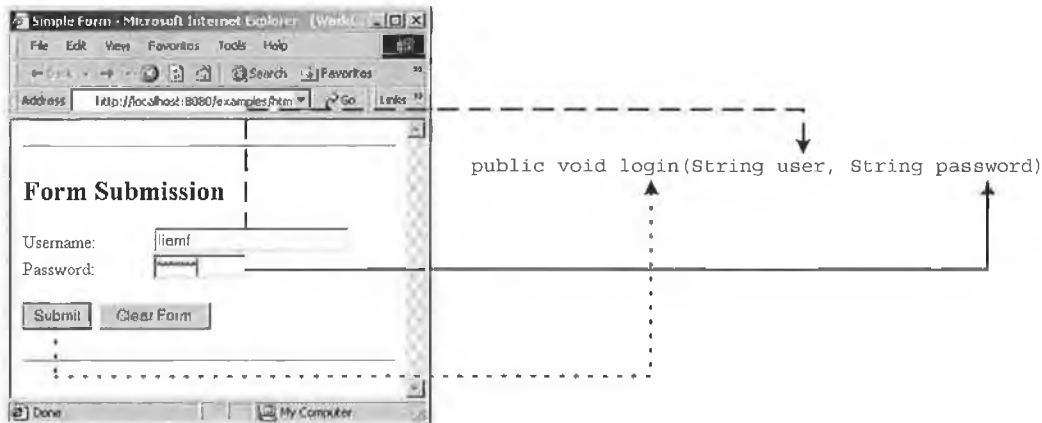


Figure 3.13. “Submit” button mapped to a method in a Servlet

The method footprint actually includes a request and response object also as the web application designer will definitely need access to these (to write the output to the `HttpServletResponse` object). Below is the actual method footprint.

```
public void login(HttpServletRequest req,
    HttpServletResponse res, String user, String
    password)
```

As mentioned previously, the YBox also supports custom classes. This means the web application designer can write his/her own classes to validate the input fields of a web form. The feature enables the designer to group one or more input fields together, and validate them as one object. This allows code reuse across a web application design, as the same validation object may be used by several developers for several different web forms.

Take the example of a form that requires the user to enter their date of birth. The form has three input fields. The first is an integer that represents the day of birth. The second is another integer that represents the month of birth, and the third is another integer that represents the year of birth.

The web application designer can specify in the source XML that he/she wants to validate all three input fields as follows:

- Firstly, each input field is validated on their own. They are all required, therefore should not be blank. They are all integers, therefore they should cast into an `Integer` (`java.lang.Integer`) object.
- Secondly, after the first validation has taken place, they are grouped together and validated as a single, user defined object. In this case the class is called “`UserDate`”.

For this type of validation to work within the YBox, the object must have a valid constructor that has a similar format to the `UserDate` constructor:

```
public UserDate(int day, int month, int year)
```

The YBox will try to create an instance of `UserDate` and return it to the called Servlet method.

There are several combinations of results that can occur (day valid, month invalid, year invalid; day invalid, month valid, year valid...etc). The YBox is only concerned with two outcomes:

1. The object was instantiated correctly.
2. The object was instantiated incorrectly.

If the object was instantiated correctly (i.e. the date entered is valid), then there are no complicated issues – the `UserDate` object gets passed to the requesting method and the response is given. See Figure 3.14 for an example of this.

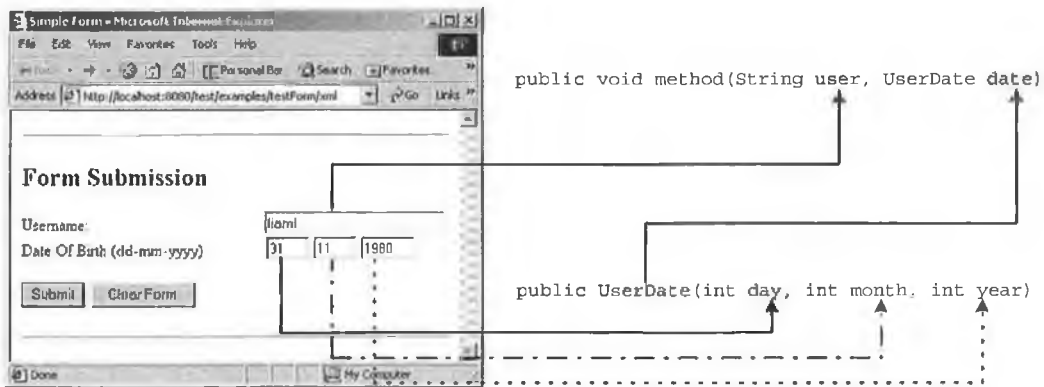


Figure 3.14. Object Validation in the YBox

If the date entered is invalid (e.g. 31-11-1980), then the YBox has to report an error message to the user that the day entered is invalid (November only has 30 days). In this case the `UserDate` object must tell the YBox which parameter is incorrect and why. The YBox can then return the form to the user with this error message.

3.2.4. Analysis of Session Management

Session management in the YBox must allow the web application designer to save session state if the Servlet Container is removing the session information from memory. Session management is based on the abstract class, the `YBoxUser` (discussed in section 3.2.2). This class is developed further in this section as more methods are added to it to implement session management.

The web application designer must code the YBoxUser class to allow it to store session information to a persistent storage device. Each YBoxUser object is responsible for saving and restoring all session variables if the session is invalidated. The YBox informs the instance of the YBoxUser object when the session is invalidated, by calling a specific method and passing it the session variables.

These new methods in the YBoxUser class can be seen in Figure 3.15. saveSession() and restoreSession() are the methods that need to be added to the YBoxUser class to enable the YBox to support persistent session storage.

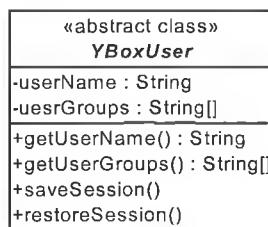


Figure 3.15. Class Diagram of the updated YBoxUser

The saveSession() method is passed a Hashtable of name value pairs (session variable name, session variable value) from the YBox, when it detects a session has expired or when a session is deliberately invalidated. The web application designer implements the saveSession() method, so he/she can store the session variables in a flat file or database.

The restoreSession() method is not actually called by the YBox ever. It is only present for the benefit of the web application designer. The web application designer should use this method to extract the data associated with the user session that is stored in a file/database, and copy it back into the users session when the user returns to the web application.

Take the example of the class EE553 again. The typical student uses the web application as follows:

1. He/she gets a username and password from the administrator.

2. He/she logs into the web application for the first time. At this point, the web application may call `restoreSession()` of the `YBoxUser` but there will be no data to restore.
3. The user logs out, and the Servlet Container invalidates the session. This causes the YBox to call the `saveSession()` method of the `YBoxUser` object (the Servlet Container also invalidates the session if the session expires).
4. The `saveSession()` method writes the session information to a database or file.
5. The next time the user logs into the web application, the `restoreSession()` method of the `YBoxUser` is called. This time there is data to be restored.

There is no configuration required in the YBox configuration file to enable session persistence. If the `YBoxUser` object is present, then it must have the required methods. These methods will get called, even if the web application designer has not implemented any storage mechanism in these methods.

3.3. Summary

This chapter examined each requirement in detail and the architecture of the YBox was described relative to these requirements. It has stated the problems associated with each requirement. The way the YBox should behave in given circumstances is described with the aid of examples.

The next chapter describes the actual implementation of the YBox. This includes the core technologies used and why they are required for the YBox framework. It also discusses the problems encountered during the implementation phase and how these problems were overcome.

Chapter 4 - Implementation of the YBox

The YBox is implemented completely using Java and XML. As mentioned previously, when work began on the YBox the version of the Java Servlet API was 2.2. Also, XML technologies were rapidly maturing as XML became accepted by Java developers. As a result, the implementation of the YBox changed as the technologies used became more developed. Often the implementation had to be completely changed as a result of a change/improvement to one of the core technologies used. In all cases the implementation used is discussed, and any technology changes/improvements are identified. The impact these changes had on the YBox is discussed in detail.

This chapter follows the same format as the previous chapter. It is broken up into the same sub-sections:

- Content Presentation.
- Security.
- Form Validation.
- Session Management.

This chapter also has an additional sub-section on the configuration file. This is the first topic covered, as it is the hub of the design. Three out of the four core functions of the YBox could not function without it (Session Management does not need the configuration file).

4.1. The Configuration File

The configuration file is used at the initialisation stage of the YBox only. It is loaded by the init method of the YBox Filter, the only entry point in the YBox. The Filter interface was released with version 2.3 of the Servlet API. When using the Servlet 2.2 API, a Servlet was used (instead of a Filter), but for the discussion of the configuration file, there is no difference between the two initialisation methods. The

Servlet Container invokes the initialisation method when the web context associated with the YBox (for a Filter or a Servlet) is loaded.

4.1.1. Accessing the Configuration File

The YBox has to be able to get access to the configuration file so that web application specific configuration can take place. It only needs to access the configuration file during the initialisation of the web application. After initialisation, the YBox has extracted all the information it requires. Therefore, if the configuration file is modified while the web application is running, the modifications are only reflected on the next reload of the web application or restart of the Servlet Container.

The most important aspect to accessing the configuration file is all references to the file are in a platform independent manner. Take the example of a file called *config.xml* on UNIX and on Microsoft Windows Operating Systems. Under UNIX, the file may be located at:

/usr/local/tomcat/webapps/test-app/config/config.xml

The equivalent file on a Windows Operating System may be located at:

C:\tomcat\webapps\test-app\config\config.xml

Therefore the configuration files location must be **relative** to the directory the web application is located in. In the above example, the web application is called *test-app*. The path to the configuration file then becomes:

config/config.xml

This relative path is valid for a UNIX or a Windows Operating System. The YBox can now access the configuration file in a platform independent manner.

The next aspect to take into consideration is how the Servlet Container tells the YBox to use the configuration file to initialise itself. To understand this, the lifecycle of a Filter needs to be examined. The Servlet Container that the Filter is deployed in controls its lifecycle. The lifecycle consists of the following steps (this procedure is identical for Servlets):

- The Filter class is loaded when the web application is started.

- Creates an instance of the Filter class.
- Initialise the Filter by calling the `init` method of the Filter and passing it a `FilterConfig` [22] object.
- Invoke the `doFilter()` method of the Filter and pass it a `ServletRequest`, `ServletResponse` and `FilterChain` object.
- Call the `destroy()` method of the Filter if the Servlet Container is being shut down or the web application associated with the Filter is being removed.

Now that the lifecycle of a Filter has been explained, it can be seen where the YBox is initialised – in the `init()` method of the Filter. This is also where the Servlet Container passes the location of the configuration file to the YBox. It is contained within a `FilterConfig` object that is passed to the `init()` method.

The `FilterConfig` object gets the location of the configuration file from *web.xml*, the deployment descriptor associated with every web application. Below is an extract from *web.xml* that shows the Filter definition and the initialisation parameter associated with it. This must be specified in every web application descriptor to allow the YBox to be initialised correctly.

```
<filter>
    <filter-name>YBoxFilter</filter-name>
    <filter-class>ie.dcu.liamf.ybox.YBox</filter-class>
    <init-param>
        <param-name>YBoxConfig</param-name>
        <param-value>config/ybox-config.xml</param-value>
    </init-param>
</filter>
```

The `filter-name` tag contains the name the Filter is referred to for its entire lifecycle. The `filter-class` tag contains the actual name of the class file that contains the Filter. As can be seen, the Filter is located in the `ie.dcu.liamf.ybox` package.

The `init-param` is an optional tag in *web.xml*. If it is present, the contents of the `param-name` and `param-value` tags are loaded as a name-value pair in the

`FilterConfig` object. In the above extract, the `parm-name` is `YBoxConfig`. The `parm-value` represents the location of the configuration file. In the above example, the location is `config/ybox-config.xml`. The configuration file does not need to be named the same for every web application – the file name is dynamically accessed through the `parm-value`.

Every web application that uses the YBox must have an `init-parm` with a `parm-name` called `YBoxConfig`. This is the only method of telling the YBox the location of the configuration file.

At this point, the `FilterConfig` object has the information about the location of the configuration file. The YBox gets this information from the `FilterConfig` object by calling one of its methods:

```
getInitParameter("YBoxConfig")
```

The String that is returned is the location of the configuration file. If the file exists, the YBox continues to load the file, otherwise it throws an Exception and the web application is not loaded.

4.1.2. The Structure of the Configuration File

The structure of the configuration file is important because it defines the layout of the configuration file and how each aspect of the file is related. Since it is an XML file, the file must be structured correctly. If the file is not structured correctly, then the web application associated with the YBox is not loaded and hence is not accessible.

Before discussing the configuration file any further, two important concepts of XML need to be understood [23]:

1. A **well-formed** XML document.
2. A **valid** XML document.

A **well-formed** XML document is one where the mark-up contained within the document is legal. This means:

- All opening tags must have closing tags.
- An equals sign followed by a quoted value follows every attribute.

- There is only one top-level (root) element.

This sounds like a simple concept, but HTML fails to implement these simple rules. Many of the HTML pages on the web do not conform to the simplest rules in the HTML specification. This makes processing more difficult for web browsers. Because some browsers display ill-formed HTML, people continue to produce pages that do not conform to the HTML specification.

All XML processors (SAX or DOM) and the Java Architecture for XML Binding (JAXB) [27] do not load any XML document that is not well-formed. This implies the first requirement on the structure of the configuration file is that it must be well-formed.

A **valid** XML document is one where the contents of the document can be validated against an internal or external DTD or schema. The rules of the XML specification are not broken if a document is well-formed **and** invalid. The configuration file for the YBox must be valid and well-formed for the YBox to load it properly. This is discussed further in section 4.1.3.

There are separate sections for the major components to the YBox as can be seen in Figure 4.1. The only section that does not have a role in the configuration file is Session Management as this is implemented in the `YBoxUser` object. The tree structure shown in Figure 4.1 displays how the configuration file is structured. The `ybox-config` element is the root element. The branches underneath the `ybox-config` element form the major aspects of the YBox configuration.

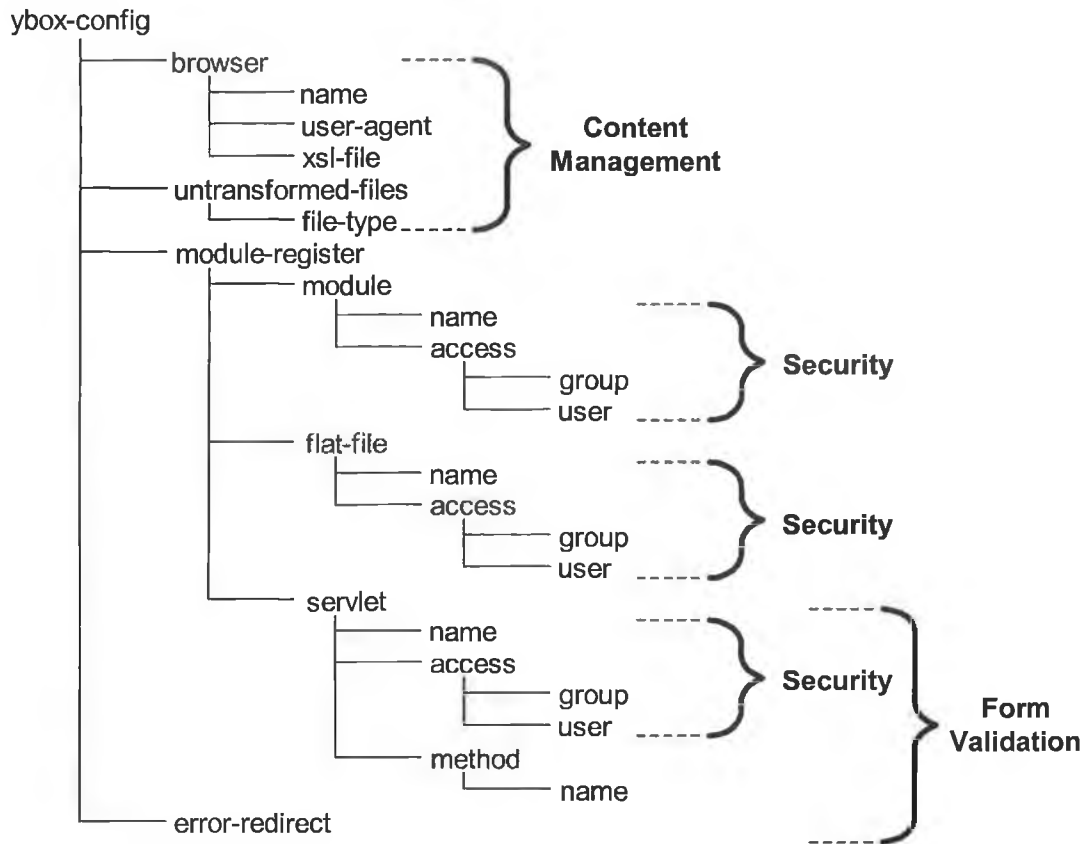


Figure 4.1. The tree structure of the Configuration file

4.1.3. Loading the Configuration File

During the initial design work on the YBox, XML processors in Java were just being released and bugs being ironed out. The JAXB specification was still being finalised by Sun Microsystems through the Java Community Process and was not even usable as an early access release⁴.

Therefore, when the design of the YBox began, there was only one choice in the loading of the configuration file; an XML processor had to be used. Later, when JAXB was released, the loading of the configuration file was updated to use the powerful features of JAXB.

⁴ Early Access Release is where users can download an API from Sun Microsystems that has not been fully completed and tested. It is the stage before the Beta release of the API.

Using an XML processor to load the Configuration File

Xerces-2 [24] from Apache⁵ is the XML processor the YBox uses because the source is freely available and the processor complies with the World Wide Web Consortium (W3C) recommendations for XML 1.0. It supports both DOM 2.0 (Document Object Model) and SAX 2.0 (Simple API for XML) processing, but DOM is used for loading the configuration file in the YBox.

DOM is a set of API's that provide methods of placing the entire XML file into an object in memory, called a DOM document. It is then possible to traverse all nodes and elements within the DOM document quickly and read/modify them as desired. As can be seen from Figure 4.2, Xerces loads the configuration file from the hard disk into memory. The YBox can then traverse the DOM document and extract the configuration information.

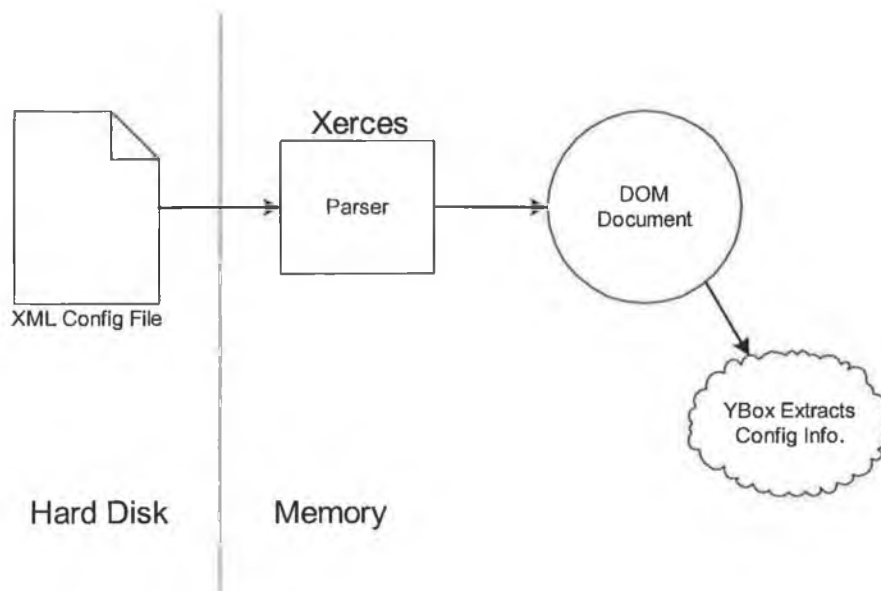


Figure 4.2. Xerces loading the XML Configuration File into memory

Xerces also ensures the document is well-formed and valid. If the XML file is invalid or ill-formed, it throws an exception. The YBox catches the exception and reports it to the user by logging the exception stack trace to the log file associated with the web

⁵ Xerces-2 is the latest XML processor from Apache. Xerces-1 also exists on the Apache XML website, but this is an older implementation. Xerces-2 is referred to as "Xerces" for the remainder of this document.

application. This exception may explain that a node is ill-formed or the configuration file did not validate against a DTD correctly.

Using JAXB to load the Configuration File

JAXB is a design time tool for two-way mapping between XML documents and Java objects and vice versa. It has design-time libraries and run-time libraries. At design time, the JAXB compiler loads a schema, or in the case of the YBox, a DTD. There is also a **binding schema** input file to the JAXB compiler. This contains the binding rules and any additional information the JAXB compiler needs. The JAXB compiler processes the DTD and outputs Java source file that represent the XML.

It is important to understand that the JAXB compiler only operates during the design phase of the YBox. The resulting Java source files are imported into the overall project. If the needs of the configuration file change, the DTD will change. This means the JAXB compiler is called to recompile the DTD and generate the new Java source files. The process can be seen in Figure 4.3.

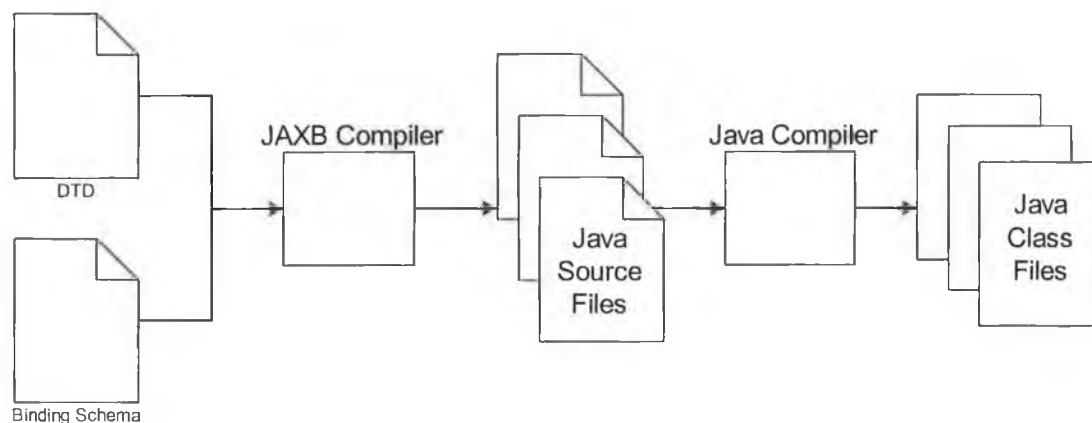


Figure 4.3. JAXB Compiler generating Java source files

The Java source files generated during the JAXB compile stage, implement and extend the classes in the runtime API of JAXB (the **binding framework**). This means the JAXB runtime JAR⁶ file has to be in the classpath of the web application. The binding framework has three primary functions:

1. **Unmarshal**: This is the process of converting a flat XML file into Java objects in memory.

⁶ Java Archive (JAR) files are used to bundle all files needed to run an application into one archive file.

2. **Validation:** Verifies that the Java object representation conforms to the rules specified in the DTD (equivalent to well-formed and valid).
3. **Marshalling:** Producing an XML document from Java objects.

The YBox is only concerned with the first two. It will never be producing an XML document from Java objects.

To appreciate the power of JAXB it is best to take an example from the YBox configuration file. Take the security control on every directory, file and Servlet. The *access* tag in the configuration file controls this security and has the following structure:

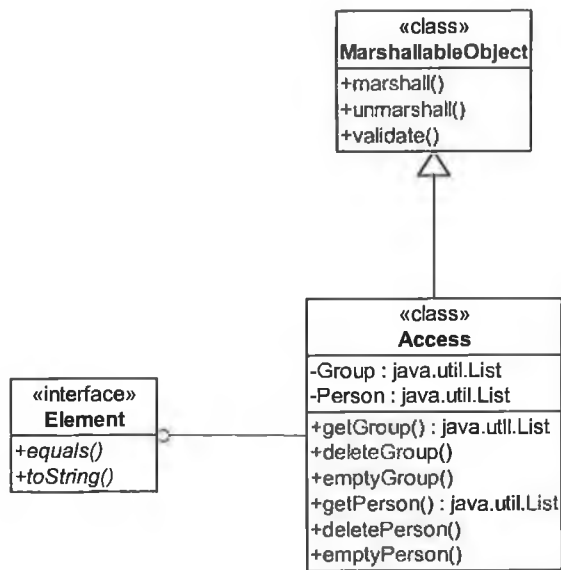
```
<access>
  <group name="ee553"/>
  <group name="all"/>
  <person name="liamf"/>
</access>
```

The DTD for the above has the following structure:

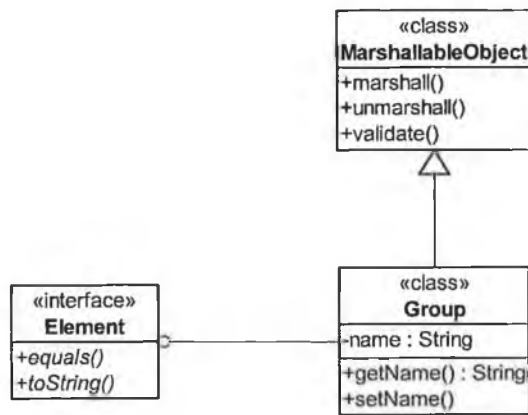
```
<!ELEMENT access ( group*, person* )>
<!ELEMENT group ( #PCDATA )>
<!ATTLIST group
  name CDATA #REQUIRED >
<!ELEMENT person ( #PCDATA )>
<!ATTLIST person
  name CDATA #REQUIRED >
```

Examining the DTD further: The first *ELEMENT* is the *access* tag. It contains zero or more *group* tags, and zero or more *person* tags. The next *ELEMENT* is the *group*. It has one attribute (*ATTLIST*), *name* that is required. The *person* element is identical. During the unmarshalling, the Java class files that load and validate the XML document are based on the DTD that generated the classes.

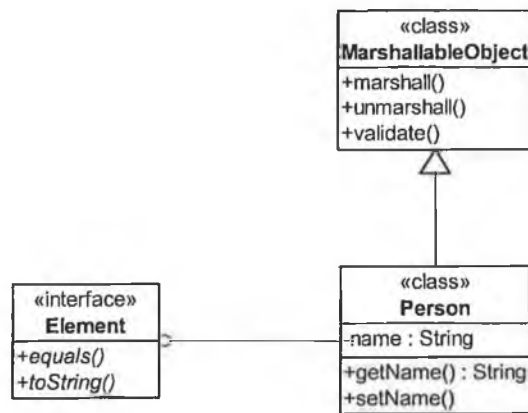
The resulting Java source code has a UML structure shown in Figure 4.4.



(a)



(b)



(c)

Figure 4.4. UML representation of (a) the Access class, (b) the Group class and (c) the Person class

The `getGroup` method of the `Access` class returns a `List` of `Group` objects. Each of these `Group` objects has a `getName` method. The same applies for the `Person` objects. The YBox uses the `getName` method to extract the name of the groups that are allowed access the resource. This is how the configuration information is extracted from the Java objects that are unmarshalled from the XML configuration file.

The same principal applies to the rest of the configuration file.

4.1.4. Storing the Configuration Information

Now that the configuration file has been successfully loaded using the `Servlet Container`, the configuration information needs to be shared with the rest of the web application. The configuration information must reside in a location where it is accessible to the complete web application.

As all `Servlet Containers` are multi-threaded, the issues that arise with different threads accessing the configuration information at the same time must also be investigated. Two threads cannot be modifying the configuration information at the same time.

Using the Scope Objects

Components within a single web application can share information by using the four scope objects [25]. These can be seen in Table 4.1. The information is stored as attributes in the scope objects. These attributes are accessed by using the `getAttribute` and `setAttribute` methods of the scope object.

Scope Object	Class	Accessible From
Web Context	<code>javax.Servlet.ServletContext</code>	Any web component within the web application
Session	<code>javax.Servlet.http.HttpSession</code>	The web component handling the request that belongs to the requesting

		session
Request	<code>javax.Servlet.HttpServletRequest</code>	Web components handling the request
Page	<code>javax.Servlet.jsp.PageContext</code>	The JSP page that creates/uses the object

Table 4.1. Scope Objects within a Web Application

The scope object used to store the configuration information in the YBox is the Web Context scope object. The reason is the Web Context scope object is accessible from every component (Servlet, JSP, Filter) within the web application. Therefore, the configuration information is a shared resource across the entire web application.

During the `init` method of the `YBoxFilter`, the configuration information is stored in the `ServletContext` scope object using its `setAttribute` method. This can be seen in Figure 4.5.

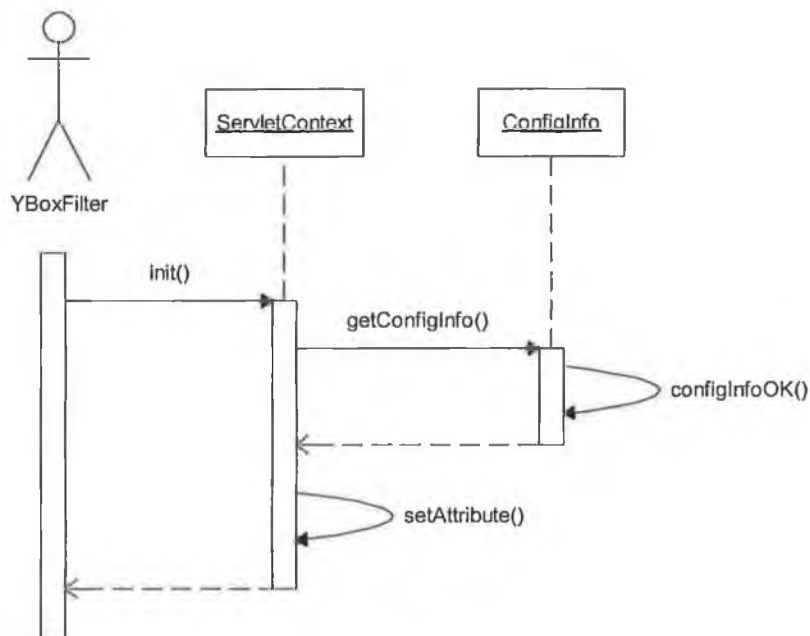


Figure 4.5. Sequence diagram for the `init` method of the `YBoxFilter`

For every request (the `doFilter` method of the `YBoxFilter` or `service` method of the `YBoxServlet`), the `getAttribute` method of the `ServletContext` is called. This returns the configuration objects needed for security, content management and form validation. This is shown in the sequence diagram in Figure 4.6.

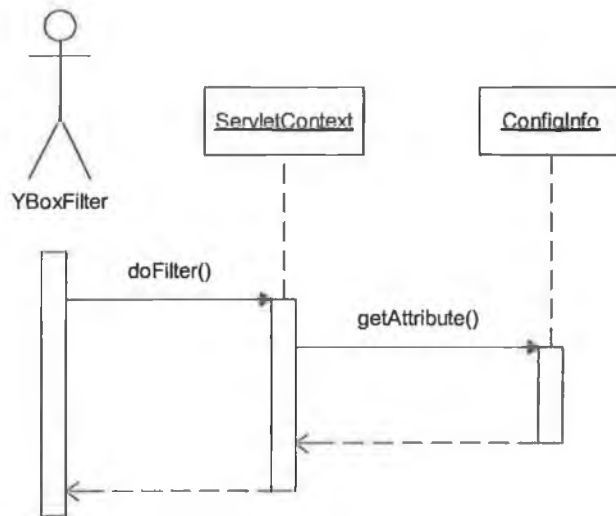


Figure 4.6. Sequence diagram for the doFilter method of the YBoxFilter

Concurrent access to the Configuration Information

The Servlet Container is multi-threaded because it must deal with several requests at the same time. Therefore, a number of web components could be accessing the configuration information simultaneously. As a result, the question arises: does access to the configuration object need to be synchronised?

The answer is no. The reason is the only web component that modifies the configuration object is the YBoxFilter. The YBoxFilter only does this during the `init` method. The Servlet Container is not processing any requests during the initialisation stage, so there is no way two threads can be trying to modify the configuration object at the same time.

4.2. Content Presentation

Dynamic content (Servlets and JSPs) is discussed with respect to two different APIs, firstly version 2.2 of the Servlet API and then version 2.3. The version of the API has very little impact on XML file transformation or the ability to support legacy HTML (or any format) content. Therefore, this section is the same for both APIs.

The common aspect between version 2.2 and 2.3 of the Servlet API is the XSLT processing that takes place on the server if required is identical. When the source content format is XML, the YBox transforms the XML, otherwise, the content is sent to the client unmodified.

4.2.1. Static Content - Flat Files

Flat files refer to files that are not executed on the server. Their content is static and does not change over time or depend on any external data. Examples of flat files include:

1. HTML documents.
2. PDF, Word Documents.
3. Images (jpeg and gif).
4. Java Applets.
5. XML documents.

Files that are not XML

Points one to four above do not need to be transformed. These file types are sent to the client unmodified. HTML is legacy content (not in XML format) and must be supported by the YBox. Images, Word Documents and Applets must also be supported. XML documents must be transformed based on the connecting client.

The YBox must process every request and decide if it has to transform the response or not. The Servlet Container directs all requests to the YBox because of the mapping section in the deployment descriptor (*web.xml*). For version 2.2 of the API the Servlet mapping has the structure:

```
<servlet-mapping>
  <servlet-name>YBoxServlet</servlet-name>
  <url-pattern>/*</url-pattern>
</Servlet-mapping>
```

The extract from *web.xml* shows the mapping of all requests (*/** in the *url-pattern*) to the YBoxServlet.

For version 2.3 of the API the Filter mapping has a similar structure to the Servlet mapping:

```
<filter-mapping>
  <filter-name>YBoxFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

With version 2.3 of the Servlet API, the YBox is based on a Filter and not a Servlet as in version 2.2.

The YBox must then decide if it is to transform the requested resource or not. The YBox does this from information specified in the configuration file. By default, the YBox attempts to transform **everything** before sending the response back to the client, so this section of the configuration file is very important for web applications that have any content that is not in XML format.

The following XML is from the YBox configuration file. It specifies that the YBox is not to transform any files of type; html, jpg and gif. This can be expanded at deployment to cater for more file types if necessary.

```
<untransformed-files>
  <files type="html"/>
  <files type="jpg"/>
  <files type="gif"/>
</untransformed-files>
```

Figure 4.7 shows a client requesting a HTML page. This page loaded by the YBoxFilter and sent back to the client untransformed.

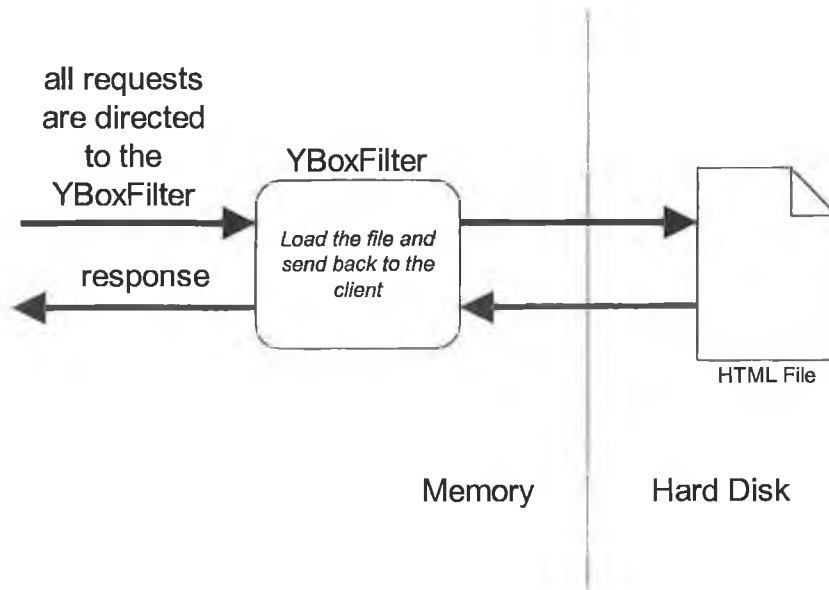


Figure 4.7. YBoxFilter dealing with a request for a HTML file

XML Files

XML needs to be transformed by an XSLT processor, Xalan in the case of the YBox, before the result is sent back to the client. This is also true for Servlets and JSPs, which are discussed in more detail later.

The YBox must have one or more XSL style sheets specified in the configuration file for it to function correctly. It needs to know what transformation rules to apply to the XML source. For the YBox to support multiple client types, it must also have an XSL style sheet for each supported client.

The YBox also supports special transformations for file types with a predefined file extension. To understand this concept, it is best to use an example. An extract from the configuration file is shown in Figure 4.8.

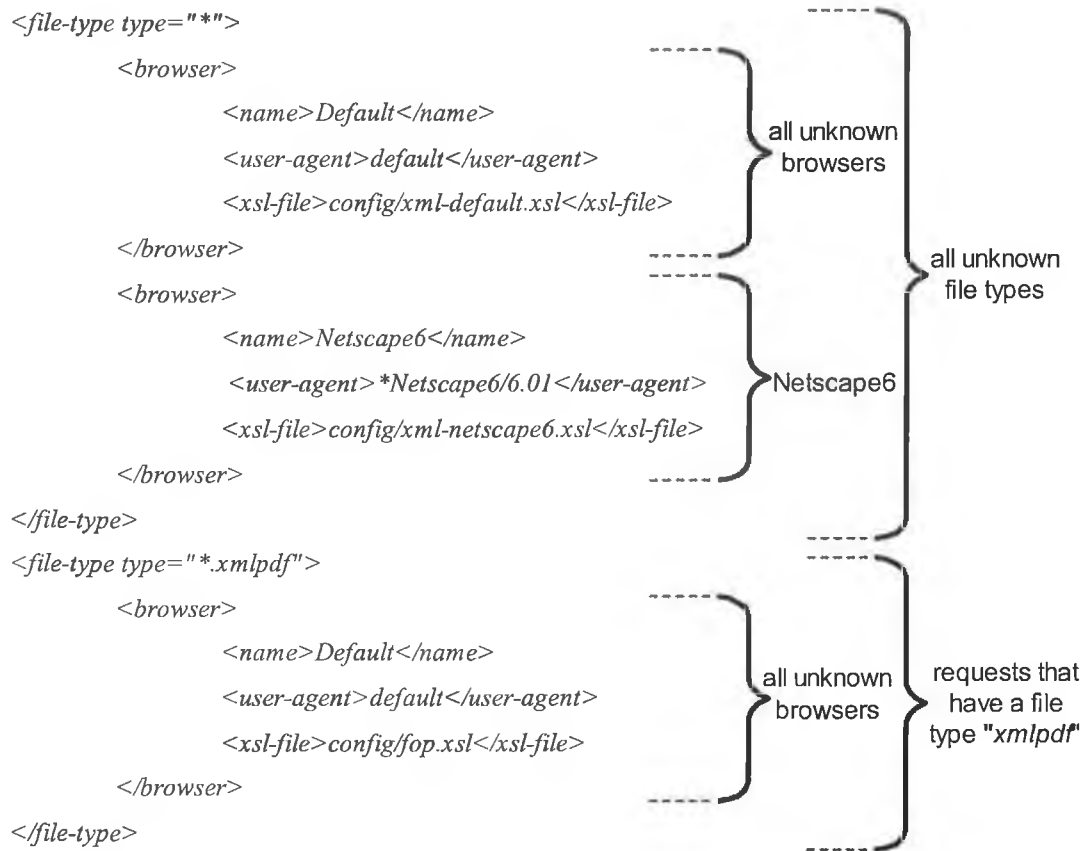


Figure 4.8. Configuration file showing the browser/XSL style sheet mapping

The first `file-type` in this example is `"*"`. This means that any unknown file types get their style sheets from this section of the configuration file. The second `file-type` is `"*.xmlpdf"`. Requests that have a file type that match this will use the `fop.xml` style sheet. `"*.xmlpdf"` is actually a special case. When the YBox detects a request for a file with an ending `".xmlpdf"` it uses FOP to transform the XML source into PDF.

Every file type must have a *default* user agent also. When the requesting client is unknown, the style sheet between the `xsl-file` tag is used for the XSL transformation. Otherwise a compare is performed on the user agent of the client to the user agents located in the configuration file. If a successful match is found, that XSL file is used for the transformation.

Lastly, the YBox can also validate all XML content against a DTD. This means the XML content is valid and well formed. The need for this feature arises when content is being generated by several designers. The common DTD ensures all designers

conform to common tag naming and structure. The `valid-dtd` tag in the configuration gives a path to this DTD.

4.2.2. Dynamic Content - Servlets and JSPs

The differences between version 2.2 and 2.3 of the Servlet API greatly affects the design of the dynamic content support in the YBox. It is possible to implement the YBox that was based on version 2.2 of the Servlet API in version 2.3 of the Servlet API, but not vice versa. This means that the YBox only runs under version 2.3 of the Servlet API and that the Servlet Container that contains the YBox must also support version 2.3 of the Servlet API.

Servlet API Version 2.2

Filters are not supported in version 2.2 of the Servlet API and as a result, the YBox was originally base on a Servlet. All requests are directed towards the `service` method of the `YBoxServlet`. This Servlet was used to call the requested resource, transform the output from the resource if required, and return the result to the client.

The single biggest problem with this architecture is the YBox does not have permission to modify the response from the Requested Servlet. This can be seen in Figure 4.9. The response gets written to the `PrintWriter`, in the `HttpServletResponse` object by the requested Servlet. The YBox is not able to produce a response for the client based on the output from the Requested Servlet.

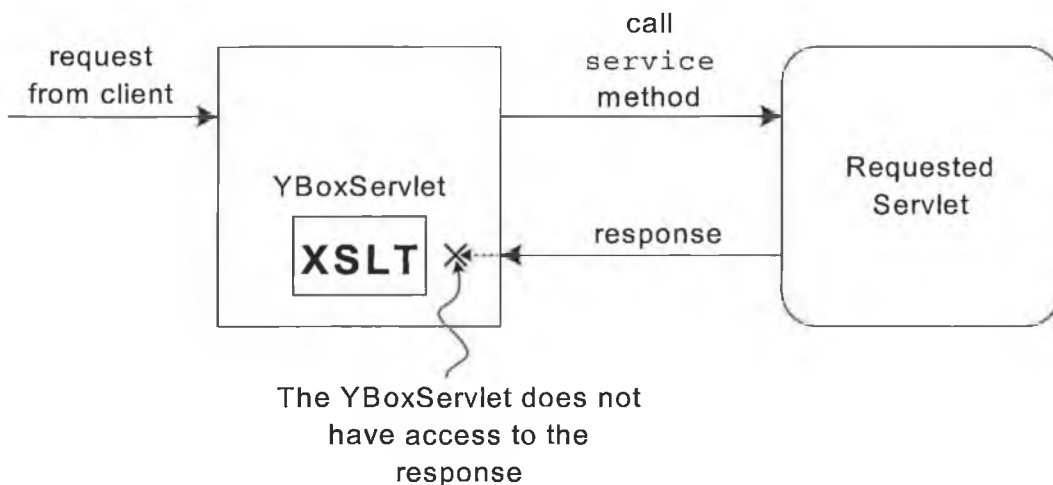


Figure 4.9. YBoxServlet cannot modify the Response from a Servlet

The `PrintWriter` object is a **private** attribute of the `HttpServletResponse` object. The `HttpServletResponse` only has a `getWriter` method. It does not have a `setWriter` method. Therefore, the `PrintWriter` cannot be modified by the `YBoxServlet`. As a result it is not possible to use an XSLT processor on it.

This limitation was overcome by using a `PipedInputStream` and `PipedOutputStream` and calling a method other than the `service` method of the Requested Servlet (this is also related to Form Validation and is explained in detail in section 4.4).

A `PipedOutputStream` is passed to the Requested Servlet as a parameter of this new method. The Requested Servlet now writes its response to the `PipedOutputStream`. This is connected to the `PipedInputStream`, which is used as the input to the XSLT processor. Figure 4.10 shows how this is represented in the YBox.

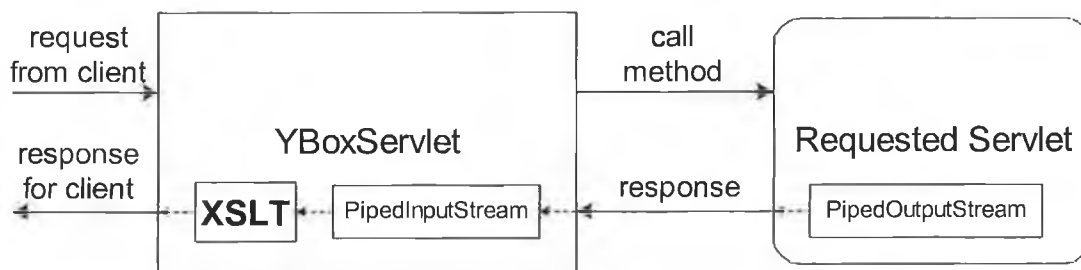


Figure 4.10. Piped I/O Streams used to allow XSLT

With this method of streaming data, it was possible to support XSLT using version 2.2 of the Servlet API. This implementation had one major drawback: **it did not support JSPs**. Very little work had been done on XSLT with JSPs when version 2.3 of the Servlet API was released, so there was no need. Using this technique, it is difficult to see how the YBox would support XSLT with JSPs. Inside a Servlet Container, a JSP is handled by a special Servlet that translates the JSP page into a Servlet class and compiles it. Therefore, the compiled JSP will only have one service method that has two input parameters, a `HttpServletRequest` object and a `HttpServletResponse` object. It will not be possible for the JSP to access the `PipedOutputStream` to write the response, so the `PipedInputStream` will not be available for use with the XSLT processor.

Servlet API Version 2.3

Filters were included in version 2.3 of the Servlet API. As a result, this allowed the content presentation in the YBox to be implemented in a much easier manner. There were also other new classes released in the 2.3 API that greatly help XSLT. These new classes are discussed in detail in this section.

The YBox class actually extends a `GenericFilter` class that implements the `javax.Servlet.Filter` interface. This `GenericFilter` is used to simplify the design of the YBox class. The YBox class only has to implement two methods of the Filter interface:

- `init`
- `doFilter`

In this design, every request to an application using the YBox invokes the `doFilter` method of the YBox class.

The YBox also uses the `FilterConfig` interface. The Servlet Container returns a `FilterConfig` object to the `GenericFilter` after initialisation is complete. Hence, the YBox can access the initialisation parameters, Container information, ...etc. This can be seen in Figure 4.11.

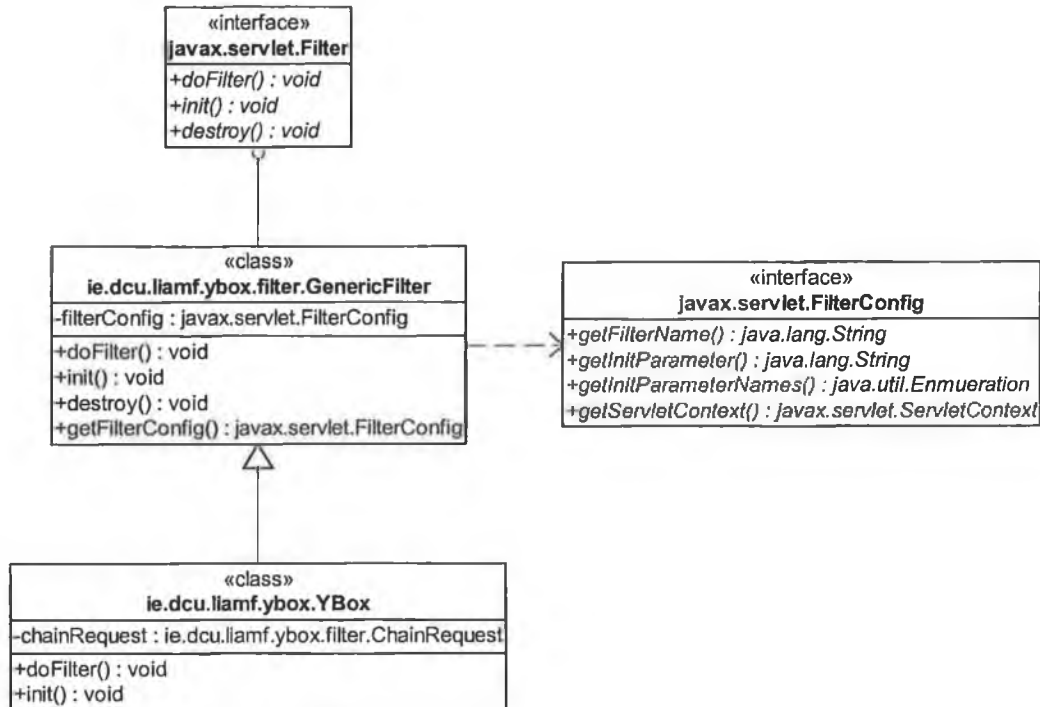


Figure 4.11. The Class diagram for the YBox class

The YBox must be able to manipulate the response from the requested resource (e.g. to transform it from XML to HTML). To do this a new OutputStream object is needed. The problem with the ServletOutputStream is that the YBoxFilter is not allowed to modify it after the requested resource has finished writing it. To achieve this, the ServletOutputStream must be extended and its write methods over-ridden. Now the requested Servlet/JSP is writing to an object the YBox has full access to. Therefore, it is possible for the YBox to modify the response before sending it back to the client. The UML representation for this new class, FilterServletOutputStream, can be seen in Figure 4.12.

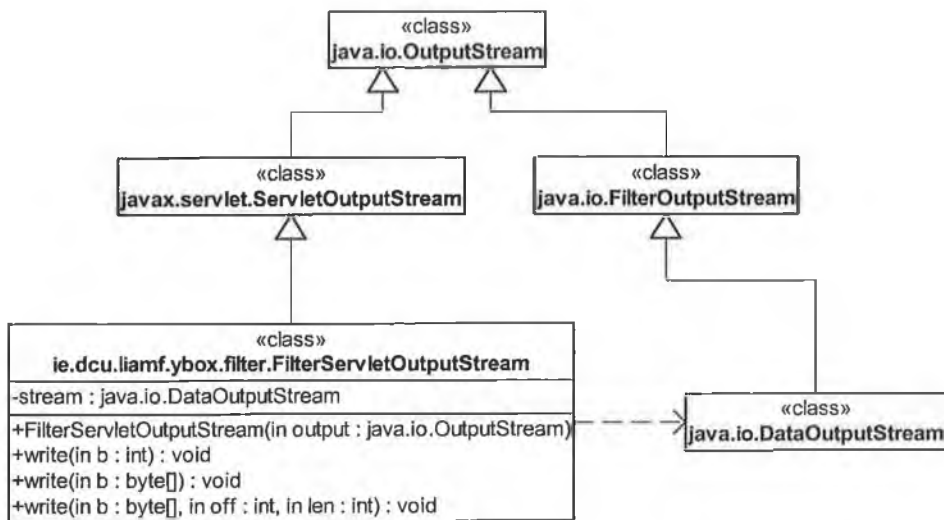


Figure 4.12. UML Class diagram for the FilterServletOutputStream class

The most important class for the YBox design released with the Servlet 2.3 API is the HttpServletResponseWrapper class. This new class extends the ServletResponseWrapper class, allowing any Filter to access/modify the contents of the response object, not just the output stream. For example, it can be used to modify the content type specified by the Servlet/JSP.

The YBox design has extended the HttpServletResponseWrapper class and implemented the methods it needs to allow XSLT to occur before the stream is committed back to the client. The PrintWriter used in the response is generated as an instance of FilterServletOutputStream. This can be seen in Figure 4.13. The getData method of GenericResponseWrapper returns the byte array to the YBox filter for it to transform.

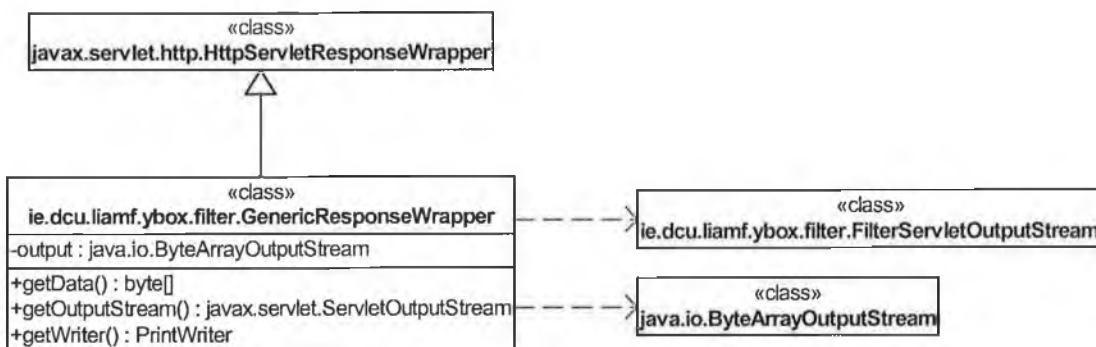


Figure 4.13. Class diagram for the GenericResponseWrapper class

When all the classes described above are put together, the result can be seen in Figure 4.14. The response object received from the Servlet Container is **not** sent to the requested Servlet/JSP. Instead the response wrapper object is sent. The Servlet/JSP writes the output to the `FilterServletOutputStream` object inside the response wrapper object.

The YBox can then extract the contents of this stream as an array of bytes. This is used to construct an `InputStream` object that is used as an input to Xalan, the XSLT processor. The output from Xalan is then sent back to the client.

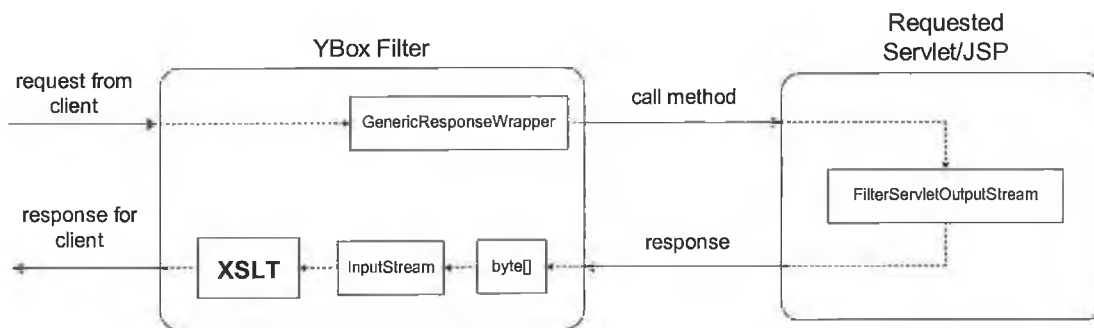


Figure 4.14. Dynamic Content manipulation using XSLT

4.3. Security

For the YBox to implement security for a web application, all requests must be directed through the YBox. The Servlet Container does this from the information it received from the `web.xml` during start-up. The url-mapping contained within `web.xml` must map all requests to the YBox filter.

There are three vital pieces of information needed for security:

1. The resource that is being requested.
2. The user who is requesting it.
3. The access permissions on every resource in a web application.

The first piece of information is easily obtained from the URL. The second piece of information, which is the user, is extracted from the session (this is discussed later).

The final piece of information needed is the access permission. These permissions are extracted from the YBox configuration file.

4.3.1. URL of the requested resource

The YBox must examine every request that the client makes. The URL for a request has the structure:

http://host/web_app/module_name/resource

The *web_app* is the name of the web application that the YBox is controlling. Each web application has a separate instance of the YBox engine.

The *module_name* is the name of the directory the resource is contained within. If the *resource* is a flat file then the *module_name* is a real path to the flat file. If the *resource* is a Servlet, then the *module_name* is the package name the Servlet is contained within, therefore the security can be used to protect certain packages within a web application.

4.3.2. The User making the request

The YBox framework is designed with user-based web applications in mind. A user definition in the YBox is based on the YBoxUser abstract class. The web application designer must extend this class and implement all of its methods to automate security in the web application.

The YBoxUser class is located in the `ie.dcu.liamf.ybox.user` package. Figure 4.15 shows the UML class diagram for the abstract class.

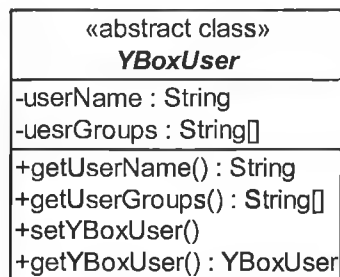


Figure 4.15. UML class diagram of the YBoxUser

The `setYBoxUser` method is used by the web application designer. When a user is successfully authenticated, the web application designer creates an instance of the `YBoxUser` class. `getYBoxUser` is used by the YBox to get the `YBoxUser` object from the user's session. `setYBoxUser` and `getYBoxUser` are static methods of the `YBoxUser` class.

The `HttpServletRequest` object must be passed to both methods. In the case of `setYBoxUser` the YBox must store the user object in the session associated with the correct user. For the `getYBoxUser` method, the YBox uses the `HttpServletRequest` object to retrieve the `YBoxUser` from the session, as the session is stored in the `HttpServletRequest` object.

There is no need to synchronise the methods accessing the instance of the `YBoxUser` class stored in the session. The reason for this is the only time the `YBoxUser` object gets updated, is when the user is being logged into the web application. Once successfully logged in, the `YBoxUser` object is only ever read. There can only be one user per session as defined by the Servlet specification.

4.3.3. Access permissions

All the permissions for the web resources contained within a web application are located in the YBox configuration file. As explained earlier, this file is loaded during the initialisation of the YBox, so they are available in the Servlet Context during every request. This can be seen in Figure 4.16.

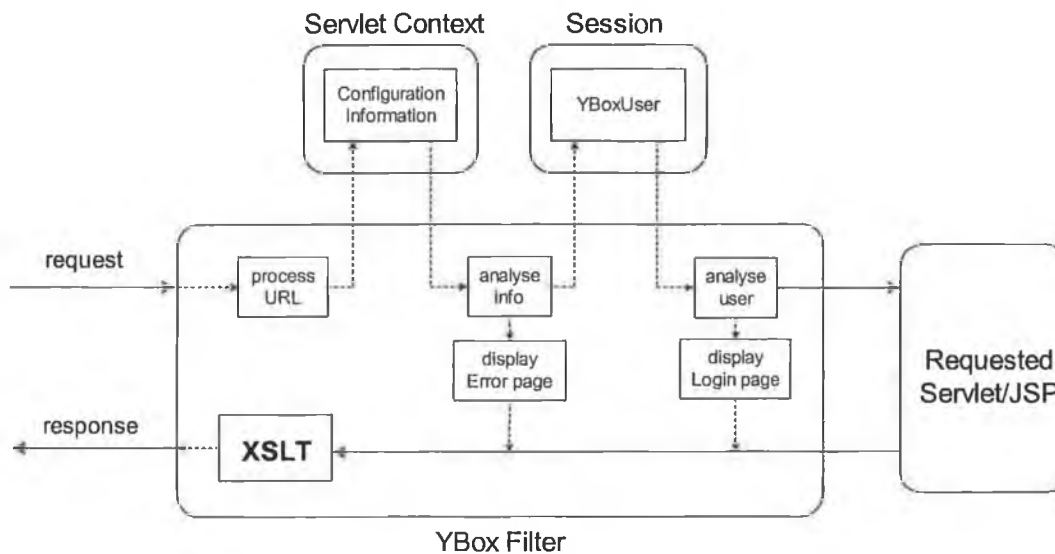


Figure 4.16. Security Control using the YBox

As shown in Figure 4.16, the Configuration Information is stored in the Servlet Context. This is where the information specified in the Configuration file is stored for the entire lifecycle of the web application. There are three types of resources that can be protected:

1. Flat files.
2. Servlets.
3. Directories (including packages for Servlets).

There is an object stored in the Servlet Context to represent each one of these resource types. All three extend a `ContentObject` as can be seen in Figure 4.17, and so all classes have the `getGroups` and `getPersons` methods. The YBox uses these methods to return the group names and user names that are allowed access to the requested resource.

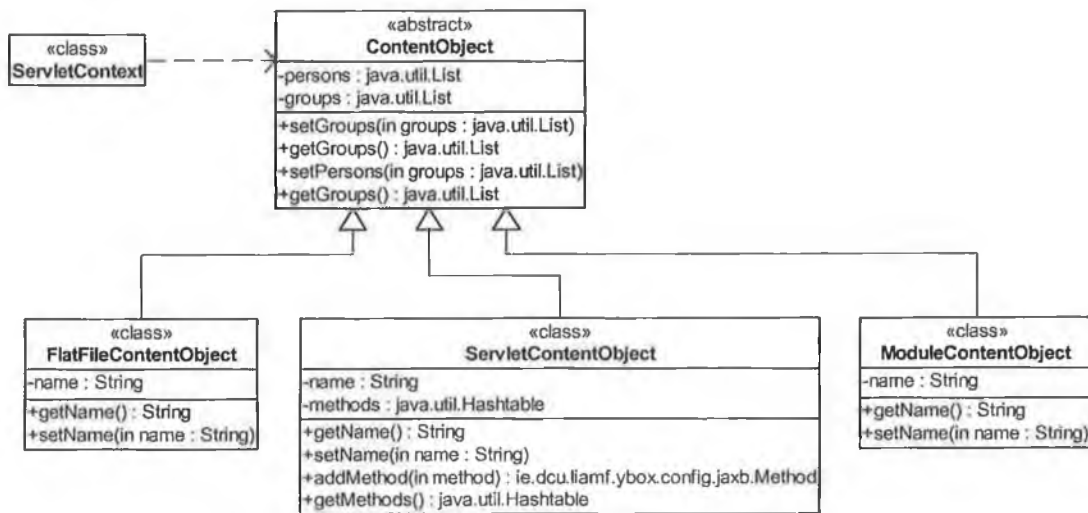


Figure 4.17. Class diagram of the resource types

In the case of a `ServletContentObject`, there are also some additional methods. These are to do with the methods associated with each Servlet. `getMethods` is used by the YBox to check if the requested Servlet has a particular method. If it does, then the request is processed further; if not, the client is sent an error message.

Figure 4.18 shows an example of the security section in the configuration file. The root module (`module name="."`) is accessible to all users. `index.jsp` is located in the root directory but is only accessible to the groups `ee553` and `ee554` and the person, `liamf`. This permission over-rides the permission on the root directory.

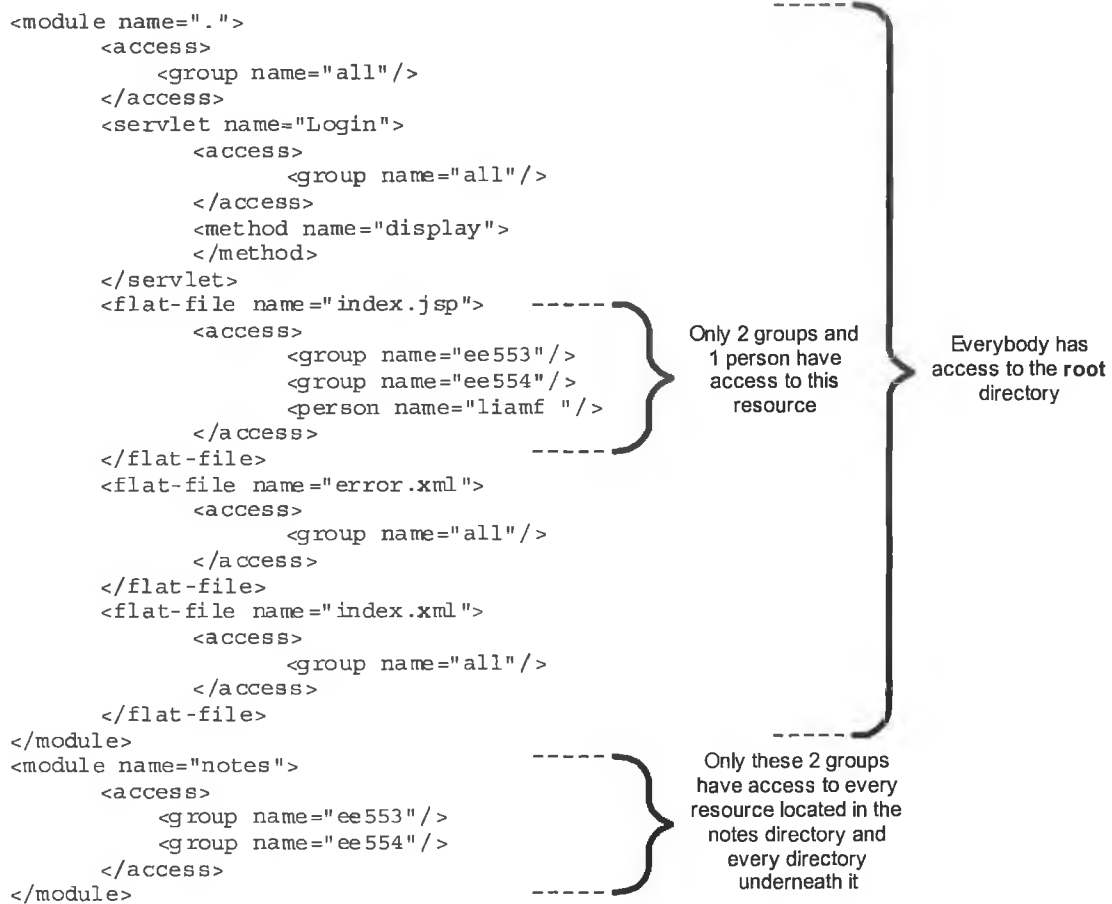


Figure 4.18. YBox Configuration of the Security

The final part of the configuration file associated with security is the error page that is shown if the user does not have the required permissions to view a resource. The YBox will re-direct the user to this page if their user privileges do not allow them to view a resource. In the configuration file, the error-redirect resource is represented by:

```
<error-redirect>error.xml</error-redirect>
```

The error.xml must be listed in the configuration file also (as in Figure 4.18) and all users must have access to it. If not, the YBox will not be able to display the error page, as the user will not have access to it.

4.4. Form Validation

In the YBox, all form validation takes place in the Servlet Container. There is no validation on the client (e.g. JavaScript). Form validation takes place before the resource is invoked, therefore the Servlet Container never executes the Servlet code, if the form does not validate correctly.

4.4.1. Instantiating a requested Servlet

The Servlet Container is not in direct control over the resources (Servlets/JSPs) in a web application when using the YBox. There is a new level of indirection introduced by the YBox. The Servlet Container requests the YBox, and the YBox makes a request to the Servlet/JSP. This means the YBox has to be a **class loader** and dynamically load the requested Servlet at the time of request.

Figure 4.19 shows the steps take by the YBox when getting an instance of a requested Servlet. Step 1 and Step 2 are discussed in previous sections.

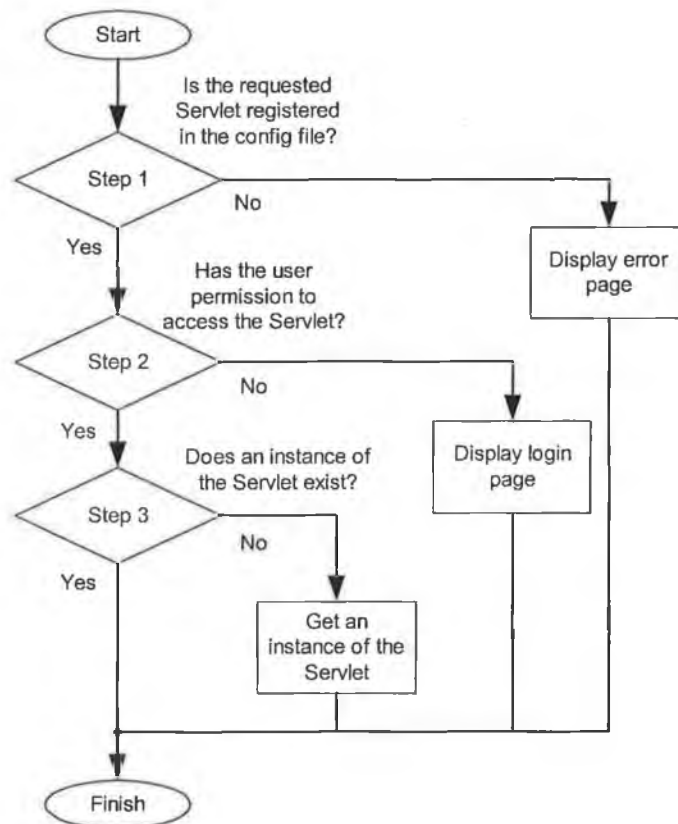


Figure 4.19. Getting an instance of a Servlet

In Step 3, the YBox checks if there is an instance of the requested Servlet exists. An instance of the Servlet will only exist if this is not its first time being requested. Once a request is made, the YBox checks the Servlet context object in the web application for an instance of the Servlet. If an instance exists, then it is used. Otherwise a new instance of the Servlet object is instantiated using the `Class.forName` method. This instance is put in the Servlet context object for use by other requests.

All the instances of Servlets are kept in a `Hashtable`. They are accessed by extracting the name of the resource requested from the URL. There is only one way to force the YBox to re-instantiate a Servlet class file: reload the entire web application. This will “flush” the `Hashtable` in the Servlet context and all Servlet class files will be reloaded again as needed.

4.4.2. Using Reflection to invoke methods

The Reflection API [35] allows the inspection of classes, interfaces and objects. Using Reflection it is possible to get information about a class's modifiers, fields, methods, constructors, and superclasses. The YBox implementation is only concerned with the public methods contained in a Servlet in order to invoke the requested method correctly.

Now that an instance of the requested Servlet exists, the YBox uses the Reflection API to invoke the requested method. The requested method name is extracted from one of the HTTP form parameters. This is returned from the client as a hidden form parameter.

An array of `Method` objects is extracted from the Servlet class with the following piece of code:

```
Class c = ServletInstance.getClass();  
java.lang.reflect.Method[] m = c.getDeclaredMethods();
```

With all the public methods of the requested Servlet now known, they can be compared to the requested method name.

More than one method match can occur. This is because Java supports method **overloading**, as does the YBox. To explain this in more detail, take the following example.

The `LoginServlet` takes control of all logins for a particular web application. The web application developer is required to implement two different `login` methods: one for testing, and one for deployment. When testing, the tester will not be required to enter a password, only a user name. When deployed, every login will require a user name and password. Therefore, the class `LoginServlet` has a structure represented by Figure 4.20. The `login` method is overloaded in this case.

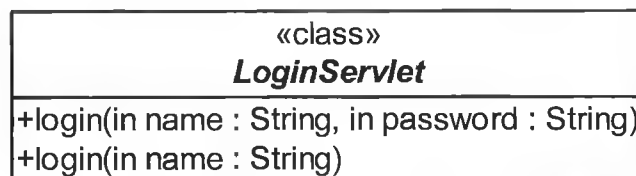


Figure 4.20. The LoginServlet Class

As can be seen, there are two `login` methods. Using the reflection API, the YBox does not know which one to invoke. To overcome this, the YBox invokes each method (where the method name matches) in turn until no exception is thrown. If the YBox has attempted to invoke every method and all threw an exception, then the requested method does not exist. In this case an error message is displayed on the client.

The public methods in the requested Servlet that the YBox can invoke must return nothing (i.e. they must be **void**). This is equivalent to the `doGet`, `doPost` and `service` methods of a normal Servlet as they too return **void**.

4.4.3. Rules for Validation

The rules for validating a form are based on the XML source that generates the form. The form includes hidden parameters to indicate to the YBox the rules for validating each input field. These hidden fields tell the YBox if the field is required, the field type and if it belongs to a group of fields representing an object.

To see this in action, take the case of the source XML being transformed into HTML. The XML for this form is:

```
<form      Servlet="LoginServlet"
          method="login(name) ">
  <input-text      type="String"
                  name="name"
                  descr="Name" size="10"
                  required="true"
                  value=""
                  errorMsg="Please enter your User Name">
</input-text>
  <input-button
          value="Submit">
</input-button>
</form>
```

The resulting HTML has hidden fields for the type and whether it is required or not. It also has hidden fields for the method name that is being requested. There is no need to have a hidden field for the requested Servlet, as that is already part of the URL. The `errorMsg` attribute is used to redisplay the form with error messages. This is discussed in section 4.4.4.

Required field

If the `required` attribute of the `input-text` tag is true, then if the field is left blank, the YBox will not instantiate the Servlet and hence not invoke the requested method. The same page will be displayed again with an error message beside the field that failed to validate. If the `required` attribute is false, then the YBox will allow a blank input field.

The YBox only compares the input to the null string in Java (""). If there are spaces entered, then the YBox considers this an input and attempts to cast it to the specified type. This may fail, depending on the type.

The example in Figure 4.21 displays a very simple form. There is only one input field ("Username") and it is required.

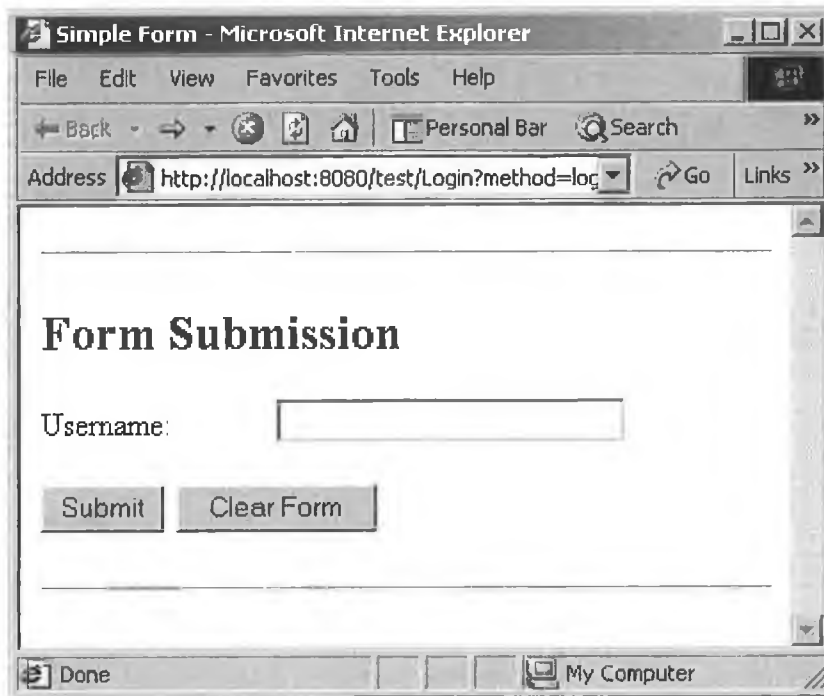


Figure 4.21. Simple form with one required input field

The user clicks “Submit” without typing anything in the input field. The result can be seen in Figure 4.22. The LoginServlet does not get instantiated in this case and the login method does not get invoked. Instead, the same page gets redisplayed with a new, more urgent message on the failed input field.



Figure 4.22. Validation on simple form failed

Basic types – int, double and String

The YBox supports validation of these three basic types. If the field is not blank and is required, then the YBox attempts to **cast** the value of the input field to the required type. For the int and double types, the `java.lang.Integer` and `java.lang.Double` classes are used respectively.

If validation fails, the page is re-displayed with the appropriate error messages. Take another simple example. The form only has one input field and it is for the age of the user. The form is expecting an integer, but the user could enter a String. Figure 4.23 and Figure 4.24 displays the results.

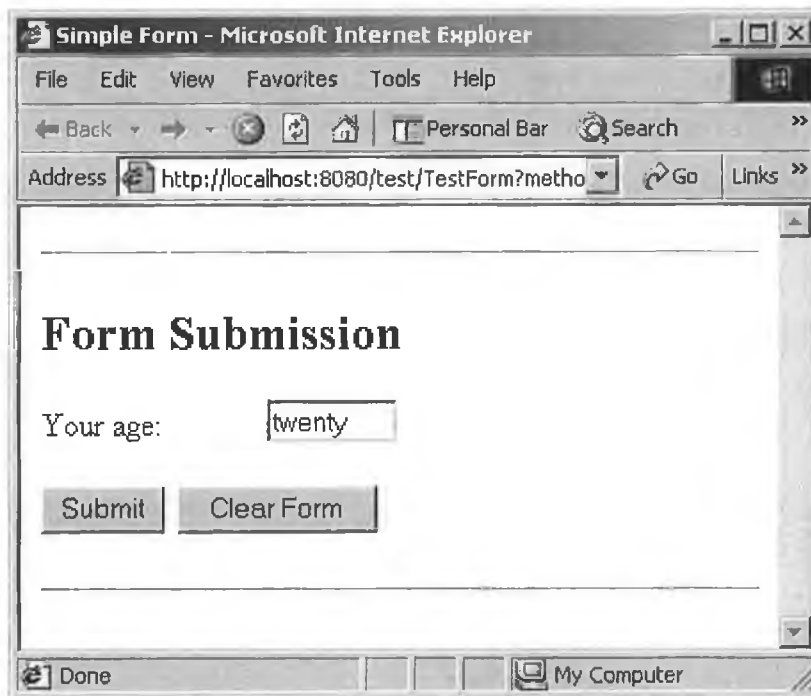


Figure 4.23. Simple form with one integer field

The user enters “twenty” instead of “20” in the input field. The validation fails as “twenty” cannot be cast into an `Integer` object.

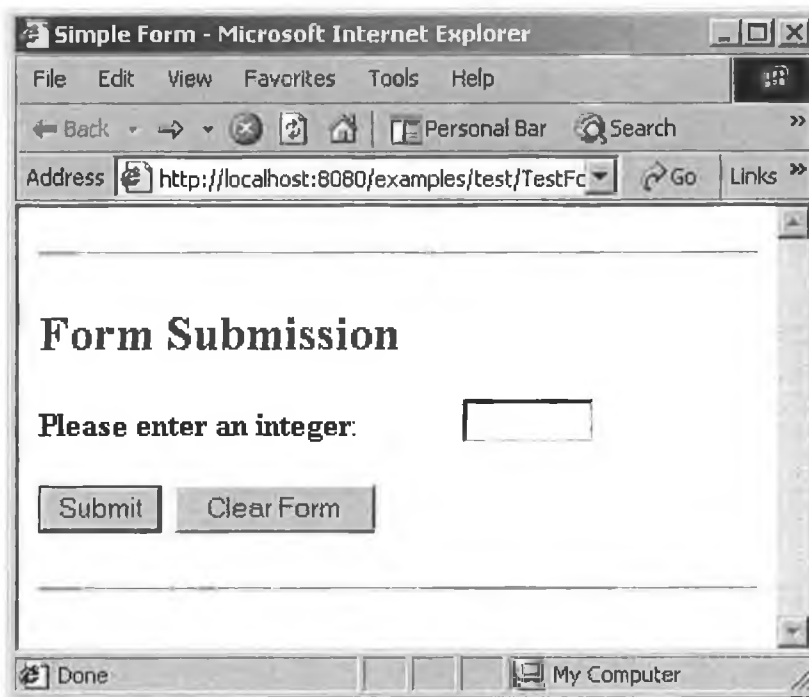


Figure 4.24. Failed to cast input to Integer

Custom classes

The YBox also supports one or more input fields being grouped together for validation. The methods of validation mentioned above cannot actually check if the **value** of the input field is correct – they can only check the type. This is where user defined custom classes can be used to validate the value of multiple input fields together.

These are the steps the YBox must perform to validate a form with one or more input fields, where all fields are associated with one custom class:

1. Validate each input field against the rules for that field (required and type).
2. Only when all fields have validated correctly, will the results be passed to the constructor of the custom class.
3. The YBox then attempts to instantiate the custom class based on the validated input fields.

The validation gets more complicated the more fields and custom classes a form has, but the principle stays the same. Figure 4.25 shows the basis of form validation with custom classes. From this figure it can be seen what happens when the validation

fails. The same resource must be redisplayed with error messages beside any input field that failed.

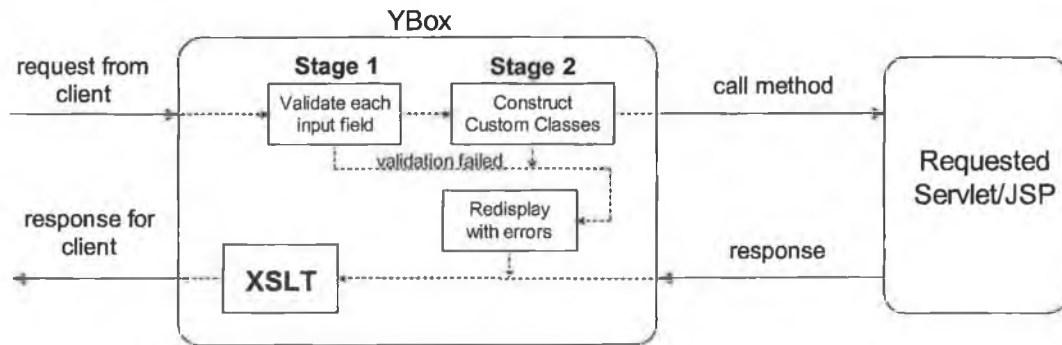


Figure 4.25. Form Validation using a Custom Class

For **Stage 1** in Figure 4.25, the error message is extracted from the `errorMsg` attribute of the XML source. This message is passed to the Redisplay processor and the page is generated with error messages included. The processor class that does this redisplaying is:

```
ie.dcu.liamf.ybox.form.ReDisplayFormWithErrors
```

The difficulties arise when the YBox must associate an error message with **Stage 2** in Figure 4.25. If the value of one of the input parameters to the Custom Class constructor is not the desired value, the Custom Class has to notify the Redisplay processor. It does this by throwing an exception. The object does not get instantiated if the constructor throws an exception, therefore the custom class will not hold on to any resources.

The constructor of the Custom class must throw a special type of exception:

```
java.text.ParseException
```

The class diagram for this exception can be seen in Figure 4.26. The reason this type of exception is so important is the YBox must be able to identify which input parameter of the constructor is incorrect.

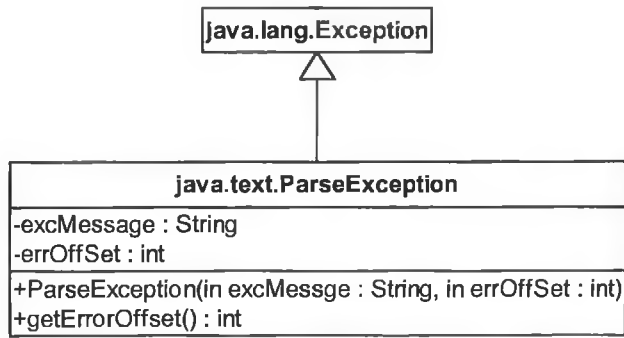


Figure 4.26. Class diagram of the ParseException class

The `excMessage` (exception message) is specified in the exception thrown by the constructor and is passed to the redisplay processor. This is placed beside the input field that caused the constructor to throw an exception. The integer, `errOffset`, is used to tell the YBox which input parameter failed validation. This is also specified in the exception thrown by the constructor.

To understand this it is best to use a simple example. A form has two input fields, one String and one integer. The String represents the brand name of a shoe, and the integer represents the size of the shoe. These two input fields should be validated based on:

- They are both required.
- The brand name is a String.
- The size is an integer.

The two input fields are grouped together into a Shoe object. For example, this shoe object can check a database to ensure the brand name is valid, and the shoe size is in stock. The source code for the XML form is shown in Figure 4.27.

Inside the `class` tag, the instance name of the class is given by the `name` attribute. The constructor for the class is given by the `constructor` attribute. The constructor to the `shop.Shoe` class has two input parameters, a `size` and a `manufacturer`. These get mapped to the input fields of the form using the method shown in Figure 4.27.

The scope of the object is used to access its constructor variables. `shoe.size` and `shoe.manufacturer` are the names of the input fields.

```

<form servlet="PurchaseShoe" method="purchase(shoe)">
<class      name="shoe"          constructor="shop.Shoe(size, manufacturer)">
</class>
<input-text type="int"
  name="shoe.size"
  descr="Shoe Size"
  size="3"
  required="true"
  value=""
  errorMsg="Please enter your size">
</input-text>
<input-text type="String"
  name="shoe.manufacturer"
  descr="Shoe Manufacturer"
  size="10"
  required="true"
  value=""
  errorMsg="Please enter the shoe manufacturer">
</input-text>
<input-button value="Purchase Shoe"></input-button>
</form>

```

Figure 4.27. XML source for a form

The constructor of the `shop.Shoe` class must now be examined. Figure 4.28 shows the syntax for this constructor. It does no database look-ups on the manufacturer or size. It is only for test purposes, so it is as simple as possible.

```

public Shoe(int size, String manufacturer) throws ParseException
{
    if(! ( manufacturer.equals("echo") ||
           manufacturer.equals("clarks") ||
           manufacturer.equals("cats")))
    {
        throw(new ParseException("Shoe manufacturer " +
manufacturer + " not recognised", 1));
    }

    if(size<4 || size>14)
        throw(new ParseException("Please...shoe size must be
between 4 and 14", 0));
}

```

Figure 4.28. Constructor of Shoe class

For the constructor, the 0th parameter is size. The 1st is manufacturer.

If the manufacturer does not match some hard coded values, then a `ParseException` is thrown. The YBox catches this exception and extracts the required information from it. The YBox passes this information to the Redisplay processor, so the error message associated with constructing the object is displayed for the client.

The following figures show how the form is displayed when viewed in Internet Explorer. The case where the user does leaves a field blank or does not enter an integer for the size is not covered. The cases that are dealt with are:

1. User enters an invalid manufacturer.
2. User enters an invalid size.



Figure 4.29. User fills out "Purchase Shoes" form

In Figure 4.29, the user has filled out the form in a manner that he/she thinks is correct. When he/she clicks "Purchase Shoe", the YBox attempts to validate the form. Referring to Figure 4.25, Stage 1 of the validation process succeeds but Stage 2 fails. Figure 4.30 shows what happens after the user clicks "Purchase Shoe". Note the YBox will only clear fields that failed validation.



Figure 4.30. "Purchase Shoes" with error messages on manufacturer

For this example, the user enters "clarks" for the next attempt (see Figure 4.31). This is a correct manufacturer as can be seen from the constructor code in Figure 4.28. But the shoe size is still incorrect. It is possible for the YBox to display a combination of original messages (`descr`) and error messages (`errorMsg`) again. In these examples, the error message is only displayed.

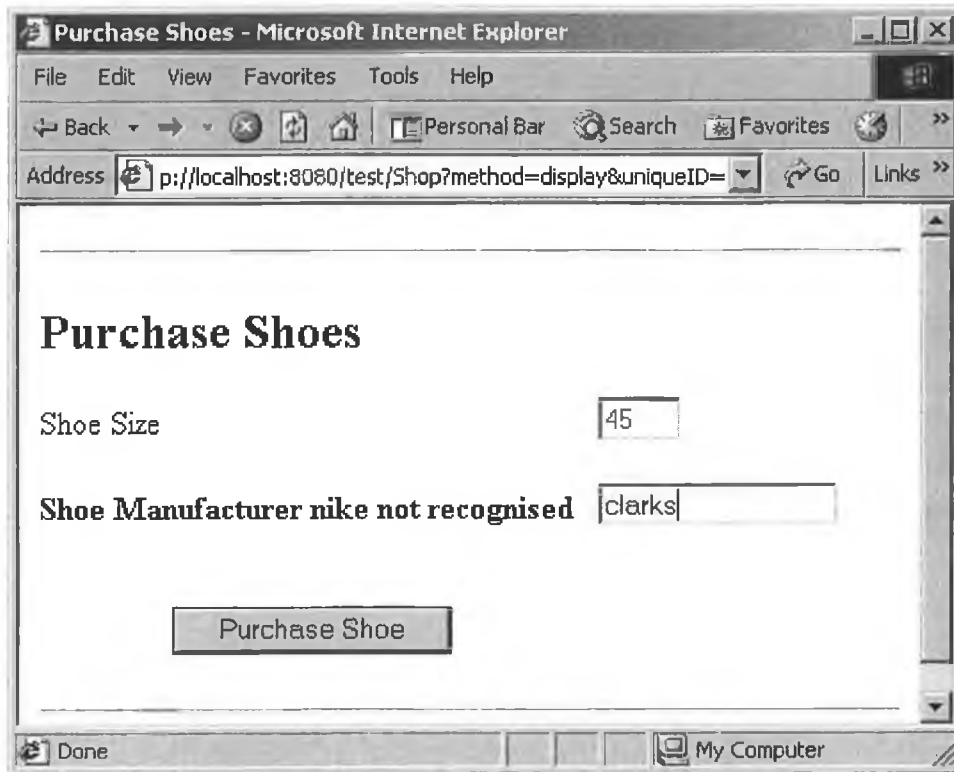


Figure 4.31. User fills out “Purchase Shoes” form again

The resulting page can be seen in Figure 4.32. The message for the manufacturer is gone back to the original. This informs the user that the manufacturer was recognised correctly. Now, the error message is displayed beside the shoe size. This cycle of error messages are repeated until the user gets the form correct.

When constructing the Shoe object, only one exception can be thrown because any Java method cannot throw more than one exception. By using a ParseException class, only information on one of the parameters can be contained. Therefore, using the ParseException class it is not possible to detect more than one error while constructing the Shoe object.

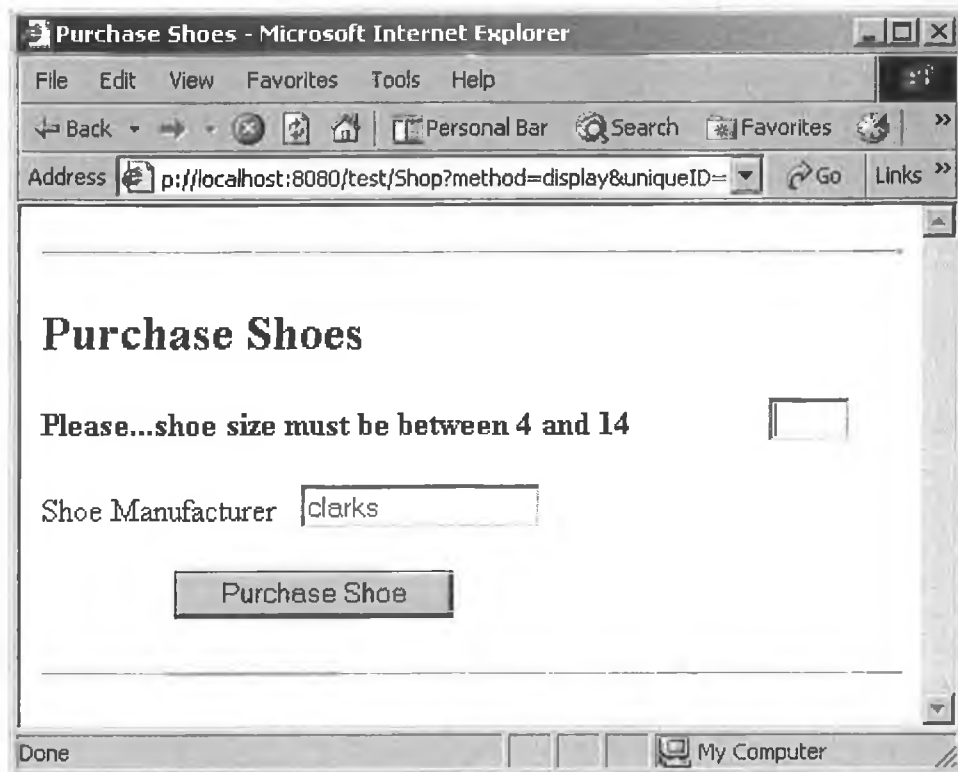


Figure 4.32. "Purchase Shoes" with error messages on size

This means the requested method of the requested Servlet never gets invoked unless the form is filled out correctly. This helps separate the validation logic from the presentation logic. The Servlet does not have to validate any user input as the YBox does it all.

4.4.4. Redisplaying the resource with Errors

Every time the user incorrectly fills out a form, the YBox must redisplay the same form again. This process is repeated until the user fills out the form correctly. The class that controls this process is:

```
ie.dcu.liamf.ybox.form.ReDisplayFormWithErrors
```

Stage 3 in Figure 4.33 shows how the YBox achieves this. It stores all XML responses from Servlet/JSPs or static files in the session. Therefore, if the user incorrectly fills out the form, the YBox can get the original form from the session, insert appropriate error messages and send the result back to the client.

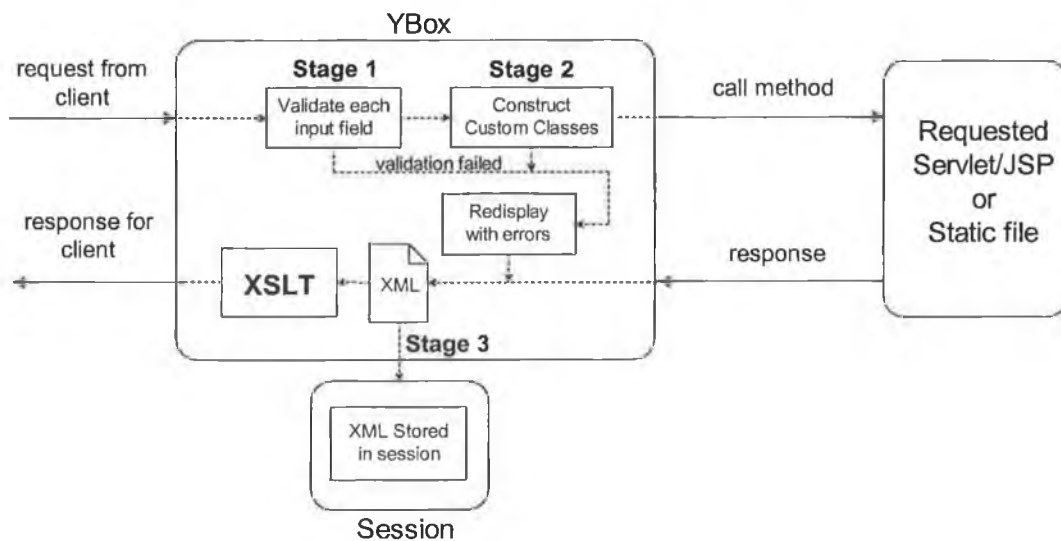


Figure 4.33. Every response is stored in the Session

The error message that is redisplayed is extracted from the original XML source. The `input-text` tag has an `errorMsg` attribute that is used by the web application designer to specify the error message that will be displayed, in the event of that input field not validating correctly.

The `ReDisplayFormWithErrors` loads the XML into a DOM tree and traverses its source looking for `input-text` elements. Once it finds a matching element, it checks to see if this element has failed validation. If it has not failed, then it will not change the `input-text` element in the DOM tree. One important feature to note is that the YBox will preserve the correct value of the input field. It does this by replacing the `value` attribute with the value the user entered.

In the event the validation fails at Stage 2, the error messages that are displayed are the `ParseException`s thrown by the constructor of the custom class. These are passed to the `ReDisplayFormWithErrors` class and redisplayed in the same manner as described above.

If the element fails, the `ReDisplayFormWithErrors` class will replace the `input-text` tag with an `input-error-text` tag. This means the YBox is flexible as the XSLT process can be used to chose how the error message is displayed. The `descr` attribute can be displayed again in bold and in red font. The `errorMsg`

attribute could also be displayed in brackets. This decision is the responsibility of the web application designer. In the case of the element failing validation, the `value` attribute will be cleared, as it is incorrect.

As mentioned previously, JAXB is much easier to work with than DOM, so therefore the question needs to be asked: can JAXB be used for this function? The answer is no.

The reason is JAXB relies on knowing the DTD of the XML source at the YBox design time. This is not possible for web applications that are being designed for the YBox. The DTD is only known at deployment, therefore it will never be possible for the YBox to use JAXB for this purpose.

4.4.5. Problems associated with form validation

There are some problems associated with the YBox implementation of form validation. The first problem is the back button on web browsers. It can cause the YBox and the web browser to “get out of sync”. The second problem is that some design errors are not available for the web application designer at designer time – they are only visible when the designer deploys the web application on a Servlet Container.

Back button

The “**Back**” **button** is a feature of all common web browsers (clients). It allows the user to view pages previously visited before the current page being viewed. It does not require another HTTP POST/GET to display the page. This feature relies on cached information stored on the client.

Using client side JavaScript it is possible to disable this button on some browsers, but this solution is not portable across all browsers. Also, the YBox is a server side framework, so this solution has no place in the current implementation.

The YBox deals with this by caching the content on the server also. It stores up to ten⁷ pages in the user session. Each page is accessed by a unique ID that is sent as a hidden input field to the client with every request. This unique ID is inserted into the

⁷ Ten was chosen for this implementation of the YBox. More (or less) XML files could be cached using the YBox by changing this parameter.

source XML before the XSLT stage. The XSLT processor (Xalan) detects this unique ID in the XML source and generates a hidden input field and sends it to the client.

If the user clicks back one or more times, the unique ID changes on the page they are viewing. If they were to submit a previous form again, the unique ID is sent to the YBox. Figure 4.34 shows the table that exists in the session of the user. This table is actually as `Vector` with a size of ten (only four are shown in Figure 4.34). Once the 11th form needs to be cached, the YBox overwrites the 1st entry in the `Vector`.

Session





Unique ID	Cached XML forms
1	Form A 
2	Form B 
3	Form C 
4	Form D 

Figure 4.34. XML forms cached in Session

The YBox only needs to use the unique ID if this form does not validate correctly. When this happens, the YBox uses the unique ID to get the original XML source from the session. It then sends this XML to the `ReDisplayFormWithErrors` class and the YBox deals with this as normal.

No errors available at design time

The YBox does not catch the following two design errors until the deployment of the web application.

1. Method invocation problems.

2. Custom class instantiation.

When designing a web application, all web forms will attempt to invoke methods of Servlets. It is not possible to get a compile time error if this method does not exist using the YBox. It is also possible that the types of the input parameters will not match those of the Servlet's method as a result of a mistake made by the web application designer.

Another design time error can occur when using custom classes to validate user input. Since the YBox dynamically loads the custom class, the designer does not know at design time if the custom class exists and is accessible to the YBox. The user may not use the correct constructor or the class may not be in the web applications classpath. All these errors can only be caught during deployment.

4.5. Session Management

Session Management in the YBox must ensure all sessions can be saved/restored in a manner that is Servlet Container and OS independent. Work had not started on this aspect of the design using the version 2.2 of the Servlet API. Version 2.3 of the Servlet API included new features that made this task much easier.

Using the new features of the Servlet API it is possible to register **listeners** with the Servlet Container that notify the application when certain events related to the session occur. These listeners are part of the Servlet 2.3 specification, therefore, all compliant Servlet Container support this new feature.

4.5.1. Using Session Listeners

There are several listeners that can be used but the two that are used in the design of the YBox are [22]:

1. HttpSessionListener
2. HttpSessionAttributeListener

These are interfaces that must be implemented. The methods of which are invoked when a particular event occurs.

Explanations of the methods of the `HttpSessionListener` interface are:

- `sessionCreated(HttpSessionEvent se)` – this method is invoked by the Servlet Container every time a session is created. The `HttpSessionEvent` that is passed to this method contains the actual `HttpSession` associated with the event.
- `sessionDestroyed(HttpSessionEvent se)` – invoked by the Servlet Container every time a session is invalidated (times out or user logs out).

Explanations of the methods of the `HttpSessionAttributeListener` interface are:

- `attributeAdded(HttpSessionBindingEvent se)` – the Servlet Container invokes this method when the user adds an attribute to the `HttpSession`. The `HttpSessionBindingEvent` gives visibility into the object name and value that is being added.
- `attributeRemoved(HttpSessionBindingEvent se)` – invoked by the Servlet Container when it is removing an attribute from the session. This can be because the Session is being destroyed or the user called the `removeAttribute()` method of the `HttpSession` object.
- `attributeReplaced(HttpSessionBindingEvent se)` – invoked by the Servlet Container when an existing attribute in the `HttpSession` is overwritten.

The `YBoxSessionListener` implements the `HttpSessionListener` and the `HttpSessionAttributeListener` interfaces. The methods of these interfaces described above notify the `YBoxSessionListener` when a session event occurs.

The UML class diagram for this new class can be seen in Figure 4.35.

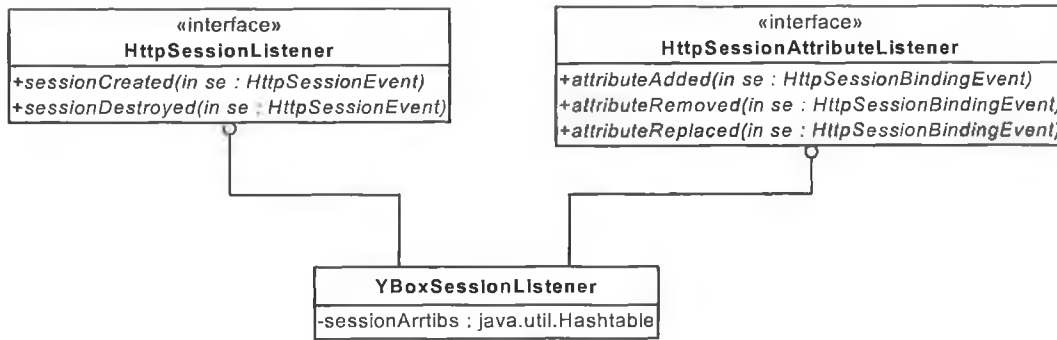


Figure 4.35. UML class diagram for the YBoxSessionListener

The private sessionArrtibs Hashtable in the YBoxSessionListener class is used to store the session attributes. Whenever the session is invalidated, this Hashtable object is populated with the session attributes.

4.5.2. Registering Session Listeners

The listeners must be specified in the web application configuration file, *web.xml*. During initialisation, the Servlet Container extracts this piece of configuration information from *web.xml*. The session listener class is then “registered” with the Servlet Container.

This is the listener extract from *web.xml*.

```

<listener>
  <listener-class>
    ie.dcu.liamf.ybox.session.YBoxSessionListener
  </listener-class>
</listener>

```

Because the YBoxSessionListener implements the HttpSessionListener and the HttpSessionAttributeListener and interfaces, it gets notified by the Servlet Container whenever a session begins, ends or an attribute is added, removed or modified within a session.

4.5.3. Saving the Session Attributes

When saving the session to persistent storage, the entire HttpSession object is not saved. Only session attributes the user has added should be saved. Attributes the YBox has added (e.g. previous XML forms) are not stored.

The `YBoxSessionListener` class does not actually save any of the session attributes. This is the responsibility of the web application designer. They can use a databases or flat file to save the session attributes. This is performed in the `saveSession()` method of the `YBoxUser` class.

Inside the `YBoxSessionListener` class, the following steps are taken to save the session attributes:

1. Each time the `sessionCreated()` method is invoked, the `YBoxSessionListener` class creates a new `Hashtable` object to store the session attributes. This `Hashtable` object is accessed by the unique session ID.
2. Whenever the Servlet Container invokes the `attributeRemoved()` method, the `YBoxSessionListener` class checks to see if the session is actually being invalidated (this method can be called directly by the web application designer). If the session is being invalidated, it adds the attribute name and value to the `Hashtable` object accessed by the session ID.
3. Finally the `sessionDestroyed()` method gets invoked by the Servlet Container. This method gets the `Hashtable` object associated with the session ID and passes it to the `saveSession()` method of the `YBoxUser` class.

As can be seen from the above steps, some of the methods of the `YBoxSessionListener` class are not used. There is no need to associate an event each time an attribute is added or modified. The YBox is only interested whenever an attribute is removed and the session is being invalidated at the same time.

Figure 4.36 shows the Servlet Container notifying the YBox that the session has been invalidated. When the YBox has got all session attributes inside the `Hashtable`, it passes the `Hashtable` to the `YBoxUser`. The `YBoxUser` can then serialise this `Hashtable` object and store it as a “blob” in a database or in a flat file. The `YBoxUser` can also store them as name value pairs in a database/flat file. There must

be a primary key associated with each session's attributes. This could be a user name, but this is left for the web application designer to implement at design time.

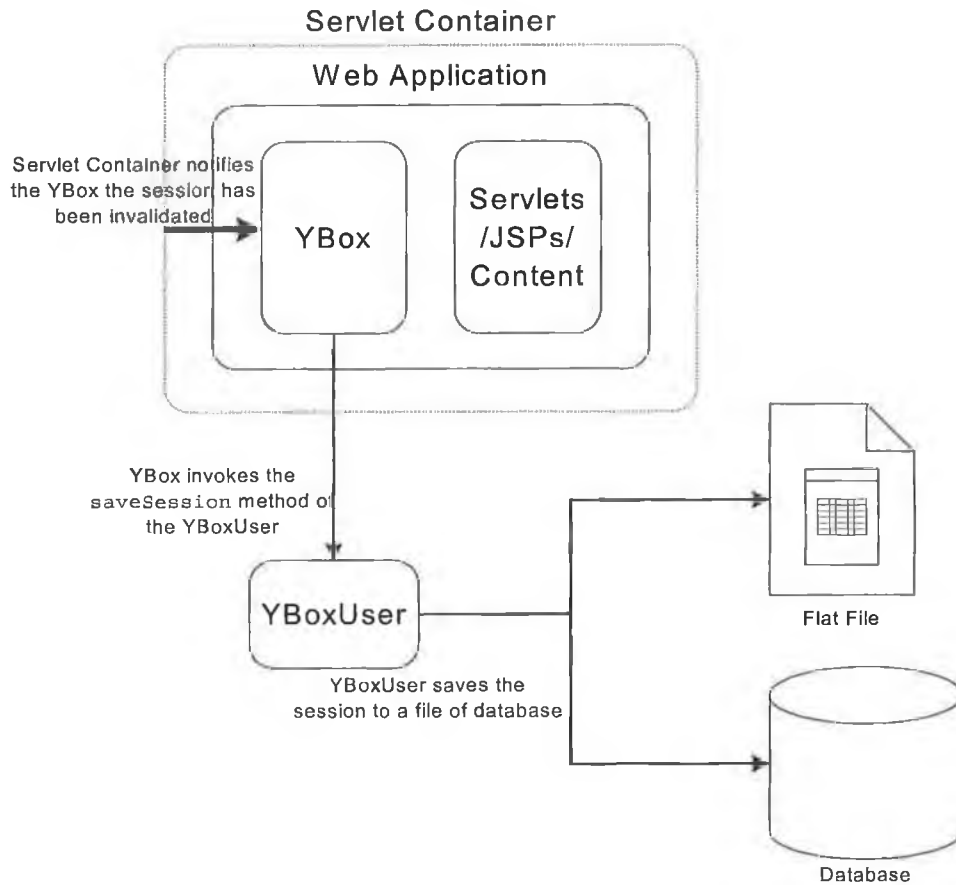


Figure 4.36. YBoxUser object saving the session attributes to persistent storage

4.5.4. Restoring the Session Attributes

When the YBoxUser object is constructed, it is possible for the session attributes to be restored. This is for the web application designer to implement. The session attributes are extracted from the flat file/database and can be restored to the HttpSession object. This can be seen in Figure 4.37.

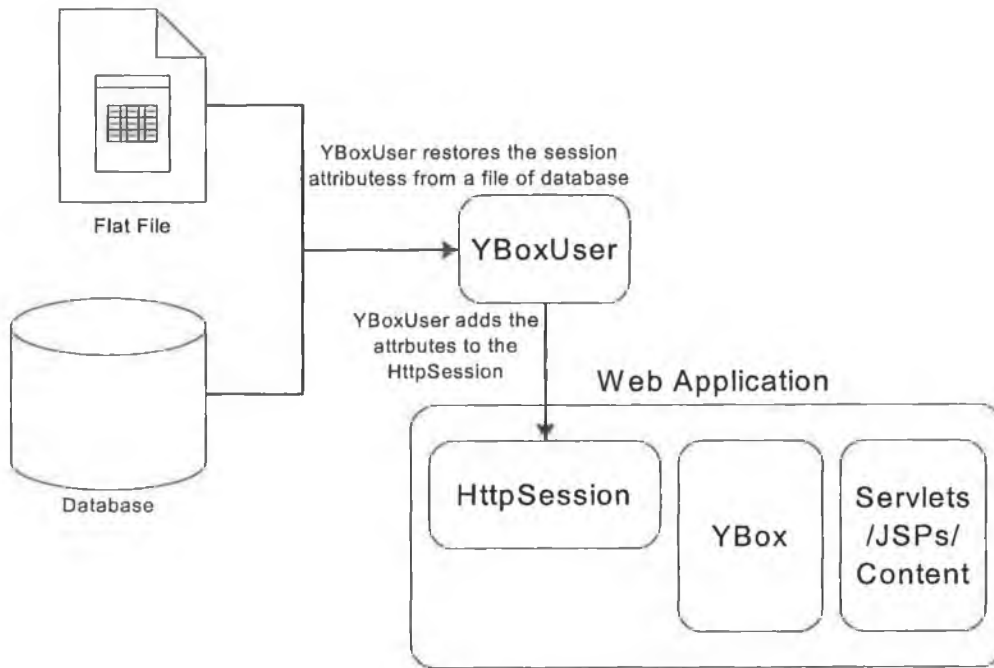


Figure 4.37. YBoxUser restoring the session attributes from a flat file/database

When restoring session attributes, the YBox or the YBoxUser does not interact with the Servlet Container. This method of saving/restoring session attributes in platform and Servlet Container independent.

4.6. Summary

This chapter examined the major features of the YBox framework and how they are implemented. The technologies used and how they evolved during the design phase is analysed with particular attention to Java and XML. The use of UML class diagrams graphically show the structure of the YBox and where it fits into the Servlet API. Appendix A shows how the full implementation of the YBox is represented and the data flow through the system.

The next chapter describes the testing of the YBox, including functional and stress testing. Functional testing involves a sample web application using the YBox and ensuring all feature operate as expected. Stress testing is used to show how the YBox performs under stressful loads and how it compares to other similar frameworks.

Chapter 5 - Testing of The YBox

Like any software application the YBox must be tested thoroughly before it can be used in a live deployment environment. Any bugs that exist must be found and fixed. The YBox must also be tested to ensure the performance of a Servlet Container does not deteriorate to a level that is not acceptable as a result of the load added by the YBox. Lastly, the YBox needs to be able to handle internal errors in the web application. These could result from errors in the configuration file, or the YBox itself getting into an unstable state.

The three approaches to testing that this chapter examines are:

1. Functional testing.
2. Performance and stress testing.
3. Error handling.

Functional testing ensures the YBox conforms to the specification. A sample application is used to test all functional features of the YBox (security, form validation, session management and content presentation). The performance of the YBox is measured using JMeter [26] (an open source testing tool from Apache). Varying loads are applied to the YBox and the results are shown. The performance is also compared to a web application that was designed without the YBox.

Error handling in the YBox is shown by forcing the YBox into various error states. It can be seen how the YBox deals with these error conditions and how the YBox does not “crash” or “hang” the Servlet Container.

5.1. Functional Testing

Before discussing the functional testing in detail, a description of the sample web application is given. It is important to note the sample web application is for demonstration purposes only and the content included is only to test the YBox and show the behaviour of the YBox.

The sample web application must test the four major features of the YBox:

- Content Presentation.
- Form Validation.
- Security.
- Session Management.

The entire source code for the sample application can be seen in Appendix B.

To test content presentation using the YBox, the sample web application has to be tested with multiple clients and content types. Form validation is tested by designing forms that perform validation based on input types and custom classes. Protecting certain resources from different users and groups of users tests security. Finally, putting objects onto the session tests the session management. This object must remain in the session even if the user logs out and logs back in again.

5.1.1. Testing Content Presentation

Basic content presentation allows the provision of content to multiple clients based on the user agent of the requesting client. To show this working, basic XML content is displayed on four different clients (Internet Explorer on Windows 2000, Internet Explorer on Windows CE, Palm V and Nokia WAP browser).

Below is the XML source that is displayed. This is transformed using XSLT based on the different clients that request the page. In the case of the Nokia WAP browser, the XML is transformed into WML.

```
<?xml version="1.0"?>
<page>
  <title>Test Page</title>
  <paragraph>This is a test page.</paragraph>
  <paragraph>If you can view this, the YBox is alive.</paragraph>
</page>
```

Figure 5.1 to Figure 5.4 show how the page is displayed on various platforms (Note: Figure 5.2 is a screen grab from a Compaq Ipaq).

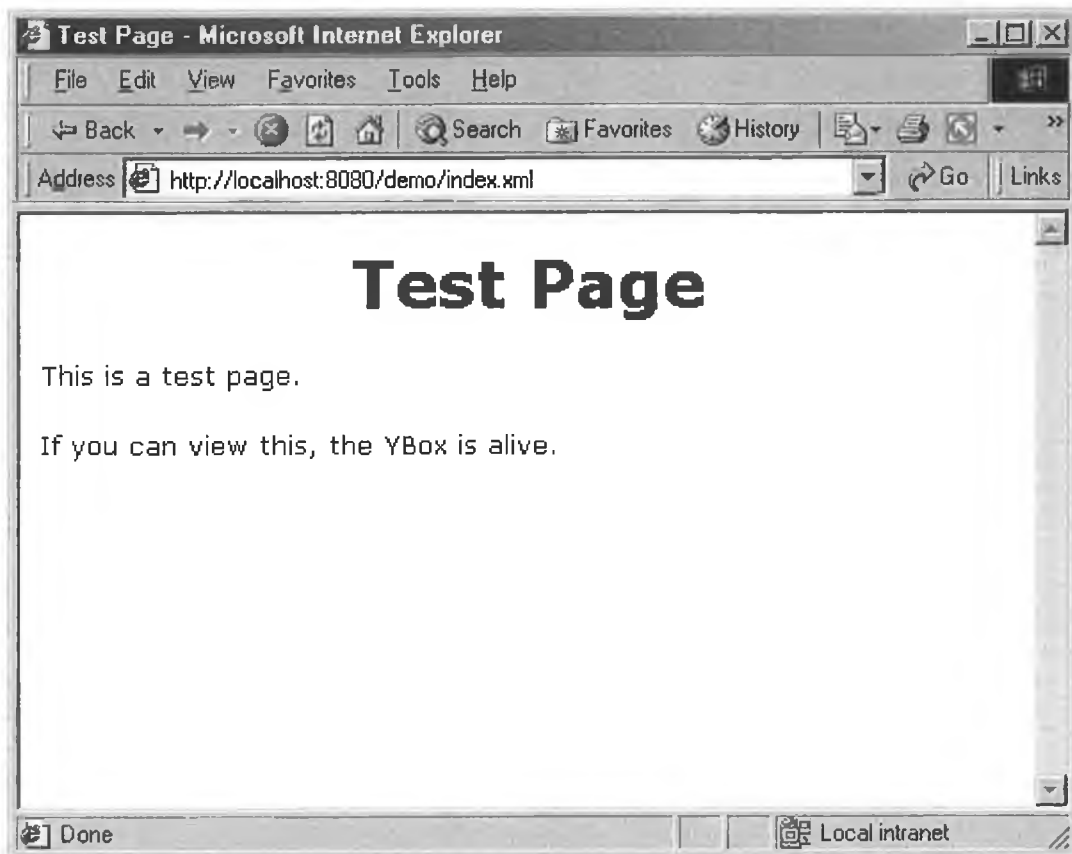


Figure 5.1. Test Page on Internet Explorer (Windows 2000)

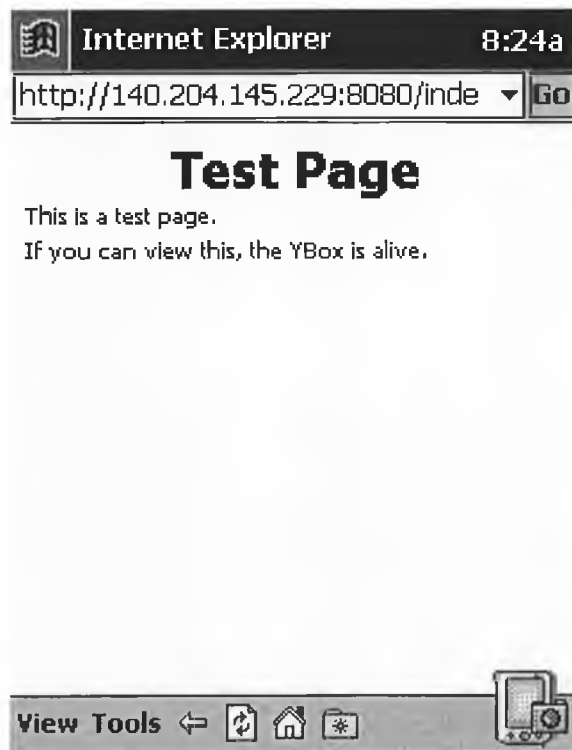


Figure 5.2. Test Page on Internet Explorer (Windows CE)

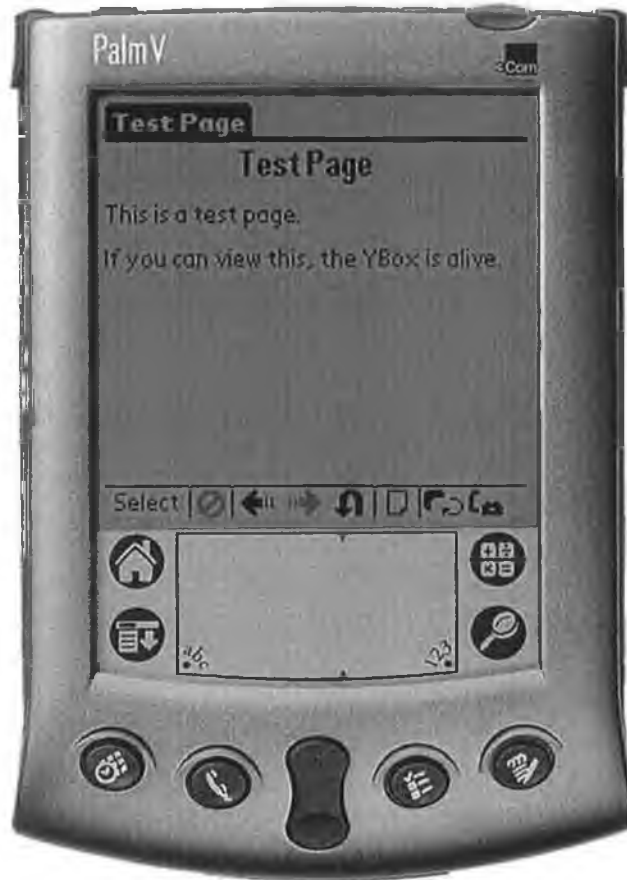


Figure 5.3. Test Page on a Palm V



Figure 5.4. Test Page on a WAP enabled Nokia mobile phone

The next part to testing content presentation is ensuring the YBox allows certain file types through the YBox without being transformed. This is required when the YBox is needed to support legacy HTML and images. An extract from the configuration file for this sample application is shown below: html, jpg and gif files are not transformed for this application.

```
<untransformed-files>
  <files type="html"/>
  <files type="jpg"/>
  <files type="gif"/>
</untransformed-files>
```

Figure 5.5 shows a sample document with an image. This is only possible if the YBox does not attempt to transform the image.

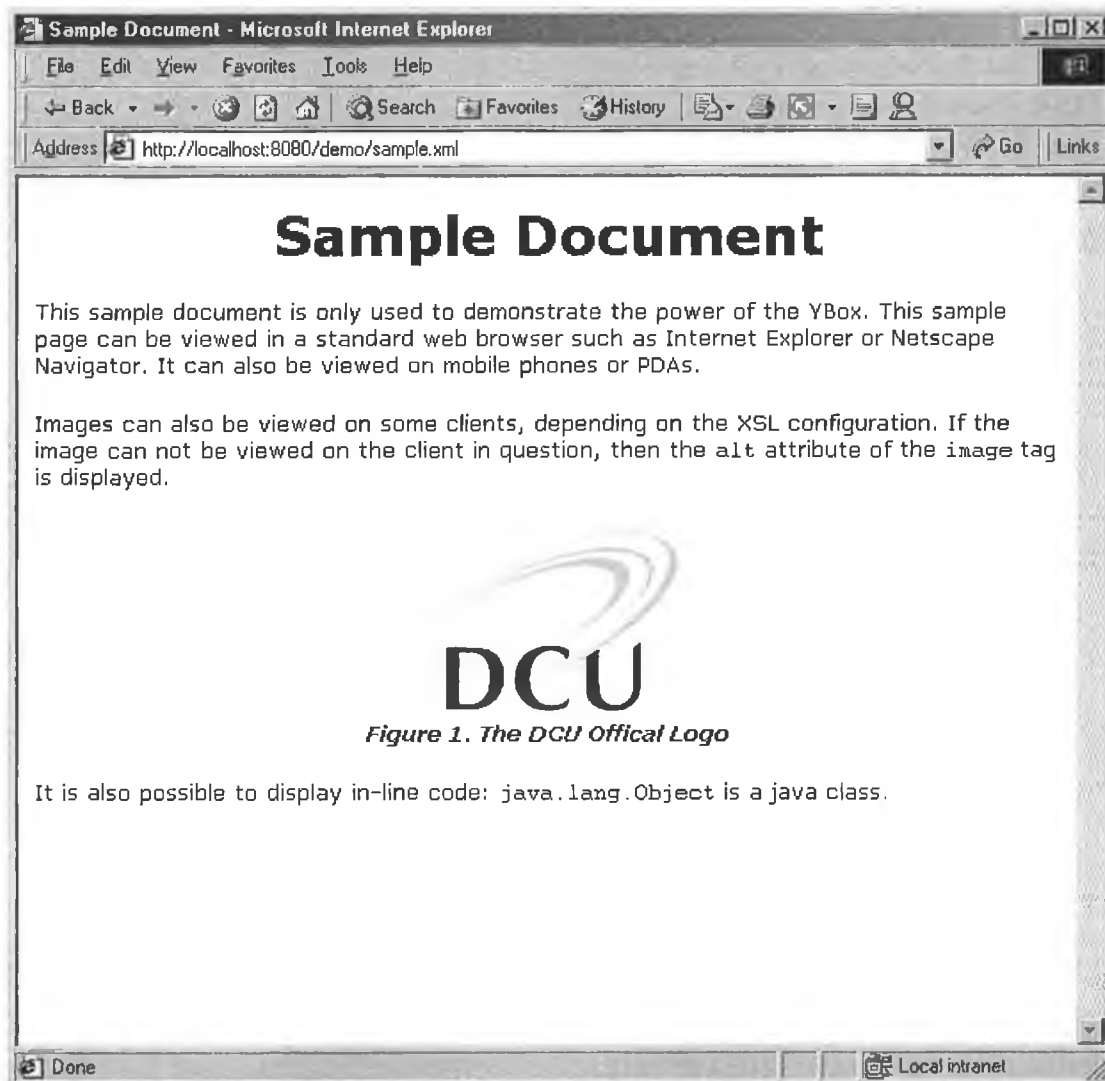


Figure 5.5. Sample Document with an image

The final part to the testing of content presentation is generating different content types. Figure 5.4 already shows WML generated for a mobile phone. This is a different content type, but it is still a Mark-up Language so it does not fully utilise all of the features of the YBox. Generating PDF documents using FOP tests the remaining presentation functionality in the YBox. Figure 5.6 shows the same content as Figure 5.5 except XSLT and FOP were used to generate the PDF in Figure 5.6. It can be seen from these two figures that the style can be preserved across all documents in a web application. The document can be saved locally or used for printing purposes by the user.

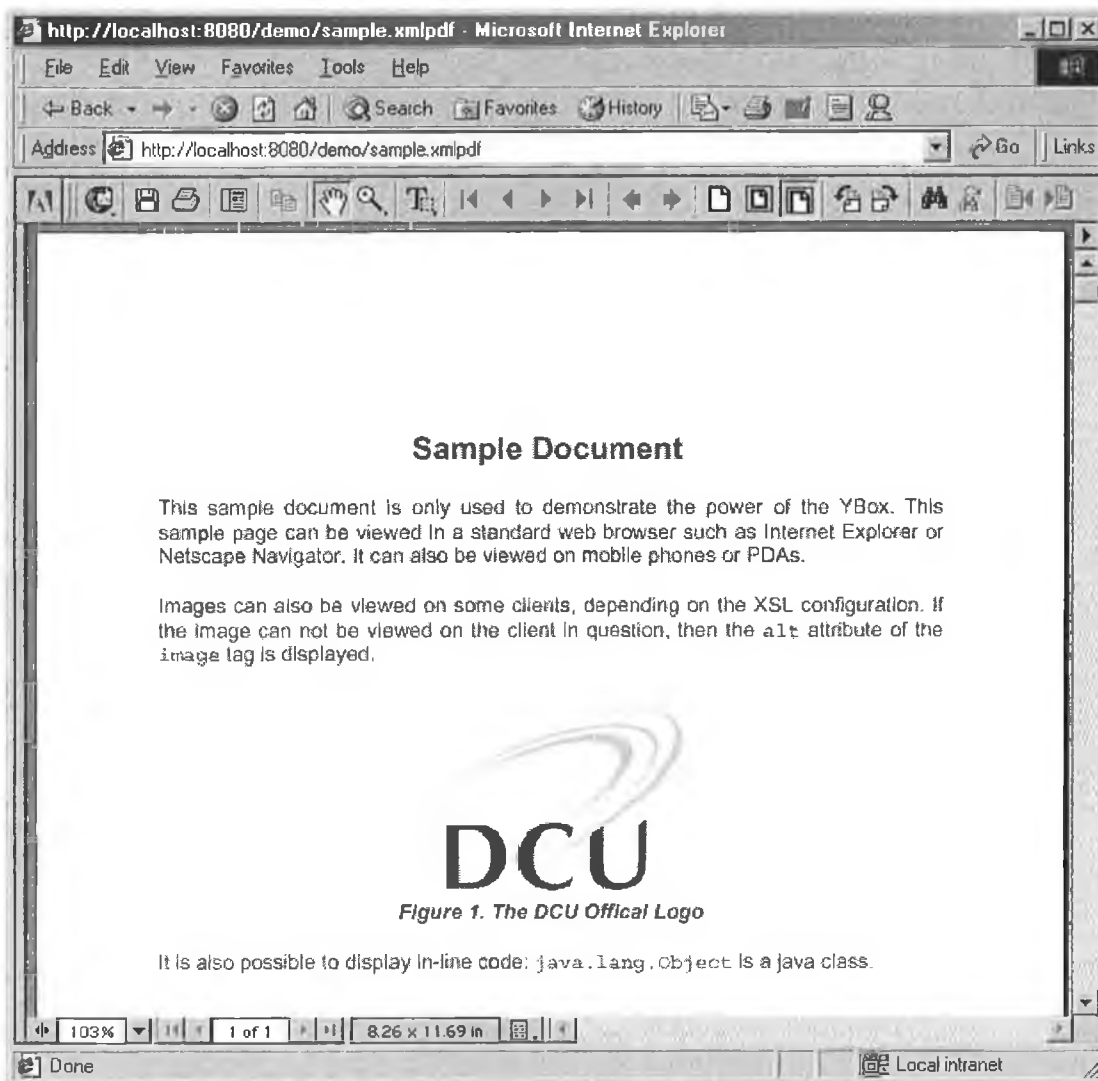


Figure 5.6. Sample Document with an image in PDF format

5.1.2. Testing Form validation

Testing form validation involves creating a sample form that tests all of the validation features the YBox supports. These features are:

- Basic type validation (strings, integers and decimals).
- Input text and whether it is required or not.
- Custom Classes.

The sample form has four questions. Each question tests a different feature of form validation in the YBox. Question 1 asks for the name of the user filling out the form. It is used to test a String input that is required. Question 2 asks the users for their height in meters. It is used to test a decimal number and it is not required. Questions 3 and question 4 ask for the users age and year of birth. They are combined to test custom class creation. Figure 5.7 shows the form as it is loaded for the first time.

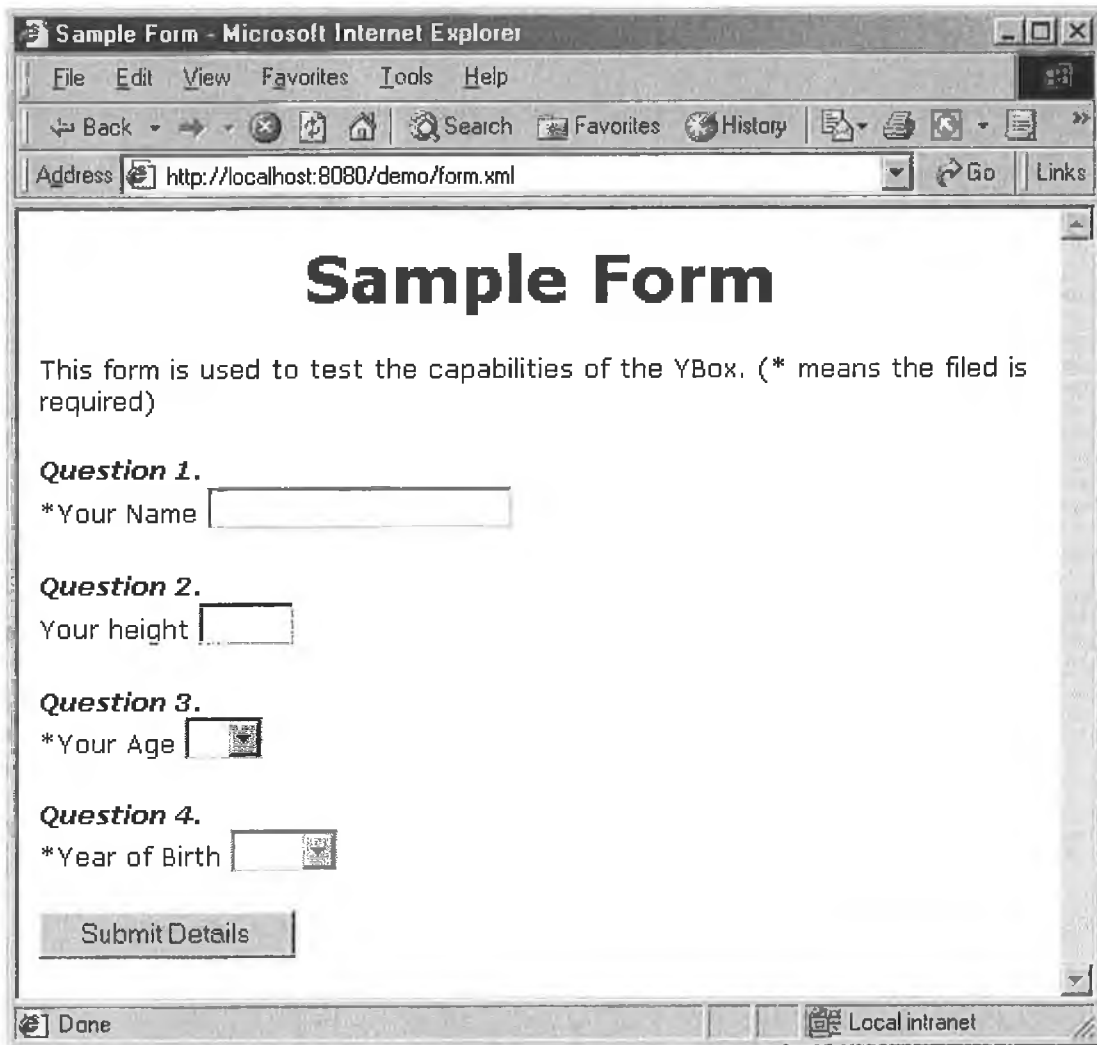


Figure 5.7. Sample form loaded for the first time

In the form in Figure 5.7 a field with a '*' beside it is required (the '*' is specified by the web application designer at design time – it is not a feature of the YBox). Therefore, only question 2 can be left blank for the form to validate correctly. The following sections show examples of form validation failure. The final section shows how the YBox and web application respond when the form is validated successfully.

Basic Type validation

All four questions in the sample form in Figure 5.7 are used to show how the basic types are validated. The basic types are strings, integers and decimals. Question 1 requires a String, question 2 a decimal and question 3 and 4 require integers. For question 3 and 4, the user is actually forced to enter an integer because of the combo

box. The YBox still validates the input and ensures it is correct. The source XML behind this form can be viewed in Appendix B.

One possible way the user can fill out the form is shown in Figure 5.8. Question 1 is correctly filled out because it is a String. Question 3 and 4 are also correctly filled out as the user is forced to choose integers for these questions. Question 2 is the only question incorrectly filled out.

Question 2 in the sample form is not very clear. The web application designer requires the user to enter their height in meters. But from Figure 5.8, the user has entered their height in feet and inches. This does not validate correctly as the inputted text is not a decimal value. Below is the XML source for question 2 (an extract from *form.xml*). From this extract, it can be seen the required type is a float (decimal). Also, the error message can be seen (“Please enter your height in meters!”).

```
<question>Question 2.</question>
<input-text
  ip_type="text"
  type="float"
  name="height"
  descr="Your height "
  size="5" required="false"
  value=""
  errorMsg="Please enter your height in meters!">
</input-text>
```

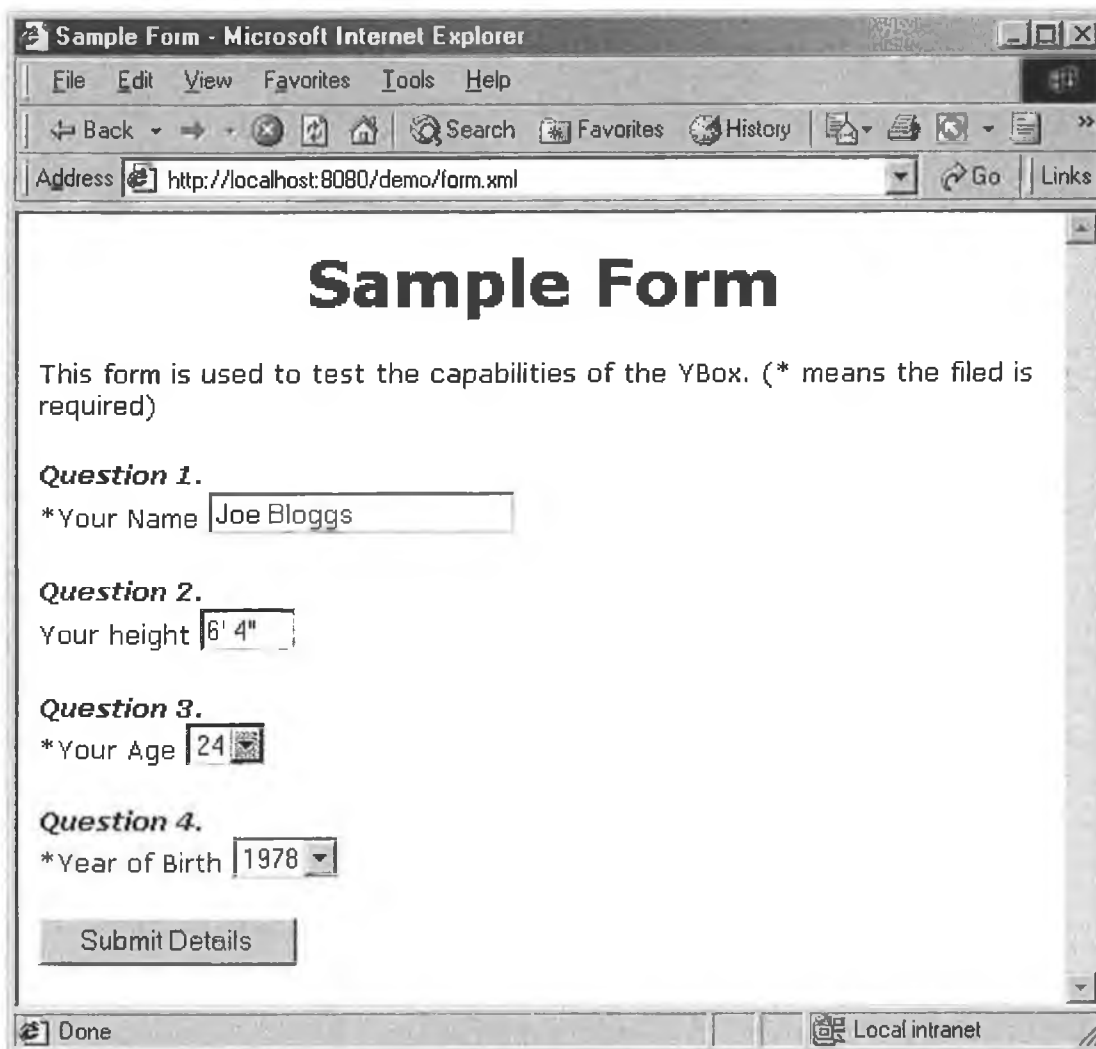


Figure 5.8. Completed for to show type validation

When the user clicks on “Submit Detail” the YBox fails on the form validation and the requested resource is not displayed. Instead, the same form is displayed again, with error messages on the fields that failed. The YBox also “remembers” the fields that validated correctly and redisplay them in the associated input box.

Figure 5.9 shows the resulting form. The YBox behaves as expected. More scenarios can be examined by repeating the process for all the input fields in the form. The YBox continues to redisplay the same form until it is validated correctly.

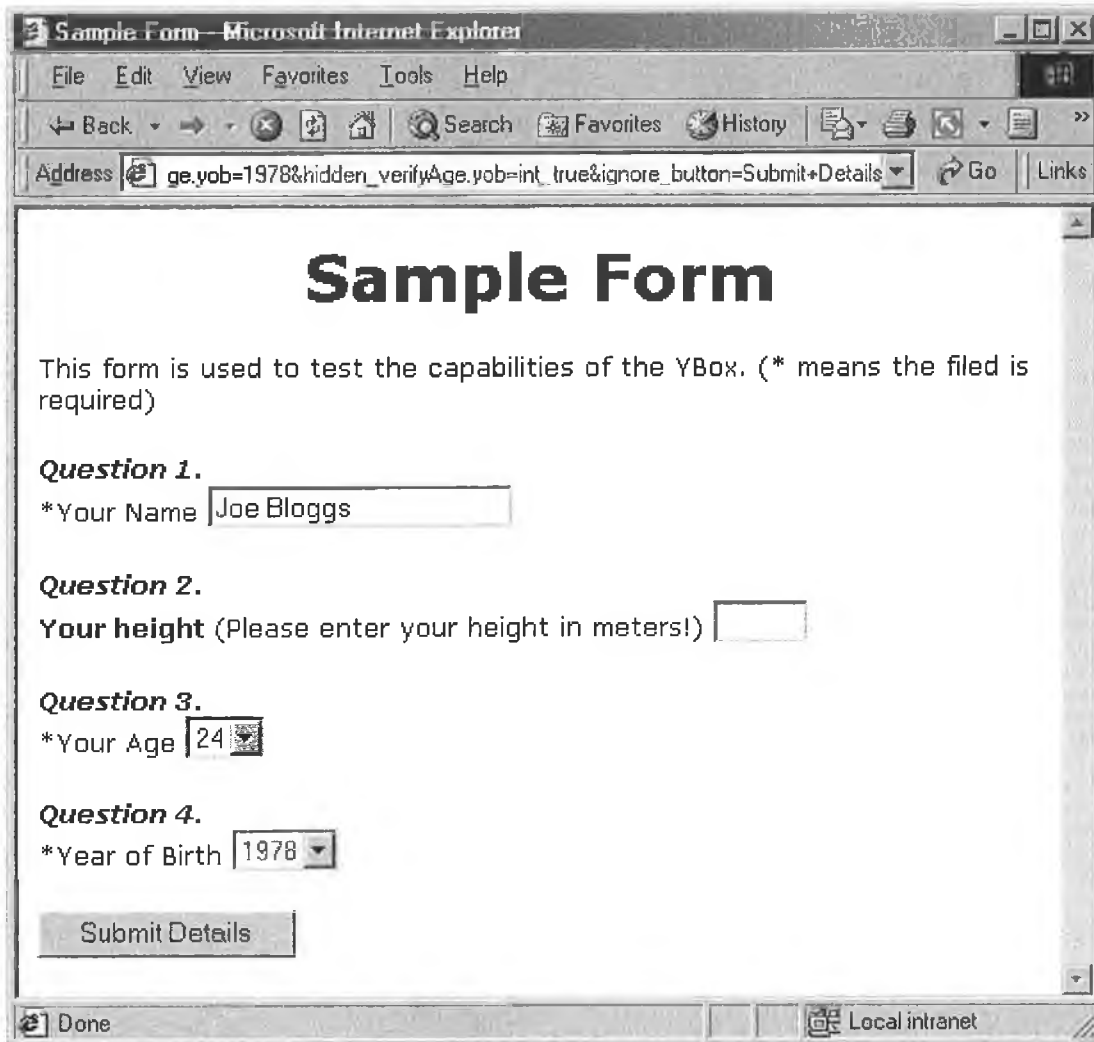


Figure 5.9. Type validation failure

Required field validation

From Figure 5.7 it can be seen that any question with a star (“*”) beside it, is required. Therefore, if the user leaves that input field blank, the field will fail to validate. Conversely, if the field is not required and the user leaves the question blank, that input field will validate correctly.

Figure 5.10 shows an example of this; question 1 is left blank even though it is required. This causes the form to fail validation. Question 2 is also left blank but it is not required. In this case the YBox will not display an error message beside question 2.

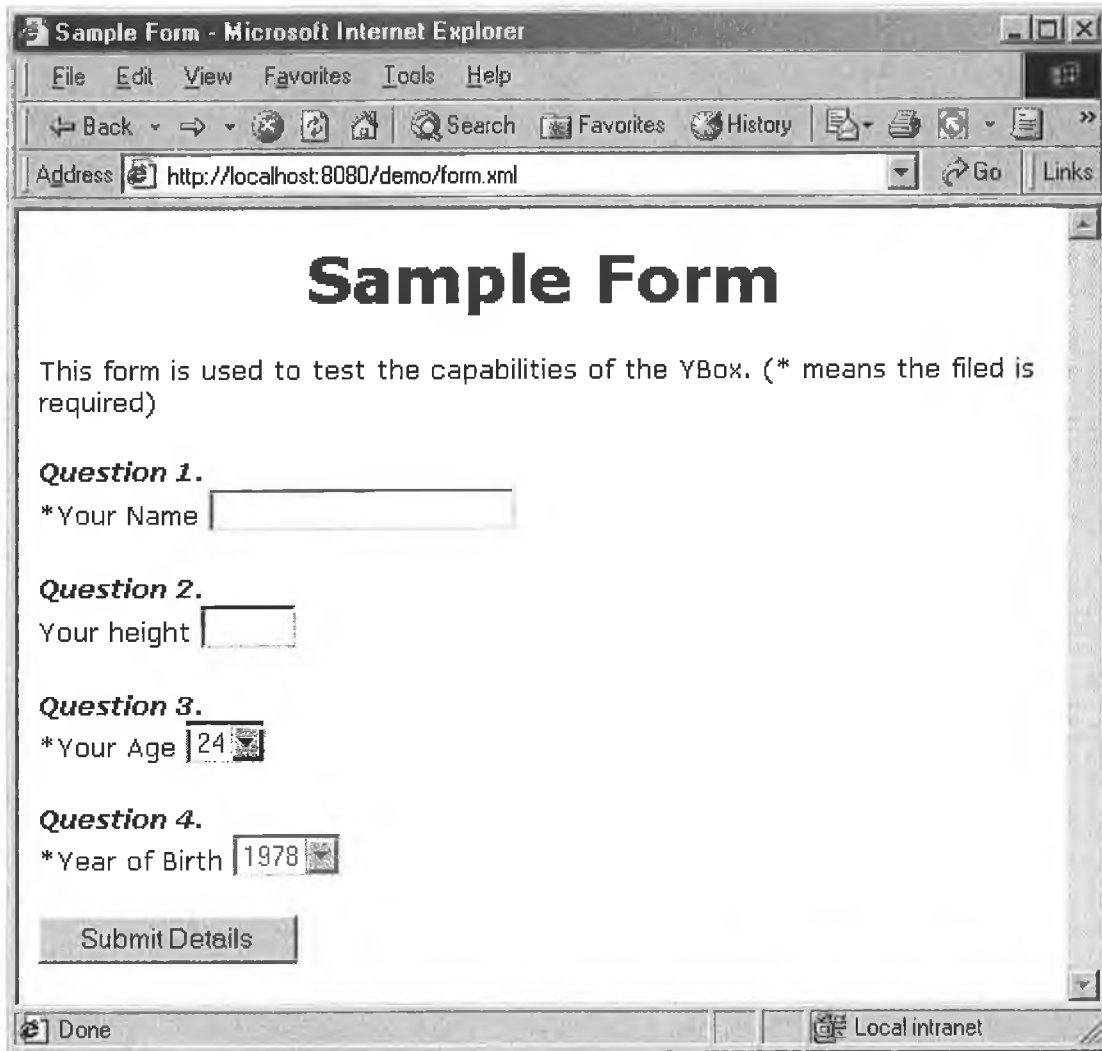


Figure 5.10. Required field left blank in incomplete form

Figure 5.11 shows the result of clicking “Submit Details” of Figure 5.10. Question 1 failed to validate correctly because its input field was left blank. The field is displayed again with an error message beside it.

Also shown in Figure 5.11 is the validation of question 2. This validation passes because the input is not required. The form will validate successfully even if question 2 is left blank. The value that gets passed to the requesting method is 0.0 (zero) if the field is left blank.

The screenshot shows a Microsoft Internet Explorer window titled "Sample Form". The address bar contains the URL "qe.yob=1978&hidden_verifyAge.yob=int_true&ignore_button=Submit+Details". The page content includes a title "Sample Form" and an introductory paragraph: "This form is used to test the capabilities of the YBox. (* means the field is required)". There are four questions: "Question 1. *Your Name (Please enter your Name!)" with an empty text box; "Question 2. Your height" with an empty text box; "Question 3. *Your Age" with a text box containing "24"; and "Question 4. *Year of Birth" with a dropdown menu showing "1978". A "Submit Details" button is located at the bottom left of the form area. The browser's status bar at the bottom shows "Done" and "Local intranet".

Figure 5.11. The required field fails validation

Custom Class validation

Question 3 and 4 combine to form an example of custom class validation. Question 3 asks the user for his/her age. Question 4 asks for the users year of birth. Both of these must be integers and the type validation feature of the YBox validates this.

There is an obvious relationship between the users age and the users year of birth; the age of the user added to the year of birth must give the current year or last year. This validation logic can be put into a class of its own and the YBox can use it to validate the inputs to question 3 and 4. The web application designer can put this logic into a VerifyAge class (see Appendix B for the source code) and use it to validate question 3 and 4.

Figure 5.12 shows the user filling out all fields in the sample form. All the field types are correct, so the form will successfully validate the input types. All the fields that are required are filled in, so the YBox will continue to perform the custom class validation.

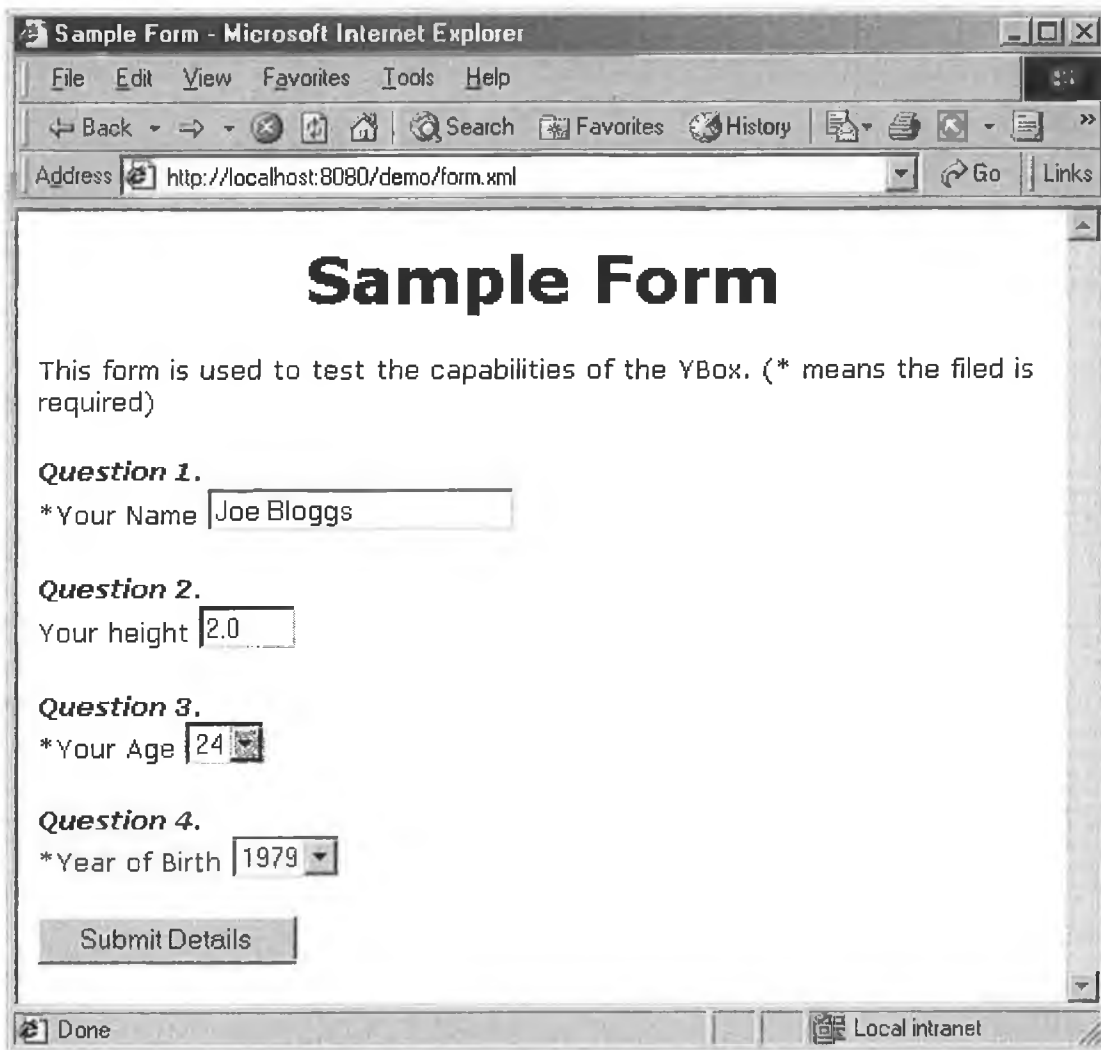


Figure 5.12. Custom class validation incorrectly filled out

The YBox will fail to validate the custom class. The user (Joe Bloggs) cannot be aged 24 and have a year of birth of 1979. For the form to validate correctly, the user must have a year of birth of 1977 or 1978. Figure 5.13 shows the resulting page when the user clicks “Submit Details”. Question 3 now shows an error message (Invalid age) beside its input field. This is message that is thrown with the exception from the VerifyAge class.

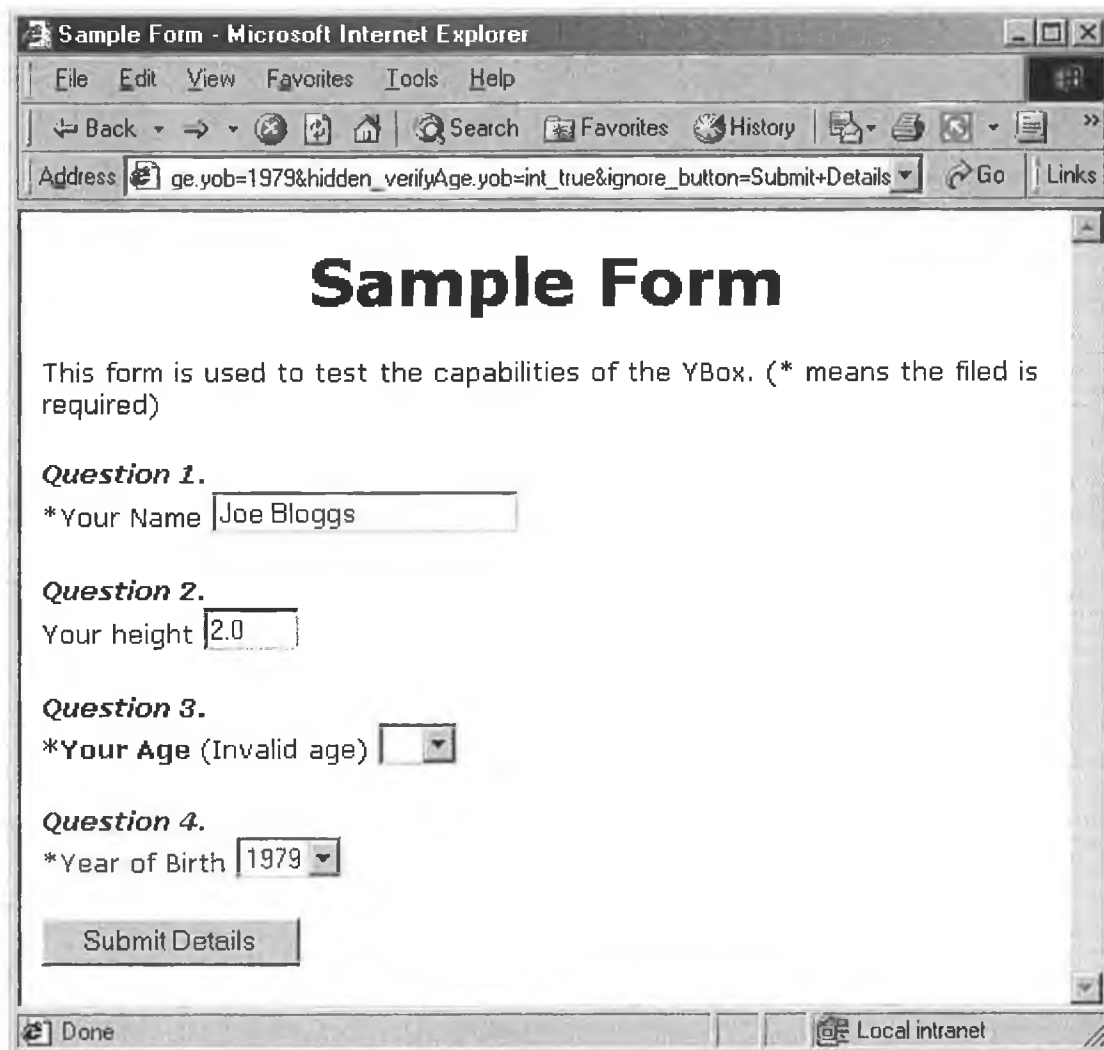


Figure 5.13. Custom class validation failure

Successful validation

Now that all possible incorrect validation scenarios have been dealt with, the case where the user enters all the correct information needs to be examined. Figure 5.14 shows the required user input for the form to validate correctly.

Figure 5.15 shows the result of the YBox validating the form correctly. The requested method of the SampleForm Servlet gets invoked and the result gets sent to the client. In this case, the web application displays what the user entered. If the sample was part of a live web application, then the SampleForm Servlet could store the user information to a database.

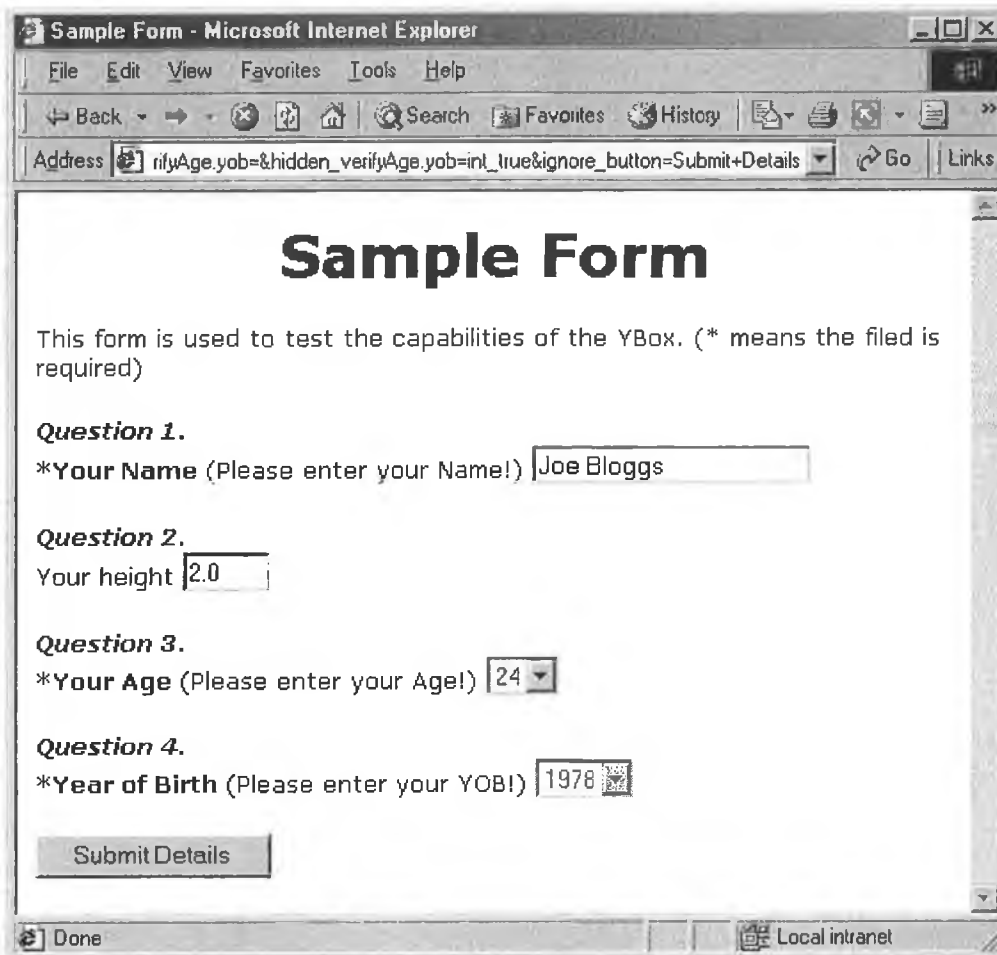


Figure 5.14. The correct input to the sample form

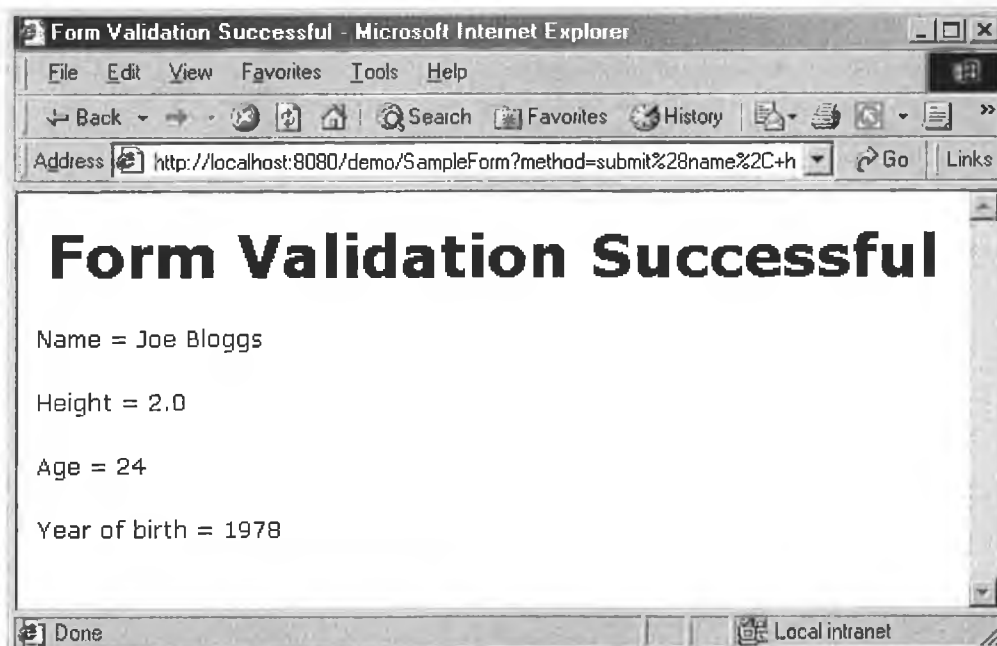


Figure 5.15. The resulting Servlet from the correct form

5.1.3. Testing Security

For the YBox to support security, the web application designer must extend the YBoxUser abstract class. This is done for the sample web application that is used for functionally testing the YBox. The YBoxUser used in the sample application does not require any authentication (to keep in simple). At login, the user only needs to supply a user name and a group he/she belongs to.

The login form can be seen in Figure 5.16.



Figure 5.16. The login to the sample web application

If the user does not have the required access permissions to view a resource, they should be shown the error page. This is specified in the YBox configuration file. An extract below show how this is configured.

```
<error-redirect>
  access_denied.xml
</error-redirect>
```

Any user that does not have access to the requested resource will get redirected to access_denied.xml in the web application. This page can be seen in Figure 5.17.



Figure 5.17. The page user sees when he/she is denied access to a resource

The YBox can implement security at two levels:

1. User-based security check.
2. Group based security check.

In this sample web application, the resource *session.xml* is to be protected. Only the user *liamf* and the group *ee553* can access the page. The YBox configuration file will look like the following.

```
<flat-file name="session.xml">
  <access>
    <group name="ee553"/>
    <person name="liamf"/>
  </access>
</flat-file>
```

The error page is shown in the following circumstances when the user requests *session.xml*:

- If the user has not logged in to the web application.
- If the user has logged in and they are not in the group *ee553* and are not *liamf*.

The user will get to view the requested resource if he/she is a member of *ee553* group or the user is named *liamf*.

5.1.4. Testing Session persistence

The web application designer must implement the `saveSession` method when he/she extends the `YBoxUser` abstract class. This method must store the session variables to a flat file or database. The YBox calls this method when the users session is invalidated. The session can be invalidated when:

1. The Servlet Container is stopped.
2. The user logs out.
3. When the session times out.

The `restoreSession` method of the `YBoxUser` object restores the session information when the user logs into the web application again. The `restoreSession` method must know which file or database to load in order to restore the users session variables. The web application designer implements this.

For functionally testing the YBox, a flat file is used to test session persistence. The objects in the session are stored in a `Hashtable` object. This object is serialised and written to the file when the session is invalidated. The user name is used as the name of the file because it is unique. When the user logs into the web application again, the session is restored for that user.

Figure 5.18 shows how session persistence is implemented in the test application. Steps 1 to 4 occur in sequence. Step 4 shows the user logging into the web application for the second time. The session information that was stored by the YBox is reloaded and the user can access his/her information once again.

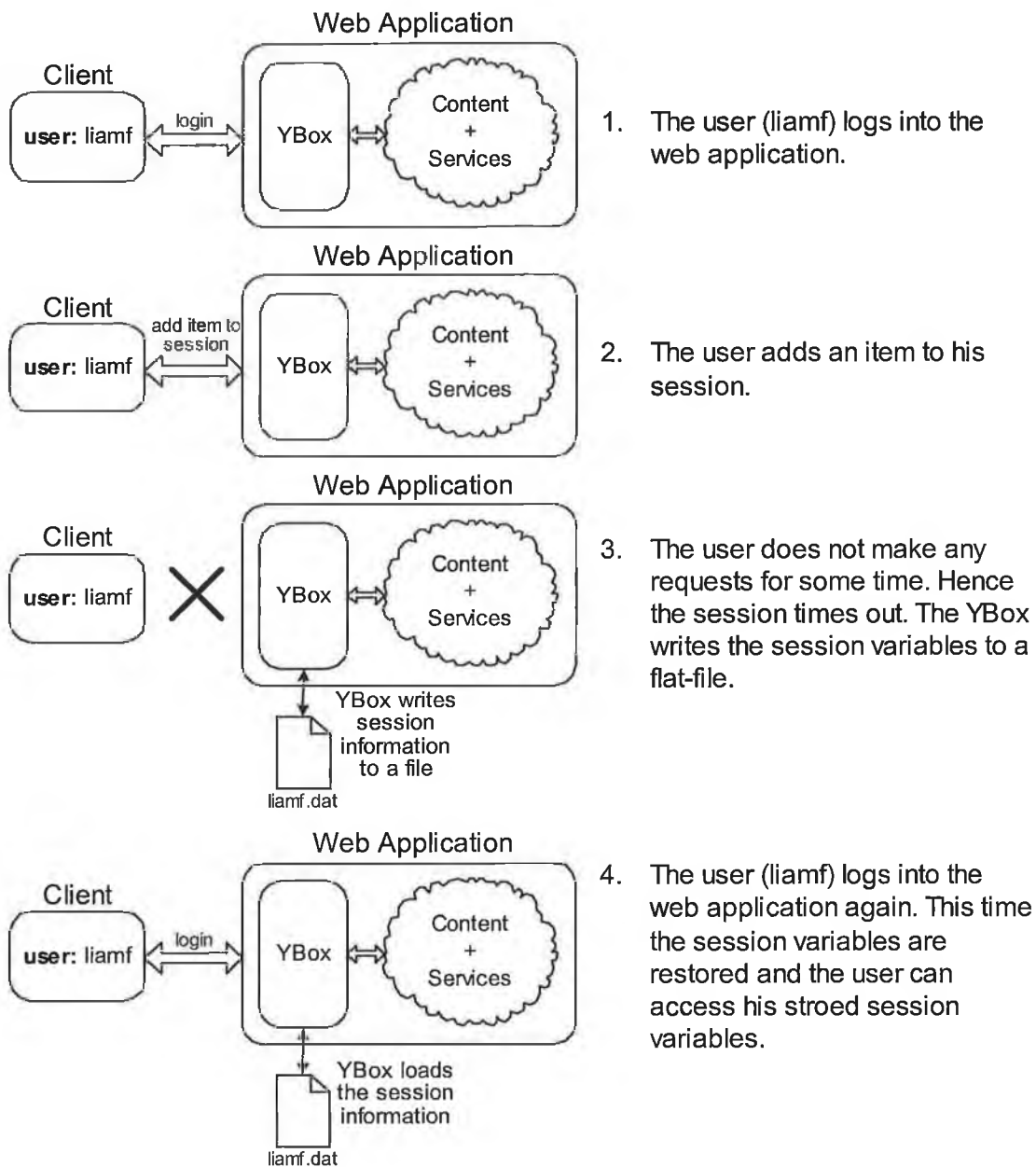


Figure 5.18. Steps involved in session persistence

There is a form in the sample application that demonstrates this capability. It is called *session.xml* and has a Servlet *SessionTest* associated with it. The resource *session.xml* is shown in Figure 5.19. It has two submit buttons. The first (“Store session variable”) will add string to the session in the web application. The second (“Get session variable”) will try to get the session variable with a name that matches the “Variable ID” input field.

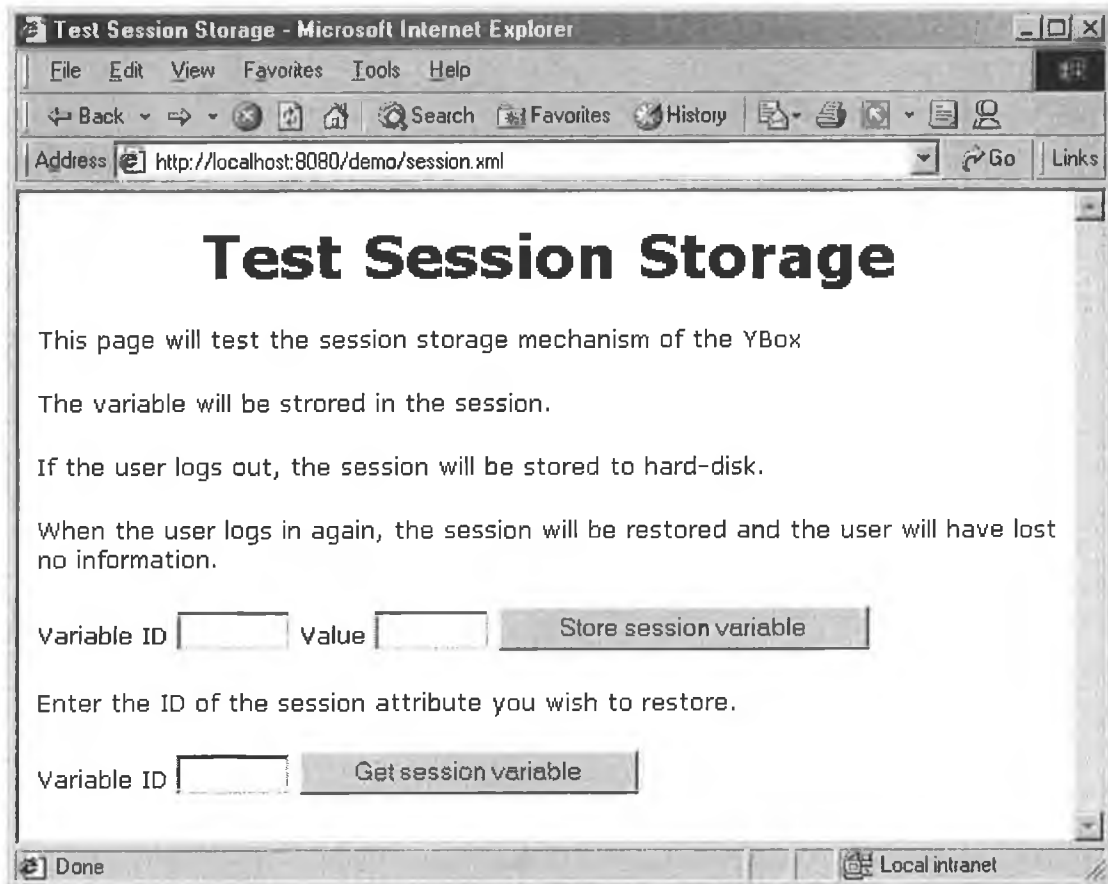


Figure 5.19. session.xml – A resource to test session persistence

To test this aspect of the web application, the user must enter a “Variable ID” and a “Value” and click “Store session variable”. This adds the variable to the session. This can be seen in Figure 5.20.

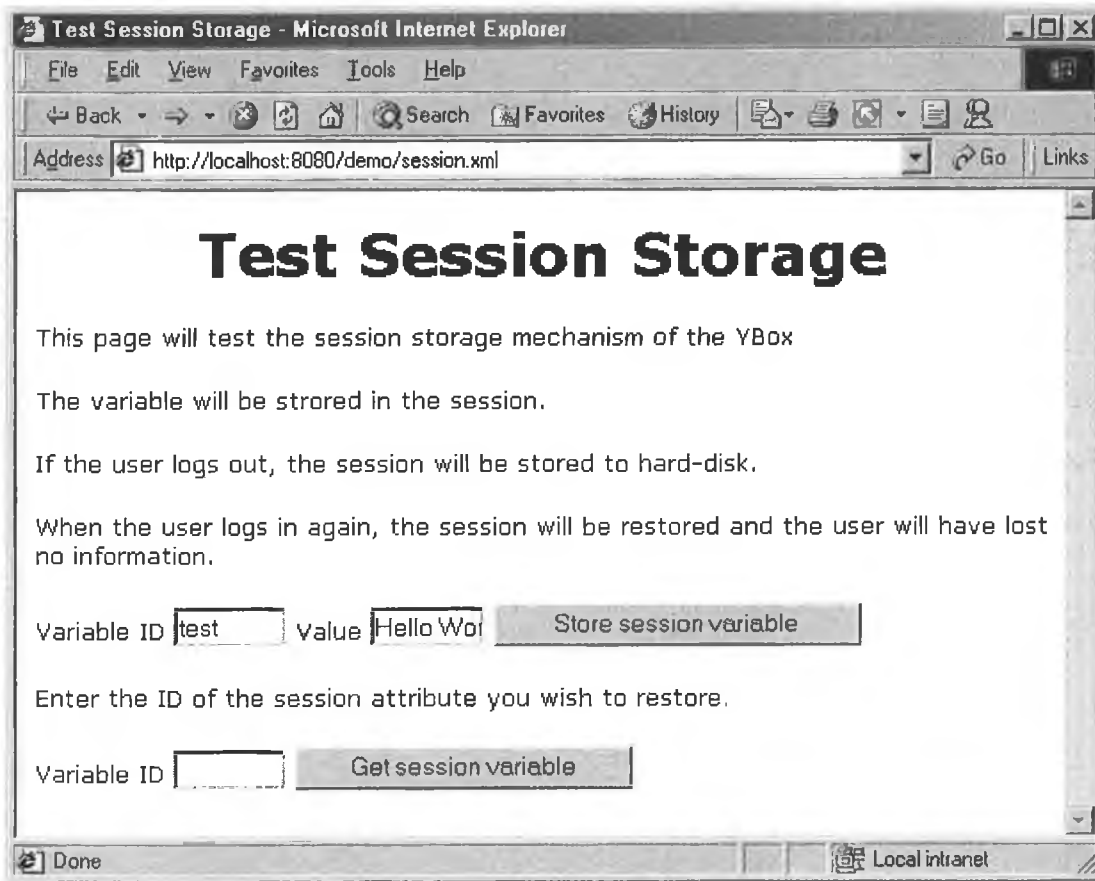


Figure 5.20. User enters test data into the input fields

To ensure the variable is added to the session, the user can enter the value “test” into the second “Variable ID” field and click “Get session variable”. The result can be seen in Figure 5.21. The session variable does exist and the Servlet displays its value.



Figure 5.21. The session attribute retrieved from the users session

To prove without doubt that this works, the user must try to retrieve a session variable that does not exist. The user enters “test1” into the second “Variable ID” field and click “Get session variable”. This session variable is not found. This can be seen in Figure 5.22.

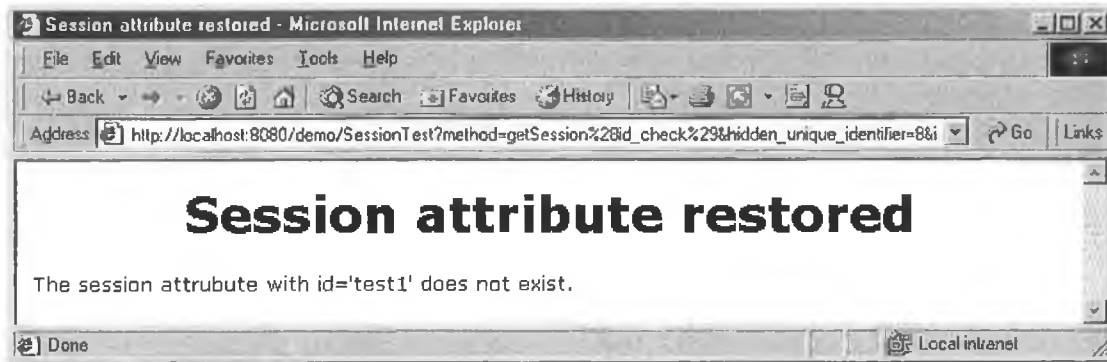


Figure 5.22. The session attribute not found in the session

To test all the capabilities of session persistence, the user must leave the web application idle for longer than the session timeout value in the deployment descriptor (*web.xml*). When this time is reached, the YBox will write the session variables to a file. The session variable with an ID of “test” and a value of “Hello World” will also be written to this file (this is also tested for the user logging out and the Servlet Container begin shutdown).

When the user returns to the web application after some time, they will not have access to *session.xml*, as the session will have expired (*session.xml* is protected by the security in the YBox; see section 5.1.3). The user must login to the web application again. This forces the YBox to load the flat file where the session variables are stored. These variables are reloaded back into the users session.

When the user re-visits *session.xml*, and tries to enter “test” into the second “Variable ID” field, the value “Hello World” gets returned (same result as Figure 5.21). This proves that session persistence works in the YBox.

5.2. Performance Testing the YBox

To benchmark the YBox, there are several comparisons that must be made. These benchmarks make it clear how a web application designed with the YBox compares to a web application designed without the YBox. These are only performance calculations and the ease of design is not taken into account.

This is the configuration used to perform the benchmarking test:

- Redhat 8.0 Operating System.
- Pentium III, 850MHz processor.
- 256MB RAM.
- Tomcat 4.01 Servlet Container.
- JMeter from Apache [26].

JMeter is an open-source performance measurement tool. It supports multiple threads, therefore can simulate multiple clients making a request at the same time. It can perform HTTP GET or POST methods on any resource. It measures the time taken for each request to be served and saves this information to a log file.

The sample web application designed with the YBox is compared to a web application designed without the YBox. They are compared under the following tasks:

- A increasing number of users making a fixed number of requests for a static file (HTML only) in succession.
- A single user making hundreds of requests for a static file (XML and HTML) in succession.
- A increasing number of users making a fixed number of requests for a static file (XML and HTML) in succession. The test is repeated for a dynamic resource.

5.2.1. HTML requests

The first testing on the YBox is performed with a static HTML file. The web application with the YBox and the web application without the YBox will serve a static HTML file. The web application with the YBox will not have to perform XSLT, therefore the performance between the two applications should be comparable.

Figure 5.23 shows the results of this test (Note: A bar chart is used as the results were too similar for a line graph). It can be seen that the performance is almost identical.

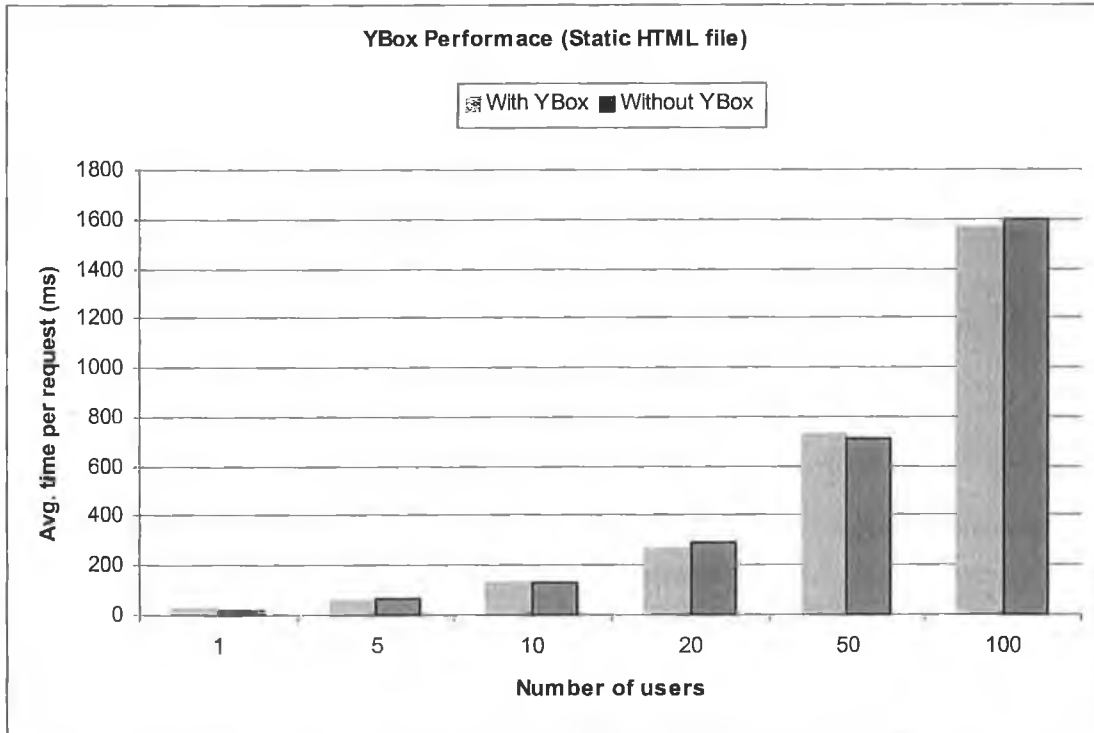


Figure 5.23. YBox performance with a static HTML resource

5.2.2. Single User, multiple requests

This section describes how the sample web application deals with a single user requesting a resource over time. Figure 5.24 shows how the sample application with the YBox compared to the same application without the YBox. The resource that is being requested is a static file in both cases. The major difference is the YBox must transform the XML source into HTML.

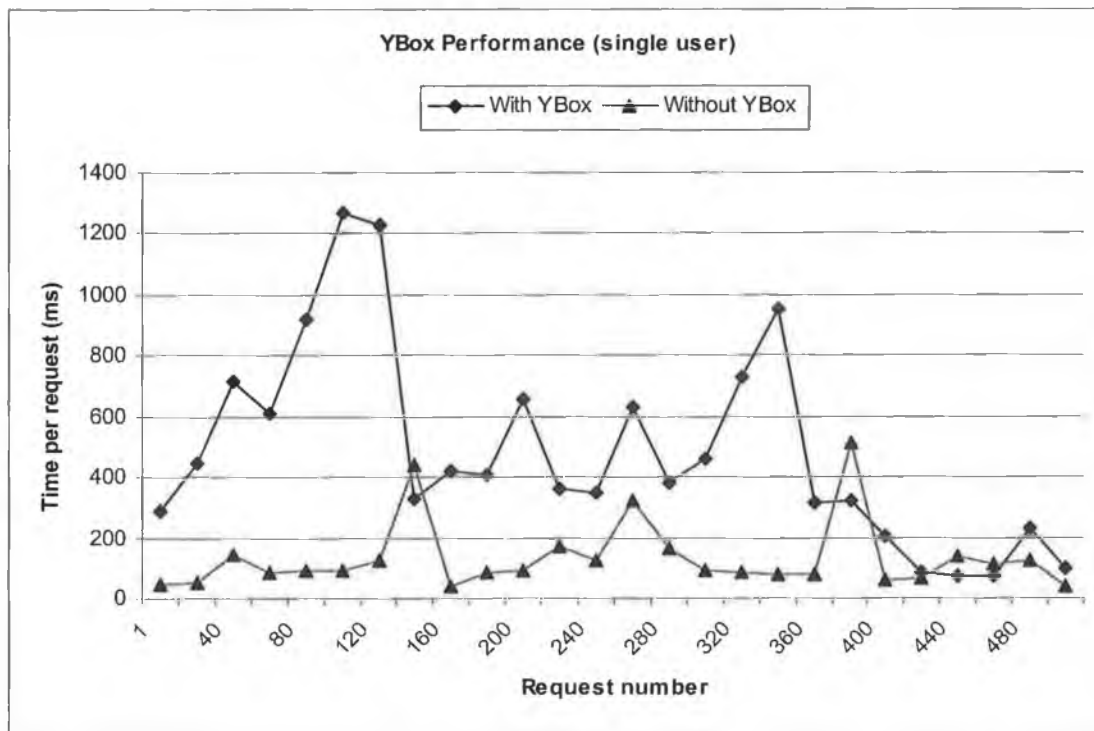


Figure 5.24. A single user making 500 requests for a single resource

On average across the 500 requests in Figure 5.24, the web application without the YBox performs 3.2 times better than the web application with the YBox. The reason for this is the web application with the YBox must use Xerces to process the XML document and use Xalan to transform the XML into HTML. This accounts for the difference in performance.

Most web application will deal with more than one user, therefore the remainder of the performance testing is performed with an increasing number of users.

5.2.3. Increasing number of users, fixed number of requests

The previous section discussed the performance of the YBox when a single user makes several requests. This section is interested in the performance of the YBox when it is tested with an increasing number of users. This type of testing ensures that the YBox can handle large numbers of users.

Figure 5.25 shows the comparison between a web application tested with the YBox and without the YBox. This comparison is when the web application is serving a static file. In the case of the web application with the YBox, this will be an XML file

that must be transformed. For the web application without the YBox, the file will be a flat HTML file. It can be seen that the web application without the YBox performs 3 times better (on average) that the web application with the YBox.

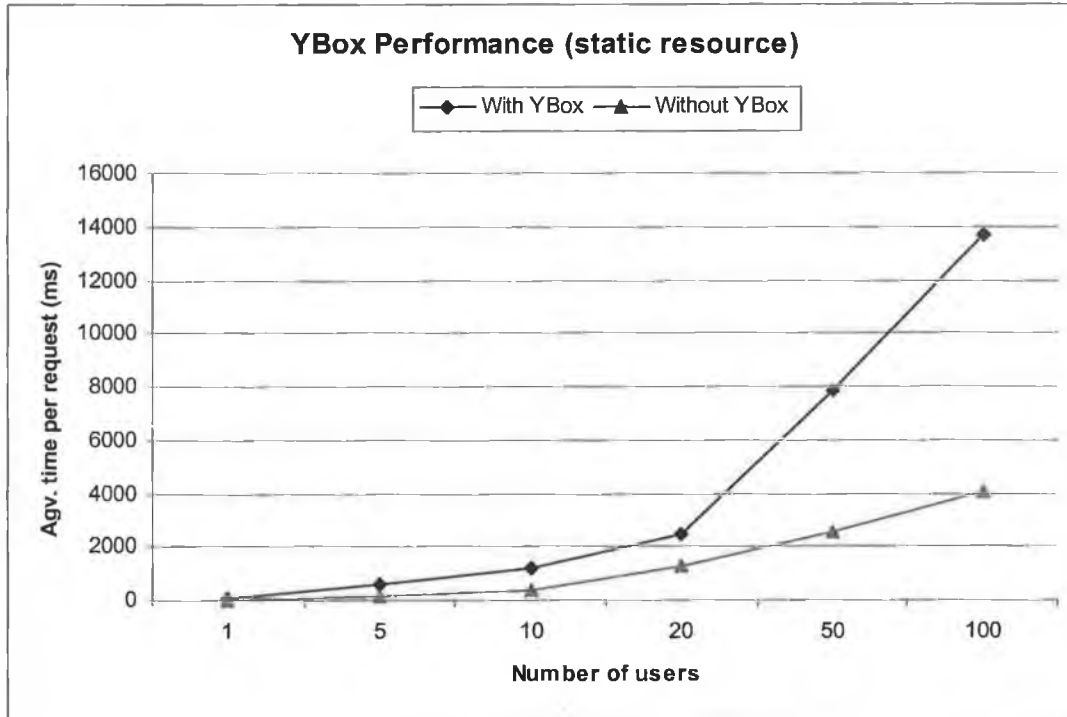


Figure 5.25. Web application performance with a changing number of users (static resource)

Figure 5.26 shows a similar comparison, but this time the resource is dynamic (a Servlet). The web application with the YBox is performing similar to that in Figure 5.25, but the web application without the YBox is not performing as well (relatively). This shows that the YBox performs better when dealing with dynamic content. On average, the performance of the web application without the YBox is only 20% better that the web application with the YBox.

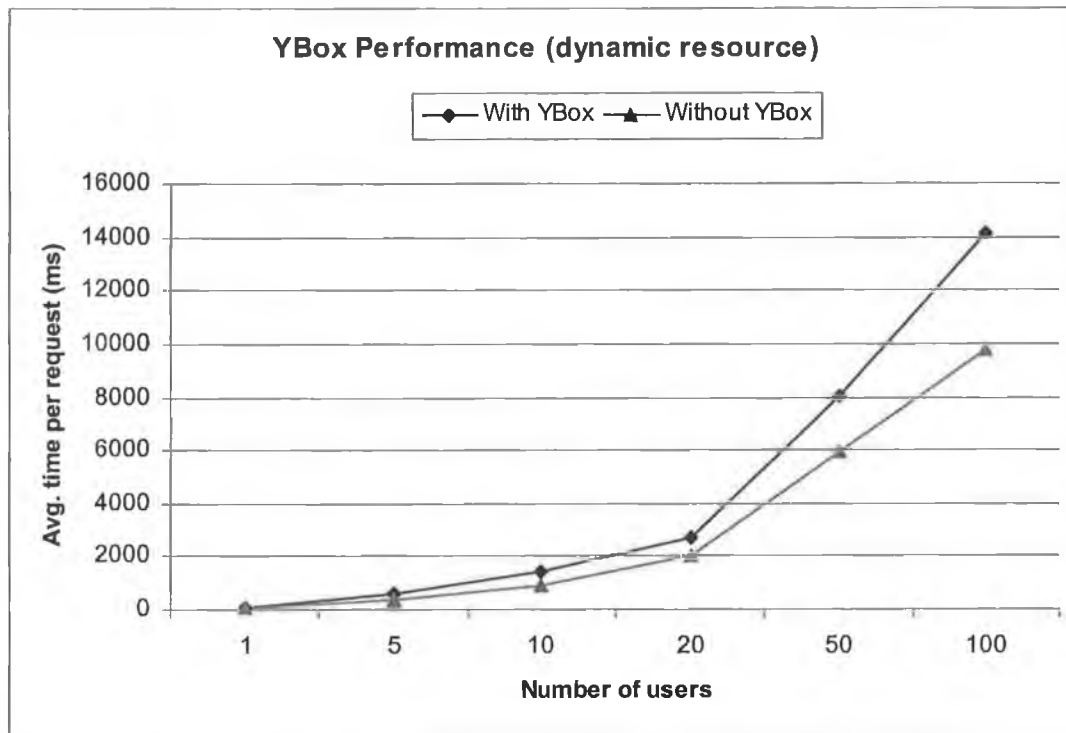


Figure 5.26. Web application performance with a changing number of users (dynamic resource)

5.3. Error handling with the YBox

The YBox must be able to handle errors that occur during its lifecycle. It must react to these errors in a predictable manner and not result in having the Servlet Container shutdown. The YBox must also log these errors for web application designers to examine in detail.

The YBox uses the recommended Java technique of throwing and catching exceptions where possible. Using this technique, the YBox will never exhaust the resources of the Servlet Container as it deals with problems gracefully.

There are two types of errors that can occur in the YBox; configuration errors and runtime errors. Configuration errors occur due to mistakes in the YBox configuration file. The YBox or the requested resource getting into an unstable state causes Runtime errors.

5.3.1. Configuration Errors

The YBox easily deals with configuration errors. If the YBox detects a configuration error, the web application associated with the configuration file will not start. The YBox will throw an exception and log the error message to the log file associated with the web application.

A configuration error has been deliberately placed in the YBox configuration file. An ending tag is missing from one of the file types not to be transformed by the YBox. This can be seen in an extract from the configuration file below.

```
<untransformed-files>
  <files type="html"/>
  <files type="jpg"/>
  <files type="gif"> ← missing
                        end tag
</untransformed-files>
```

When the web application is started, the YBox will throw an exception when it attempts to load the configuration file. This exception is actually a `ValidationException` thrown by the JAXB runtime library as it tries to marshal the configuration file. The exception is logged to the web applications log file where it can be viewed by the web application developer. This exception will actually give the type of error and the line in the configuration file that caused the error.

The sample web application will not be started as a result. If a user tries to access the web application, Tomcat will realise that the web application has not been started and will display the appropriate error message to the user. This error message can be seen in Figure 5.27.

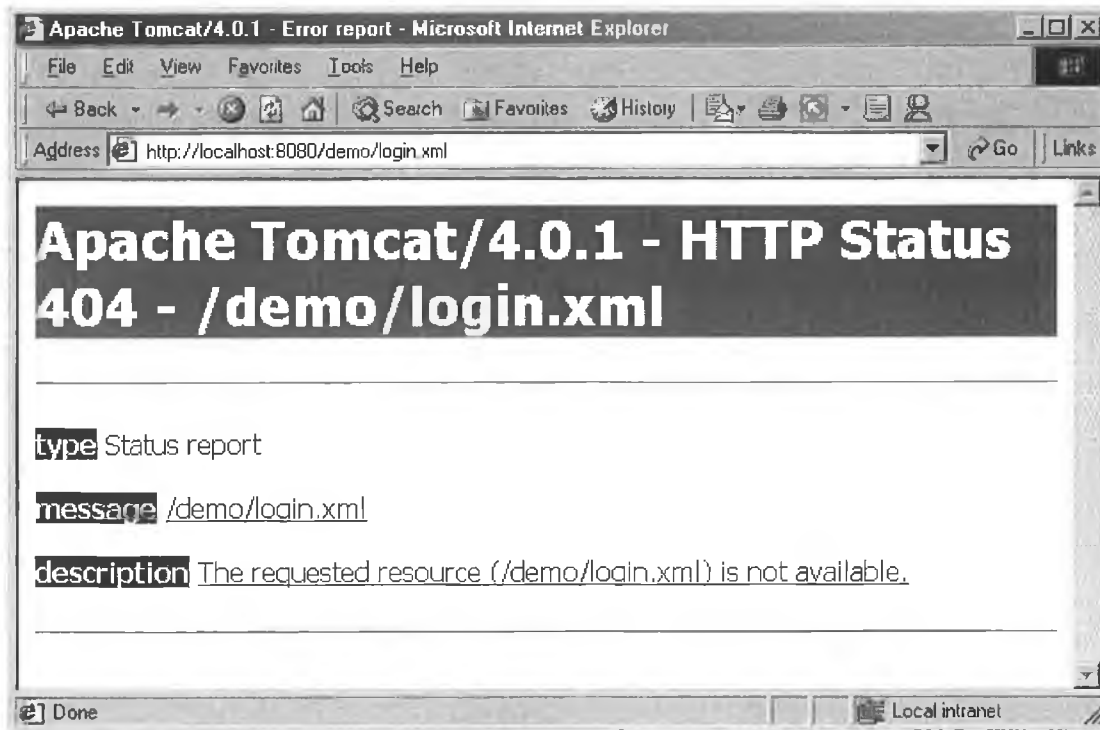


Figure 5.27. Error message when web application is not loaded

5.3.2. Runtime Errors

Runtime errors are difficult to predict. These errors depend on how the users use the web application and how the web application designer has designed the application. The YBox must be able to withstand a “hacker” attempting to access a resource by modifying the URL in the address bar of a web browser. It must also be able to withstand errors in the design the web application.

The `ie.dcu.liamf.ybox.mgt.Error` class deals with all runtime errors in the YBox. When the YBox catches an exception, it invokes the `error` method of the `ie.dcu.liamf.ybox.mgt.Error` class.

Incorrect number of parameters

The YBox flags an incorrect number of parameters error when it detects too many or too few parameters for the requested method. From the source XML the YBox know exactly how many parameters (input fields) it is expecting. It displays an error page if an incorrect number of parameters are detected with the request.

A “hacker” modifying the parameters in the URL or a mistake in the design of a web form can cause an incorrect number of parameters, and are treated in exactly the same manner by the YBox. The YBox checks all the parameter in every request and ensures all are valid. Take the following example: A “hacker” modifies one of the parameters in the URL attempting to bypass the login page. This is shown in Figure 5.28 – an extra parameter called “test” with a value of “blah” is entered in the address bar.

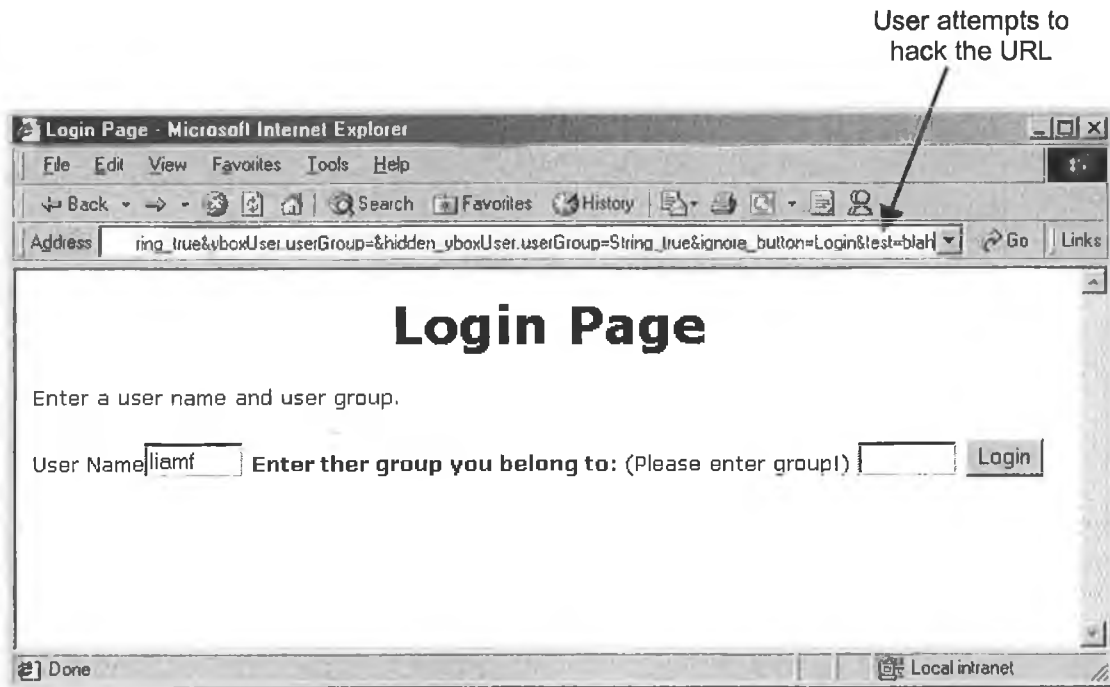


Figure 5.28. User attempting to hack the web application using the URL

The YBox detects this error and does not invoke the requested method. This error could also have been a result of a form design error. The error page that results is shown in Figure 5.29.

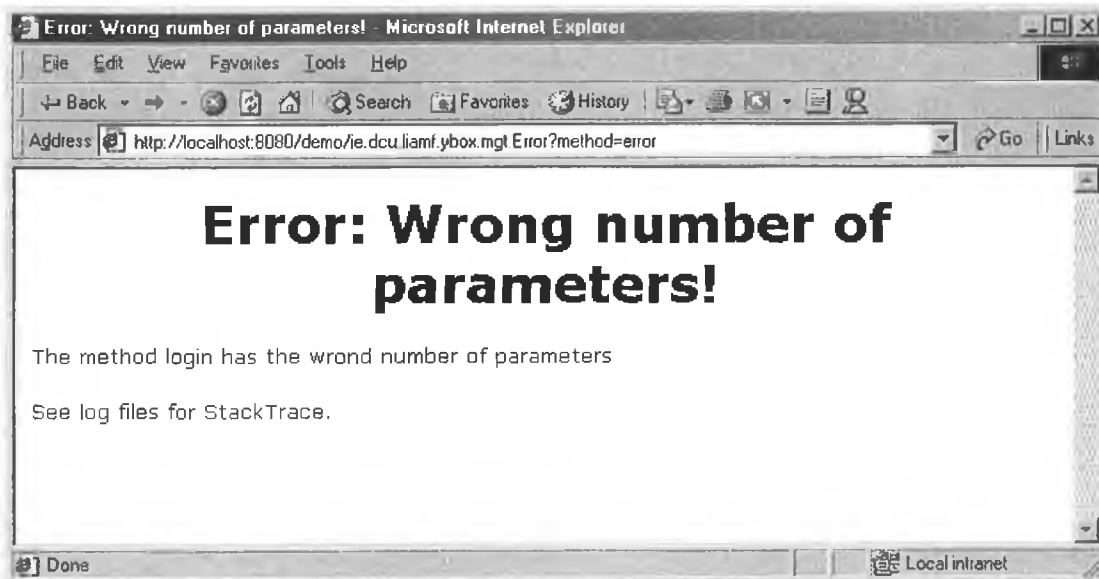


Figure 5.29. Error message associated with the wrong number a parameters

File does not exist

This error is displayed when the requested resource does not exist. If the requesting user does not have access to the directory they are requesting, then they will be shown the “*access_denied*” page by the YBox (see section 5.1.3). If the user has the required permissions, then they will be shown an error message explaining the resource could not be found. This can be seen in Figure 5.30.



Figure 5.30. File not found error

Method invocation errors

When the user is requesting a method of a Servlet there are three errors that can occur:

- The method may not be registered in the YBox configuration file.
- The requested method footprint does not exist. This occurs when the parameter types do not match the footprint of the method to be invoked.
- The method name may not specified in the requested URL.

When the method is not registered in the configuration file the error message in Figure 5.31 is displayed. When the YBox displays this message, either the web application designer has made a mistake or the user has tried to hack the URL.



Figure 5.31. Error message displayed when the method is not registered

Figure 5.32 shows the error message displayed when the requested methods footprint does not exist. The method `login` of the `Login Servlet` has a footprint that looks like:

```
login(String username, String group)
```

If the user requests a method with a footprint that looks like:

```
login(String username, int group)
```

the YBox will show an error message. A method does not exist in the `Login Servlet` where the `group` is an integer.



Figure 5.32. Error message displayed when the footprint does not match

Lastly, if the method name is not specified in the URL, the YBox will also display an error. This error can be seen in Figure 5.33. The error can be due to web application designer error or URL hacking.

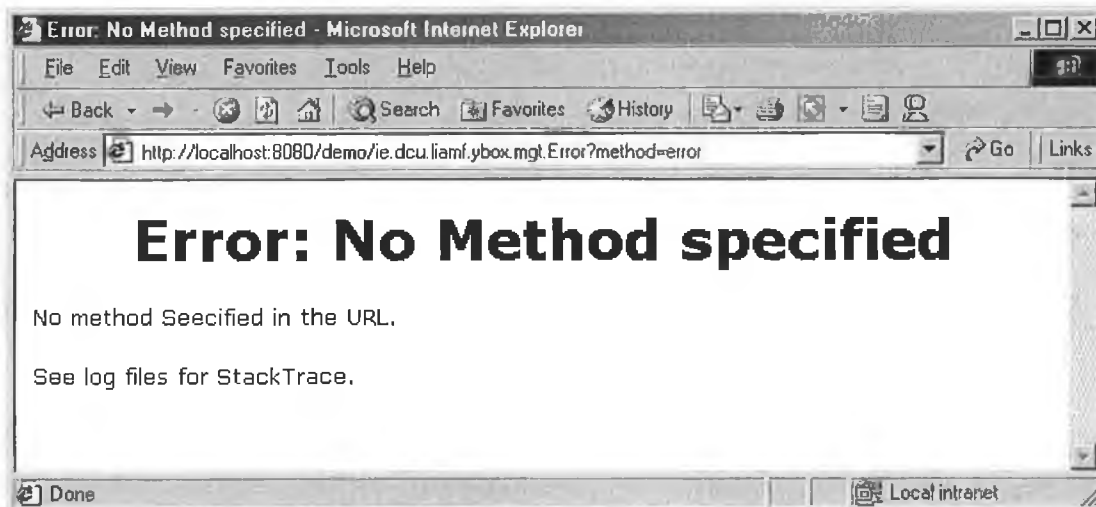


Figure 5.33. No method specified in the URL

5.4. Summary

This chapter describes a sample application that was used to functionally test and stress test the YBox. The chapter tests each feature of the YBox in great detail and shows how the web application combined with the YBox, behaves in certain circumstances. The performance of the two web applications are compared, one

designed with the YBox, the other without. These performance tests show the YBox has an impact on performance, but the impact is not significant (especially when dealing with HTML files). Finally, error conditions are explained and how the YBox deals with them. These error conditions could be due to incorrect configuration of the YBox or users attempting to hack the web application behind the YBox.

Chapter 6 - Conclusions and Further Research

As a front-end framework, the YBox complements the J2EE architecture and enables community based web applications to be designed more easily. Since the growth of Linux in the server market, it is important that web application servers are written in a platform neutral manner and not only aimed at the Microsoft platform. Hence, the YBox is an ideal choice for this.

The ability of the YBox to support XML as a primary data source ensures that the YBox framework is future-proofed, and as a framework, this is very important. This means a web application designed using the YBox is also future-proofed as it can support new clients and content types by introducing a new transformation. Therefore the YBox will have a significant place in web application design for years to come.

This research has implemented a framework that separates the presentation logic from the business logic inside a J2EE web application. This is a desirable feature of a web application design framework. This framework allows business objects to be separated from presentation code (JSPs).

The YBox framework has implemented an innovative security mechanism based on the security of a Unix file system. Security on the Unix operating system is tried and tested and the security framework implemented in the YBox is based on similar principles. No other design framework has a security system similar to this, hence making the YBox unique in this area.

The form validation features implemented in the YBox are an advanced Object Orientated approach. The new approach to form validation has not been implemented in any framework previously. The model used builds on the HTTP protocol. The HTTP protocol uses the POST [43] method to send form data to a Servlet in a web application. In the Servlet specification, this will invoke the *doGet* method of the requested Servlet. The YBox actually invokes a requested method of the Servlet directly. The input fields to the web form are used as the input parameters to the Servlet method. Groups of input fields can also be validated using a single object. The

Object Oriented approach to form validation validates the originality of the YBox framework.

6.1. Future Research

As is true for many works of research there are many opportunities for extensions and refinements to the methodologies presented. In the case of the YBox framework, the proposed developments focus on three key areas:

- Performance of the YBox.
- Reuse of existing open-source frameworks to extend the capabilities of the YBox.
- Support for new XML schemas (other than DTDs).

6.1.1. Performance of the YBox

The “bottle-neck” in the YBox implementation is the transformation of XML into client specific content (HTML, WML, PDF, ... etc) and the manipulation of the XML source as shown in section 5.2. This is the XML processing and XSL transformation of response manipulation in the YBox. This response processing is implemented using XML technologies from Apache (Xerces and Xalan).

Xerces is used by the YBox to load the XML content into a DOM object and modify the XML tree if the resource is a web form. The YBox must insert error messages if the form has not validated correctly. It must also insert a unique identifier as a hidden input field (see section 4.4.5). This implementation has only been tested using the Xerces XML processor. XML processors other than Xerces should be used and the result should be evaluated using identical testing techniques described in section 5.2.

The processors that should be used for testing are:

1. XML4j [39] from IBM.
2. Oracles XML developers kit for Java [40].
3. JAXP from Sun Microsystems [41].

The XSLT stage of the YBox is implemented using Xalan from Apache. Xalan is used by the YBox to transform XML source into client specific content. This

implementation has only been tested using Xalan. XSLT should also be tested with other transformation engines such as:

1. Saxon [36].
2. XSLJIT [37] from DataPower Technologies.
3. XT [38] written by James Clarke.

The results from these changes should be tested and benchmarked and compared to the results obtained in section 5.2.

6.1.2. Reuse open-source frameworks

Many open-source organisations are researching ways of improving the methods for implementing web applications. One such organisation is Apache. Struts [17] from Apache is an open-source implementation of a front-end framework to aid in the design of large web applications. The reasons Struts should be incorporated into the YBox framework are:

1. Struts has an advanced JSP taglib [44] that could be reused in the YBox.
2. Struts uses the **Model-View-Controller** (MVC) design paradigm for its content provision. The YBox could take advantage of this.
3. The YBox framework could take advantage of the localisation features that are supported in Struts.
4. The source for Struts is freely available so changes to the source of Struts can be made.

Figure 1.1 shows a web application with the YBox and Struts combined. The YBox is still the only entry point to the web application. The diagram describes how Struts should be integrated with the YBox framework. This new framework can take advantage of the features from both frameworks.

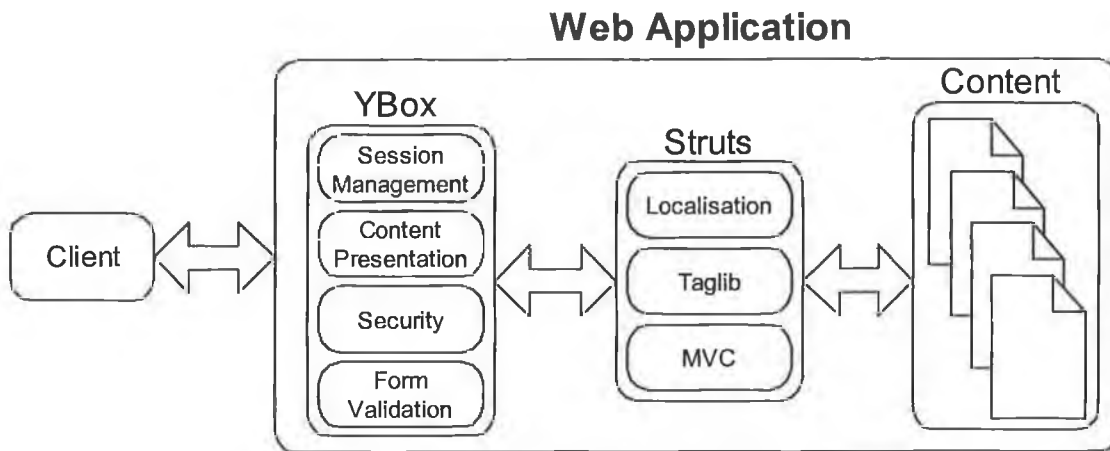


Figure 1.1. A Web Application with the YBox and Struts combined

Using Struts also removes any duplication of effort as the same functionality does not get implemented in the YBox and in Struts in the future.

6.1.3. New XML Schema

The YBox must support the latest XML schema language [42] as well as DTDs. XML schema are more advanced than DTDs as they introduce features that provide functionality above and beyond what is provided by DTDs. A schema is an XML document that defines the content and structure of one or more XML documents.

Schemas offer some very important functions:

- Content Model Validity - This ensures that the element hierarchy and document structure are correct. It checks to make sure that elements are ordered and nested correctly (much like DTDs).
- Data-type Validity - This ensures that element and attribute content adheres to the defined data-type. A data-type can define a scope for legal values as well as define a base type such as integer, decimal or string.
- Extensibility - Schemas allow for greater generalities in terms of describing the structure of the document, which in turn, allows for greater control in the creation of the XML document and in reusability of the schema mark-up to be utilized in other areas.
- Namespaces - Namespaces are the mechanism designed to help define a unique identifier for markup tags. Through the use of namespaces, a schema

can clearly identify each of these elements as having different meanings (semantics) or uses.

The data-type validation feature introduced by the XML schema could be used to improve the form validation implementation in the YBox. XML schema supports *simpleType* and *complexType* data-types. A *simpleType* definition allows the element declaration to contain only text (any data-type). The element being defined may not contain other elements or attributes. An example of a *simpleType* is:

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

The element described using this schema represents a phone number and must be an integer. This would be a useful feature for form validation in the YBox.

A *complexType* is an element that contains other elements (nested elements) between the opening and closing tags. An example of a *complexType* is:

```
<xsd:element name="fullName">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="firstName" type="xsd:string"/>
      <xsd:element name="lastName" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

This example shows an element that describes the full name of a person. The `fullName` element contains a `firstName` element and a `lastName` element (both strings). This feature of XML schema could be used for custom class form validation in the YBox framework. The `fullName` element could be mapped to a Java class in the web application design, and this class could implement the validation logic.

References

- [1] DCU School of Electronic Engineering, "Virtual Community Project", <https://vcp.eeng.dcu.ie/vcp/index.html>, 2000, (22 November 2002).
- [2] Nokia, "Nokia Developers Forum", <http://www.forum.nokia.com/main.html>, 2002, (22 November 2002).
- [3] Sun Microsystems Inc., "Java 2 Platform, Enterprise Edition", <http://java.sun.com/j2ee/>, 2000, (22 November 2002).
- [4] Microsoft, "Microsoft .NET Framework", <http://www.microsoft.com/net/>, 2000, (22 November 2002).
- [5] Hunter, Jason and Crawford, William, "Java Servlet Programming", O'Reilly, pp6-10, 1999.
- [6] Spainhour, Stephen and Quercia, Valerie, "WebMaster in a Nutshell", Chapter 9, 1996.
- [7] Sun Microsystems Inc., "Java Servlet Specification Version 2.3", <http://www.jcp.org/aboutJava/communityprocess/final/jsr053/>, 2001. (18 November 2002).
- [8] Wutka, Mark, "Using Java 2 Enterprise Edition", pp78-80, 2001.
- [9] Maddox, A, "Distributed Web Application Development: A Comparison of .Net and J2EE", www.manukau.ac.nz/EE/research/2002/am.pdf, 2002, (22 November 2002).
- [10] Sun Microsystems Inc., "Java 2 Platform, Standard Edition", <http://java.sun.com/j2se/>, 2002, (22 November 2002).

- [11] Microsoft, "Make Your Web Applications Support Pocket PC", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnppc2k/html/ppc_ppsupport.asp, 2002, (22 November 2002).
- [12] Apache Tomcat, <http://jakarta.apache.org/tomcat/index.html>, 2002, (22 November 2002).
- [13] Sun Microsystems Inc., "Java Servlet 2.2 Specification", <http://java.sun.com/products/servlet/2.2/>, 2001, (22 November 2002).
- [14] Harbourne-Thomas, Andrew, "Professional Java Servlets 2.3", pp365-368, 2002.
- [15] BEA Systems, "BEA Weblogic Server", <http://www.bea.com/products/weblogic/server/index.shtml>, 2002, (22 November 2002).
- [16] Apache Cocoon, "What is Cocoon?", <http://xml.apache.org/cocoon/index.html>, 2002, (22 November 2002).
- [17] Apache Struts, "The Apache Struts Web Application framework", <http://jakarta.apache.org/struts/index.html>, 2002, (22 November 2002).
- [18] Ostrovica, Ilirjan, "Form Processing API", <http://www3.sympatico.ca/iostro/fpapi2.0/>, 2001, (22 November 2002).
- [19] Baker, Mark, Shinichi Matsui, Ishikawa, Stark, Peter, Wugofski, Ted and Yamakami, Toshihiko, "XHTML Basic, W3C Recommendation", <http://www.w3.org/TR/xhtml-basic/>, 2000, (22 November 2002).
- [20] Apache XML Project, "FOP", <http://xml.apache.org/fop/index.html>, 2002, (22 November 2002).

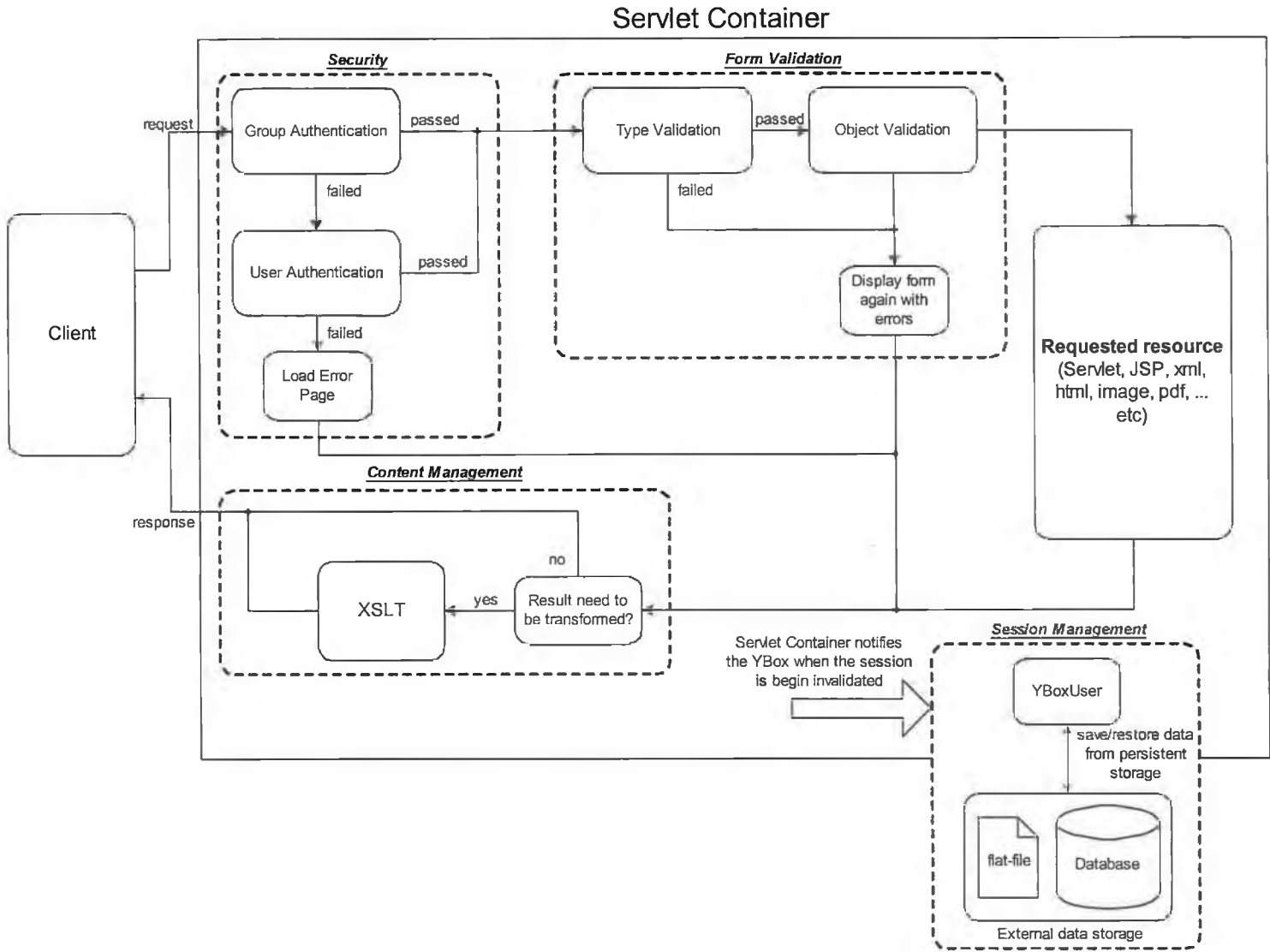
- [21] Apache XML Project, "Xalan-Java", <http://xml.apache.org/xalan-j/index.html>, 2002, (22 November 2002).
- [22] Sun Microsystems Inc., "Java Servlet 2.3 API Documentation", <http://java.sun.com/products/servlet/2.3/javadoc/>, 2001, (22 November 2002).
- [23] Floyd, Michael, "Building Web Sites with XML", pp300-305, 1999.
- [24] Apache XML Project, "Xerces2 Java Parser 2.2.1 Release", <http://xml.apache.org/xerces2-j/index.html>, 2002, (22 November 2002).
- [25] Bodoff, Stephanie, Green, Dale, Hasse, Kim, Jendrock, Eric, Pawlan and Stearns, Beth, "The J2EE Tutorial", pp216-218, 2002.
- [26] Apache Jakarta Project, "Apache JMeter", <http://jakarta.apache.org/jmeter/index.html>, 2002, (22 November 2002).
- [27] Sun Microsystems Inc., "Java Architecture for XML Binding", <http://java.sun.com/xml/jaxb/>, 2002, (22 November 2002).
- [28] Hunter, Jason and Crawford, William, "Java Servlet Programming", O'Reilly, pp3, 1999.
- [29] Zang, Ulla, "photo.net", <http://www.photo.net/community/>, 2002, (22 November 2002).
- [30] Microsoft, "Implement a Custom Common Language Runtime Host for Your Managed App"
<http://msdn.microsoft.com/msdnmag/issues/01/03/clr/default.aspx>, 2002, (22 November 2002).
- [31] W3C, "The Extensible Stylesheet Language (XSL)", <http://www.w3.org/Style/XSL/>, 2002, (22 November 2002).

- [32] W3C, “XSL Transformations (XSLT)”, <http://www.w3.org/TR/xslt>, 2002, (22 November 2002).
- [33] W3C, “Voice eXtensible Markup Language (VoiceXML™) version 1.0” <http://www.w3.org/TR/voicexml/>, 2002, (22 November 2002).
- [34] AT&T, “AT&T Natural Voices”, <http://www.naturalvoices.att.com/>, 2002, (22 November 2002).
- [35] Green, Dale, “The Reflection API”, <http://java.sun.com/docs/books/tutorial/reflect/>, 2002, (22 November 2002).
- [36] Kay, Michael, “The Saxon XSLT Processor”, <http://saxon.sourceforge.net/>, 2002, (22 November 2002).
- [37] DataPower Technologies, “XA35 XML Accelerator”, <http://www.datapower.com/products/index.html>, 2002, (22 November 2002).
- [38] Clarke, James, “XT”, <http://www.blz.com/xt/index.html>, 2002, (22 November 2002).
- [39] IBM, “XML Parser for Java”, <http://www.alphaworks.ibm.com/tech/xml4j>, 2002, (22 November 2002).
- [40] Oracle, “Oracles XML developers kit for Java”, http://otn.oracle.com/software/tech/xml/xdk_java/content.html, 2002, (22 November 2002).
- [41] Sun Microsystems, “Java API for XML Processing”, <http://java.sun.com/xml/jaxp/>, 2002, (22 November 2002).
- [42] W3C, “XML Schema: Formal Description”, <http://www.w3.org/TR/xmlschema-formal/>, 2002, (22 November 2002).

- [43] W3C, “HTTP – Hypertext Transfer Protocol”, <http://www.w3.org/Protocols/>, 2002, (22 November 2002).

- [44] Apache, “Other useful presentation tags”, http://jakarta.apache.org/struts/doc-1.0.2/userGuide/building_view.html, 2002, (22 November 2002).

Appendix A – Complete Diagram of the YBox



Appendix B – Source code for sample Application

index.xml

```
<?xml version="1.0"?>
<page>
<title>Test Page</title>
<paragraph>This is a test page.</paragraph>
<paragraph>If you can view this, the YBox is alive.</paragraph>
</page>
```

form.xml

```
<?xml version="1.0"?>
<page>
<title>Sample Form</title>
<paragraph>This form is used to test the capabilities of the
YBox.</paragraph>

<form servlet="SampleForm" method="submit(name, verifyAge)">
  <class name="verifyAge"
        constructor="ie.dcu.liamf.test.VerifyAge(yob,
age)">
    </class>

    <sel name="verifyAge.age">
      <opt>21</opt>
      <opt>22</opt>
      <opt>23</opt>
      <opt>24</opt>
      <opt>25</opt>
      <opt>26</opt>
      <opt>27</opt>
      <opt>28</opt>
      <opt>29</opt>
      <opt>30</opt>
      <opt>31</opt>
      <opt>32</opt>
      <opt>33</opt>
      <opt>34</opt>
      <opt>35</opt>
    </sel>

    <sel name="verifyAge.yob">
      <opt>1956</opt>
      <opt>1957</opt>
      <opt>1958</opt>
      <opt>1959</opt>
      <opt>1960</opt>
      <opt>1961</opt>
      <opt>1962</opt>
      <opt>1963</opt>
      <opt>1964</opt>
      <opt>1965</opt>
      <opt>1966</opt>
```

```

<opt>1967</opt>
<opt>1968</opt>
<opt>1969</opt>
<opt>1970</opt>
<opt>1971</opt>
<opt>1972</opt>
<opt>1973</opt>
<opt>1974</opt>
<opt>1975</opt>
<opt>1976</opt>
<opt>1977</opt>
<opt>1978</opt>
<opt>1979</opt>
<opt>1980</opt>
<opt>1981</opt>
</sel>

<h3>Question 1.</h3>
<input-text
                                ip_type="text"
                                type="String"
                                name="name"
                                descr="Your Name "
                                size="20" required="true"
                                value=""
                                errorMsg="Please enter your Name!">

</input-text>

<h3>Question 2.</h3>
<input-text
                                ip_type="select"
                                type="int"
                                name="verifyAge.age"
                                descr="Your Age "
                                size="7" required="true"
                                value=""
                                errorMsg="Please enter your Age!">

</input-text>

<h3>Question 3.</h3>
<input-text      ip_type="select"
                                type="int"
                                name="verifyAge.yob"
                                descr="Year of Birth "
                                size="7"
                                required="true"
                                value=""
                                errorMsg="Please enter your YOB!">

</input-text>

<paragraph/>
<input-button value="Submit Details"></input-button>
</form>

</page>

```

login.xml

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

```

<page>
<title>Login Page</title>
<paragraph>Enter a user name and user group.</paragraph>
<paragraph>Test - A group of "all" will give access to
everything.</paragraph>

<form servlet="Login" method="login(yboxUser) ">
  <class name="yboxUser"

    constructor="ie.dcu.liamf.test.TestYBoxUser(userName,
userGroup) ">
    </class>
    <input-text      ip_type="text"
                    type="String"
                    name="yboxUser.userName"
                    descr="User Name"
                    size="7" required="true"
                    value=""
                    errorMsg="Please enter user name">

    </input-text>
    <input-text      ip_type="text"
                    type="String"
                    name="yboxUser.userGroup"
                    descr="Enter ther group you belong to:"
                    size="7"
                    required="true"
                    value=""
                    errorMsg="Please enter group!">

    </input-text>
    <input-button value="Login"></input-button>
</form>

</page>

```

logout.xml

```

<?xml version="1.0"?>
<page>
<title>Logout Page</title>
<paragraph>Click on the button below to Logout</paragraph>

<form servlet="Logout" method="logout() ">
  <input-button value="Logout"></input-button>
</form>

</page>

```

sample.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<page>
  <title>
    Sample Document
  </title>
  <paragraph>
    This sample document is only used to demonstrate the
power of the YBox. This sample page

```

can be viewed in a standard web browser such as Internet Explorer or Netscape Navigator.

It can also be viewed on mobile phones or PDAs.

</paragraph>

<paragraph>

Images can also be viewed on some clients, depending on the XSL configuration. If the image

can not be viewed on the client in question, then the

<code>alt</code> attribute of the

<code>image</code> tag is displayed.

</paragraph>

<image src="http://140.204.145.229:8080/demo/dculogo.gif"

align="center" alt="DCU Logo"></image>

<figure>Figure 1. The DCU Official Logo</figure>

<paragraph>

It is also possible to display in-line code:

<code>java.lang.Object</code> is a java class.

</paragraph>

</page>

session.xml

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

```
<page>
```

```
<title>Test Session Storage</title>
```

```
<paragraph>This page will test the session storage mechanism of the YBox</paragraph>
```

```
<paragraph>The variable will be stored in the session.</paragraph>
```

```
<paragraph>If the user logs out, the session will be stored to hard-disk.</paragraph>
```

```
<paragraph>When the user logs in again, the session will be restored and the user will have lost no information.</paragraph>
```

```
<form servlet="SessionTest" method="storeSession(id, val)">
```

```
  <input-text      ip_type="text"
                    type="String"
                    name="id"
                    descr="Variable ID "
                    size="7" required="true"
                    value=""
                    errorMsg="Please enter a variable ID">
```

```
  </input-text>
```

```
  <input-text      ip_type="text"
                    type="String"
                    name="val"
                    descr="Value "
                    size="7"
                    required="true"
                    value=""
                    errorMsg="Please enter value">
```

```
  </input-text>
```

```
  <input-button value="Store session variable"></input-button>
```

```
</form>
```

```
<paragraph>Enter the ID of the session attribute you wish to restore.</paragraph>
```

```
<form servlet="SessionTest" method="getSession(id_check)">
```

```
  <input-text      ip_type="text"
```

```

        type="String"
        name="id_check"
        descr="Variable ID "
        size="7" required="true"
        value=""
        errorMsg="Please enter a variable ID">
    </input-text>
    <input-button value="Get session variable"></input-button>
</form>

</page>

```

access_denied.xml

```

<?xml version="1.0"?>
<page>
<title>Error - incorrect user.</title>
<paragraph>Error - you do not have the required permissions to view
this resource!</paragraph>
<paragraph><link href="login.xml" descr="Go to login
page."></link></paragraph>

</page>

```

Login.java

```

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.util.Vector;

public class Login extends HttpServlet
{
    public void login(HttpServletRequest req, HttpServletResponse
res, ie.dcu.liamf.test.TestYBoxUser user)
    throws ServletException, IOException
    {
        PrintWriter out = res.getWriter();
        user.setYBoxUser(req,user);
        user.restoreSesion(req);

        out.println("<?xml version=\"1.0\"?>");
        out.println("<?xml-stylesheet href=\"hello-page-html.xsl\"
type=\"text/xsl\"?>");
        out.println("<page>");
        out.println("<title>Login Successful</title>");
        out.println("<paragraph>Well done!</paragraph>");
        Vector v = (Vector)user.getUserGroups();
        out.println("<paragraph>user name = " + user.getUserName() +
"</paragraph>");
        for(int i=0; i<v.size(); i++)
        {
            out.println("<paragraph>user group = " + (String)v.get(i)
+ "</paragraph>");
        }
        out.println("</page>");
    }
}

```


}

Logout.java

```

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

public class Logout extends HttpServlet
{
    public void logout(HttpServletRequest req, HttpServletResponse
res)
    throws ServletException, IOException
    {
        PrintWriter out = res.getWriter();
        req.getSession().invalidate();

        out.println("<?xml version=\"1.0\"?>");
        out.println("<?xml-stylesheet href=\"hello-page-html.xsl\"
type=\"text/xsl\"?>");
        out.println("<page>");
        out.println("<title>Logout Successful</title>");
        out.println("<paragraph>You have logged out of the demo web
application.</paragraph>");

        out.println("</page>");
    }
}

```

SampleForm.java

```

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.util.Vector;

public class SampleForm extends HttpServlet
{
    public void submit(HttpServletRequest req, HttpServletResponse
res, String name, double height, ie.dcu.liamf.test.VerifyAge vAge)
    throws ServletException, IOException
    {
        PrintWriter out = res.getWriter();

        out.println("<?xml version=\"1.0\"?>");
        out.println("<?xml-stylesheet href=\"hello-page-html.xsl\"
type=\"text/xsl\"?>");
        out.println("<page>");
        out.println("<title>Form Validation Successful</title>");
        out.println("<paragraph>Name = " + name + "</paragraph>");
        out.println("<paragraph>Height = " + height +
"</paragraph>");
        out.println("<paragraph>Age = " + vAge.getAge() +
"</paragraph>");
        out.println("<paragraph>Year of birth = " + vAge.getYob() +
"</paragraph>");
        out.println("</page>");
    }
}

```

```
}
}
```

SessionTest.java

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

public class SessionTest extends HttpServlet
{
    public void storeSession(HttpServletRequest req,
        HttpServletResponse res, String id, String value)
        throws ServletException, IOException
    {
        PrintWriter out = res.getWriter();

        req.getSession().setAttribute(id, value);

        out.println("<?xml version=\"1.0\"?>");
        out.println("<?xml-stylesheet href=\"hello-page-html.xsl\"
type=\"text/xsl\"?>");
        out.println("<page>");
        out.println("<title>Session attribute added</title>");
        out.println("<paragraph>The session attribute with id='" + id
+ "' and value='" + value + "' has been added to the
session.</paragraph>");
        out.println("</page>");
    }

    public void getSession(HttpServletRequest req,
        HttpServletResponse res, String id)
        throws ServletException, IOException
    {
        PrintWriter out = res.getWriter();

        out.println("<?xml version=\"1.0\"?>");
        out.println("<?xml-stylesheet href=\"hello-page-html.xsl\"
type=\"text/xsl\"?>");
        out.println("<page>");
        out.println("<title>Session attribute restored</title>");
        try
        {
            String value = (String)req.getSession().getAttribute(id);
            if(value == null)
                out.println("<paragraph>The session attribute with
id='" + id + "' does not exist.</paragraph>");
            else
                out.println("<paragraph>The session attribute with
id='" + id + "' has a value='" + value + "'.</paragraph>");
        }
        catch(Exception e)
        {
            out.println("<paragraph>The session attribute with id='"
+ id + "' does not exist.</paragraph>");
        }

        out.println("</page>");
    }
}
```

```
}
}
```

TestYBoxUser.java

```
package ie.dcu.liamf.test;
import ie.dcu.liamf.ybox.user.YBoxUser;
import java.util.*;
import java.io.*;
import javax.servlet.http.*;

public class TestYBoxUser extends YBoxUser
{
    String userName;
    Vector userGroups;

    public TestYBoxUser(String userName, String userGroup)
    {
        this.userGroups = new Vector();
        System.out.println("+++++++");
        System.out.println("DEBUG: inside constructor!");
        this.userGroups.add(userGroup);

        this.userName = userName;
        System.out.println("DEBUG: leaving constructor!");
        System.out.println("+++++++");
    }
    public Vector getUserGroups()
    {
        return this.userGroups;
    }
    public String getUserName()
    {
        return this.userName;
    }
    public void saveSesion(Hashtable sessionVariables)
    {
        System.out.println("DEBUG: print out all the session
variables");
        Enumeration e = sessionVariables.keys();
        while(e.hasMoreElements())
        {
            String s = (String)e.nextElement();
            System.out.println(s);
        }
        this.write(sessionVariables);
    }
    private void write(Hashtable h)
    {
        YBoxUser user = null;
        try
        {
            user = (YBoxUser)h.get("yBoxUser");
        }
        catch(Exception except)
        {
```

```

        System.out.println("INFO: user not logged in - will not
save session attributes.");
        except.printStackTrace();
    }
    try
    {
        FileOutputStream fo = new
FileOutputStream(user.getUserName() + ".dat");
        ObjectOutputStream o = new ObjectOutputStream(fo);
        o.writeObject(h);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
public void restoreSesion(HttpServletRequest req)
{
    Hashtable h = new Hashtable();
    YBoxUser user =
(YBoxUser)req.getSession().getAttribute("yBoxUser");
    try
    {
        FileInputStream fo = new
FileInputStream(user.getUserName() + ".dat");
        ObjectInputStream o = new ObjectInputStream(fo);
        h = (Hashtable)o.readObject();

        Enumeration e = h.keys();
        while(e.hasMoreElements())
        {
            String s = (String)e.nextElement();
            req.getSession().setAttribute(s, h.get(s));
        }

    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}
}

```

VerifyAge.java

```

package ie.dcu.liamf.test;
import java.text.ParseException;

import java.util.*;

public class VerifyAge
{
    private int yob;
    private int age;

    public VerifyAge(int yob, int age) throws ParseException
    {

```

```
        this.yob = yob;
        this.age = age;

        String[] ids = TimeZone.getAvailableIDs(-8 * 60 * 60 * 1000);
        SimpleTimeZone pdt = new SimpleTimeZone(60 * 60 * 1000,
ids[0]);
        // set up rules for daylight savings time
        pdt.setStartRule(Calendar.APRIL, 1, Calendar.SUNDAY, 2 * 60 *
60 * 1000);
        pdt.setEndRule(Calendar.OCTOBER, -1, Calendar.SUNDAY, 2 * 60
* 60 * 1000);
        // create a GregorianCalendar
        // and the current date and time
        Calendar calendar = new GregorianCalendar(pdt);
        Date trialTime = new Date();
        calendar.setTime(trialTime);

        int diff = calendar.get(Calendar.YEAR) - this.yob;
        System.out.println("YEAR=" + calendar.get(Calendar.YEAR));
        System.out.println("diff=" + diff);
        System.out.println("age=" + age);
        if((age==diff) || (age==(diff-1)))
        {
            // ok to construct
        }
        else
            throw(new ParseException("Invalid age", 1));
    }
    public int getYob()
    {
        return this.yob;
    }
    public int getAge()
    {
        return this.age;
    }
}
```