NTNU

Innovation and Creativity

Framework Support for Web Application Security

Leif Ødegård

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Torbjørn Skramstad, IDI
Co-supervisor: Lillian Røstad, IDI
                   Erlend Oftedal, Bekk Consulting As

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

There is currently a number of rather well-known security pitfalls when developing web applications. Current technologies offer little or no support for avoiding these pitfalls, which results in need for acute attention to these matters from programmers. The project will consist of the following:
-Create an overview of the most common web security pitfalls and discuss the architectual patterns from which these arise.
-Discuss how and to what degree relevant web application frameworks address these pitfalls.
-Give an overview of and evaluate existing initiatives to add framework support for such pitfalls.
-Suggest how support for avoiding these security pitfalls could be added to relevant frameworks.
The project should at least address Java EE (Enterprise Edition) based frameworks, but possibly also Microsoft .NET, PHP, or Ruby on Rails.
Oppgaven utføres i samarbeid med Bekk Consulting AS.

Assignment given: 20. January 2006
Supervisor: Torbjørn Skramstad, IDI

ABSTRACT

There are several good reasons to use a framework when you are developing a new web application. We often here that:

- frameworks use known patterns that result in an easily extendable architecture

- frameworks result in loose couplings between different modules in the application

- frameworks allow developer to concentrate on business logic instead of reinventing wheels that is already reinvented several times

- frameworks are often thoroughly tested and contains less bugs than custom solutions

But security is rarely mentioned in this setting. Our main motivation in this thesis is therefore to discuss what three popular web application frameworks do to improve the overall security level.

In this thesis we have chosen to research Spring, Struts and JavaServer Faces. We use them to develop small applications and test whether they are vulnerable to different types of attacks or not. We focus on attacks involving metacharacters such that SQL-injection and cross-site scripting, but also security pitfalls connected to access control and error handling.

We have found out that all three frameworks do implement some metacharacter handling. Since Spring tries to fill the role of a full-stack application framework, it provides some SQL metacharacter handling to avoid SQL-injections, but we have identified some implementation weaknesses that may lead to vulnerabilities. Cross-site scripting problems are handled in both Spring, Struts, and JSF by HTML-encoding as long as custom `RenderKits` are not introduced in JSF.

When it comes to access control, the framework support is somewhat limited. They do support a role-based access control model, but this is not sufficient in applications where domain object access is connected to users rather than roles. To improve the access control in Struts applications, we provide an overall access control design that is based on Aspect-Oriented Programming and integrates with standard Struts config files. Hopefully, this design is generic enough to suit several application's needs, but also useable to developers such that it results in a more secure access control containing less bugs than custom solutions.

PREFACE

This report is the result of thesis work performed during the spring semester 2006 by one student at Norwegian University of Science and Technology, Department of Computer and Information Science. The project was initiated by Erlend Oftedal who works at Bekk Consulting AS.

CONTENTS

LIST OF FIGURES

x

LIST OF LISTINGS

List of Tables

CHAPTER 1

INTRODUCTION

This chapter serves as a general introduction to this thesis. We provide sections discussing our motivation, the research goals and scope and an outline of the chapters in this report.

1.1 MOTIVATION

In traditional software applications, vulnerabilities is a well-known term. We often hear that attacks like buffer overflows, format string attacks, command injections, etc. may result in execution of arbitrary code. However, there are some additional groups of vulnerabilities connected to web applications. SQL-injections and cross-site scripting are terms describing common security vulnerabilities in these applications.

To avoid introducing vulnerabilities when developing new web applications, it seems like a good idea to use some kind of framework that is providing suitable abstractions. This approach has several advantages over developing every web application from scratch:

- Using a framework often leads to a more loose coupling between the components in the system.

- A framework often recognizes typical programming tasks and provides generic solutions to these tasks.

- Frameworks are often tested by more developers than custom solutions. This lowers the number of bugs per line of source code.

- A framework may implement generic security functionality such that developer may focus on the business logic. This approach supports the security principle known as "Secure by default".

Based on these statements, our main motivation for this thesis is to get an understanding of the security functionality implemented by web application frameworks. Some of the questions we want to find answers to are:

- Are there any framework security measures?

- Are they enabled by default?

- How can we use frameworks to improve the security level?

- Can we develop custom framework extensions to improve web application security?

1.2 GOALS

In addition to the scope description in section 1.3, our task specifies at set of goals. We have chosen to rewrite these goals with some modifications, to restrict the number of pitfalls, frameworks, etc. we are concentrating on. The goals are:

- Discuss security pitfalls found in web applications and in which architectural patterns they occur.

- Discuss architectural patterns used in three popular Java-based frameworks (Struts, Spring and JavaServer Faces (JSF))

- Discuss if and to what degree these three frameworks implement functionality that may help the programmer to avoid typical security pitfalls.

- Suggest how the security may be improved in at least one of the discussed frameworks.

1.3 SCOPE

Because of this project's limited amount of resources, we think it is important to point out the key topics clearly, but also mention those we will pass over in silence.

First of all, we have chosen to focus on server side Java web application frameworks. This means that client-side web applications based on Java Applets [36], Microsoft ActiveX Controls [9] and JavaScript [72] are defined to be out of our scope. This is an important restriction, but it is not enough. [38] mentions tens of existing frameworks that still matches this project's criteria. Some of them are:

- Struts

- Turbine

- Tapestry

- WebWork

- Cocoon

- Spring

- Maverick

- Echo

- JavaServer Faces

- Stripes

Based on Bekk Consulting's wishes and general popularity among Java web application developers (see e.g. [64]), we have decided to concentrate on Struts, Spring and JavaServer Faces.

The last point we will comment on in this section is communication link security (e.g. cryptographic primitives). We do not consider flaws in the crypto software to be relevant in this thesis because we focus on flaws in the application layer and they may be exploited even if the communication link is encrypted at the IP- and transport layer and the firewalls are securely installed.

1.4 SOFTWARE USED THROUGHOUT THIS THESIS

To get an understanding of how our three chosen web application frameworks process web browser requests, we have used them to develop small examples throughout the thesis. To get these examples up and running, we needed different software:

- Acegi-JSF components 1.1.2 [62]

- Acegi Security System 1.0.0-RC2 [42]

- Apache Tomcat 5.5.16 [18]

- JSF-Security 1.0 [63]

- MyFaces 1.1.2 [16]

- MySQL 5.5.21 [4]

- Spring Framework 1.2.5 [74]

- Struts Action Framework 1.2.9 [17]

Even if there are several JSF implementations available (e.g. Sun JSF Reference Implementation [53], Oracle ADF Faces [11]), we chose to use the Apache MyFaces implementation because of its business friendly license and some quite useful extensions that is not part of the official specification.

1.5 REPORT OUTLINE

- **Chapter 1 – Introduction**
  This chapter provides a general introduction our project, containing motivation, goals, scope and a report outline.

- **Chapter 2 – Web security pitfalls**
  This chapter discusses common security pitfalls found in web applications.

- **Chapter 3 – Web application frameworks**
  In this chapter we will discuss Spring, Struts and JavaServer Faces in some detail. We will focus on their architecture and how they use different design patterns.

- **Chapter 4 – Spring security**
  This chapter discusses to what degree web applications developed using Spring are vulnerable to the security pitfalls discussed in chapter 2.

- **Chapter 5 – Struts security**
  This chapter discusses to what degree web applications developed using Struts are vulnerable to the security pitfalls discussed in chapter 2.

- **Chapter 6 – JavaServer Faces security**
  This chapter discusses to what degree web applications developed using JavaServer Faces (JSF) are vulnerable to the security pitfalls discussed in chapter 2.

- **Chapter 7 – Struts ACL extension**
  In this chapter we focus on the limited support for Access Control List (ACL)s in Struts. We use an Internet banking service to illustrate our points, and provide an overall design of a Struts ACL extension.

- **Chapter 8 – Conclusion**
  This chapter provides a summary of our findings and lessons learned throughout the project.

- **Chapter 9 – Future work**
  The last chapter outlines future work related to this project.

- **Chapter 9 – Future work**

CHAPTER **2**

WEB SECURITY PITFALLS

In this chapter we will discuss several groups of security pitfalls found in today's web applications. It includes problems like the well-known SQL-injection problem and cross-site scripting, but other problems are also mentioned. We will try to focus on the real causes of these problems, and this often means bad metacharacter handling. This chapter is inspired by the work of Sverre H. Huseby [29, 31] and the pitfalls discussed in [23, 65].

2.1 INPUT

Input is an essential part in all the web applications we know today. If web users were not allowed to enter input, the applications seem quite useless. In this section we will discuss the role input plays in web applications, the differences between client side- and server side input validation and why input validation alone is not enough to enforce a secure operation.

2.1.1 What is input?

In web applications, we usually think of input as the text that we type into Hyper-Text Markup Language (HTML) form elements like text fields, text areas, password fields, etc. This is true, but we should remember that there are at least three other sources to input as well:

1. Data from form elements like radio buttons, drop-down and check boxes and hidden fields.

2. HyperText Transfer Protocol (HTTP) headers and cookies

3. Backend databases or other software systems

The important point is that we know and understand all the sources of input such that no dangerous data is allowed to leave our web application.

2.1.2 What is input validation?

Validating input is about ensuring that the input given to our application belongs to the correct domain (e.g. an integer) and within the range accepted by the application code (e.g. $>= 0$ and < 99). Different domains requires different validation. Some the domains are:

- E-mail addresses

- Account numbers (e.g. in Internet banking services)

- Country codes

- Postal codes

- URLs

- Phone numbers

- Credit card numbers

- Usernames

- Real names

- Year of birth

We do not think of input validation as a way to enforce security by disallowing input that may threat our application. Instead we think that validation is important to ensure that our web application behaves as expected. If we allow the user to input whatever he/she would like to without any validation at all, we risk that the user may be able trick our application into making wrong decisions. This may compromise the security of the application completely.

2.1.3 Client side validation

The most common way to validate input at the client side often means using some kind of web browser scripting language (e.g. JavaScript). The input may be validated as the user enters it or a JavaScript function may be called when the user presses a submit button. No data is passed from the web browser to the web server before the input is accepted by the script.

Unfortunately, client side validation does not contribute to web application security at all. An attacker may easily bypass this validation by saving the file to his/her hard drive and then using the HTML comment tags ($<$!– –$>$) or, by using a proxy application (e.g. PenProxy, [30]) to change the input given to the web server after the client side validation has taken place.

Client side input validation may be a good technique to use when creating easy-to-use interfaces because a script may guide the user through the form fill-in process, but all the validation related to security should be placed on the server-side.

2.1.4 Server side validation

Server side input validation is required if want keep our web applications out of an attacker's hands. Regular expressions are often used to limit the set of accepted input. There are at least two ways to use these expressions:

Whitelisting
> When using whitelisting, we develop regular expressions that describes what input our application should accept. If the strings, integers, etc. given by the user matches our expression(s), we accept it and some business logic is executed.

Blacklisting
> In blacklisting we try to identify the set of dangerous strings, integers, etc. that may trick our application and develop regular expressions suiting these situations. If the user input match these expressions, we reject them. Otherwise, they are accepted.

We usually prefer using whitelisting in input validation routines because there are usually, almost impossible to identify all expressions that may defeat our web application. A creative attacker may use different character encodings and other techniques to bypass our input validation.

Server side validation do contribute to application security because there is no way for an attacker to bypass this validation without gaining access to the source code on the server itself. If he/she already has gained such access, he/she would probably spend his/her time on executing more harmful attacks than bypassing the input validation.

2.1.5 Why input validation is not enough

It is not trivial to understand why input validation all by itself is not enough to prevent web applications from being vulnerable to attacks. The root of the problem is that all software applications usually define a set of characters that have special meaning to them. For instance does the character ' in Structured Query Language (SQL) mean that we are switching to or from a context that is accepting a plain text string. As we do not want to disallow names like "Chloe O'Brian" in our web application, we need to handle these special characters at some level. We will discuss the security implications of metacharacters in the following sections.

2.2 META-CHARACTER PROBLEMS

SQL-injection and cross-site scripting are both well-known terms in web application security. These are discussed in details in section 2.2.2 and 2.2.3, respectively. A lot of security experts claim that these flaws occur when the input from the user is not validated properly, but is it really so? And what is a metacharacter?

2.2.1 Metacharacters

A metacharacter is defined by [29] as

> . . . a character that is not treated as plain text by the receiver. The metacharacters represents control information.

One of the classical metacharacter vulnerabilities occur when a web application calls command line system tools. Classes and functions in several major web programming languages (e.g. Java, PHP, Perl) provide this functionality. These functions themselves are not unsafe, but they may be vulnerable to attacks if unfiltered user input is just concatenated with the rest of the command.

To illustrate the dangers of calling GNU/Linux shell utilities, let us assume that we are developing a web application which uses `lpr` to do some printing. The command executed by the application is:

```
lpr -PMainPrinter $file
```

where `MainPrinter` represents a printer name and $file is user-specified variable indicating the file name of the file that should be printed. If we assume the there is no validation or filtering implemented, this code represents several risks:

- If a user may specify $file without restrictions, there is a trivial task to get hands on copy of `/etc/passwd` which contains information about registered users and possibly an encrypted version of their passwords.

7

- If `lpr` is running with root privileges and an attacker is able to inject something like:

  `testFile.pdf; rm -rf /;`

  into $file, every piece of stored data on the mounted filesystems is erased.

This last example fits exactly with our metacharacter definition. Semicolon is treated as an end-of-command character in a GNU/Linux shell and the attacker is able to add a new and more damaging command. The attacker forces a context switch within the shell environment by injecting a metacharacter. But what happens if a user is capable of injecting metacharacters and thus switching contexts in a SQL-statement?

2.2.2 SQL-injection

SQL-injection means to inject SQL-statements into web applications, but possible also other software. In the rest of this section we will discuss how this is possible through some examples and then, why SQL-injection problems may be classified as metacharacter problems.

2.2.2.1 The problem

To make a piece of software worth using, it needs to do some data processing. As discussed in section 2.1, input to an application may come from quite different sources. Usually, we have an architecture consisting of a user interface, some business logic and a data store (e.g. DBMS). This is shown in figure 2.1.
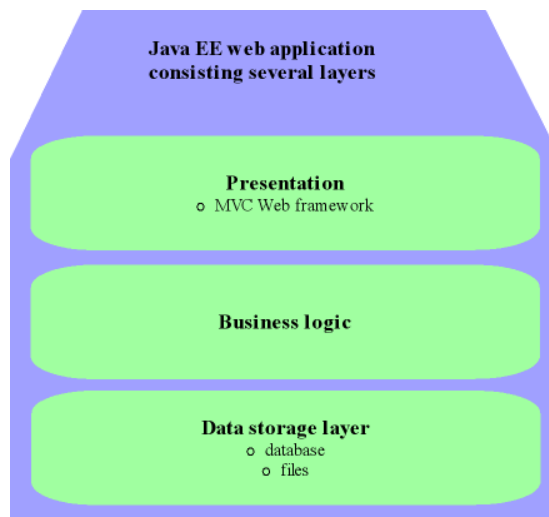
Figure 2.1: Java EE overall architecture

This architecture is found a lot a web applications as well. The user interface often consists of several HTML pages viewed in a web browser and input is accepted through HTML forms [71]. The HTML code for a simple login form is given in appendix A.

When a form is submitted, all the input given by the user (text fields, text areas, radio buttons, drop-down lists etc.), is represented as strings in the request for the

page specified in action attribute of the form tag[1]. A sample request may look like the following:

```
http://localhost:8080/form.jsp?
      username=leifode
      &password=seCretPassWord
      &submit=Submit
```

Then, when the requested page is about to render, the programmer may use the `getParameter()` method of the request object to access the submitted data. This is shown in listing 2.1.

Listing 2.1: request.getParameter()

```
1  <%
2     String username = request.getParameter("username");
3     String password = request.getParameter("password");
4
5     if(username != null && password != null)
6     {
7       String sqlQuery = "SELECT * users WHERE username='"+
             username+"' AND password='"+password+"'";
8       out.println("Executing query...:<br/>");
9       out.println(sqlQuery);
10    }
11 %>
```

This listing also displays an SQL-query that is about to executed. Let us say that a user without bad intentions enter `leifode` and `seCreTpaSsword` as username and password, respectively. Then the query given in listing 2.2, is executed.

Listing 2.2: Login query

```
1  SELECT * FROM users WHERE username='leifode' AND password='
      seCreTpaSsword'
```

The user is considered authenticated if the returned ResultSet contains a single row.
Now consider an attacker entering

```
leifode' --
```

as username and leaving the password text field empty. Since the user input is only concatenated with the rest of the query, we get the following:

Listing 2.3: Bypassing the password check

```
1  SELECT * FROM users WHERE username='leifode' -- ' AND
      password='';
```

This query returns a `ResultSet` containing a single row because the hyphens have a special meaning in MySQL. Like the '#' character they indicate a comment to the end of the line [1]. The password condition is simply commented out. Sadly, this is only the beginning...

If we assume that we have a JSP similar to the one given in appendix B, more destructive attacks are possible. Assume that an attacker enters a username like:

---

[1] This is not necessary correct. The form fields are only represented as a part of the URL as long as the method attribute of the form tag is set to GET. If we set the method attribute to POST, the data entered by the user is not append to the Uniform Resource Locator (URL).

According to several comments (e.g. [41]) it is not correct to use GET in this particular example, but we do it to clarify our examples.

```
leifode','seCreTpaSsword'); DELETE FROM users; #
```

Then, the query given in listing 2.4, is executed.

<div align="center">Listing 2.4: Injecting a DELETE query</div>

```
1  INSERT INTO users (username, password) VALUES ('leifode','
      seCreTpaSsword'); DELETE FROM users; #','')
```

This query may delete the contents of any table in the database and we may even reformulate it using the `DROP` command of MySQL [2] to remove the table from database[2].

2.2.2.2   A metacharacter problem?

Definitely. This is quite clear if we look back to the metacharacter definition given in section 2.2 and at listing 2.3 and 2.4.

In listing 2.3, two hyphens are used to bypass the password condition of the query. This is possible because these hyphens represent a piece of control information in MySQL (e.g. the rest of the line is treated as a comment).

Listing 2.4 uses a couple of other MySQL metacharacters. Some apostrophes are used to control which parts of the of input that should be treated as strings and which parts that should be considered as control information. The semicolons indicate the end of each query and a '#' at the end of the input ensures that the rest of the line is commented out.

These two examples show only a few metacharacters found in MySQL and the problems related to them. They also illustrate how important it is to handle the metacharacters properly to ensure a secure operation.

In the next section we will look at another pitfall found in various web applications, namely cross-site scripting. We will look into the root of the problem and then connect cross-site scripting back to the discussion in section 2.2.1.

2.2.3   Cross-site scripting

Cross-site scripting is another pitfall related to web applications and it is often mentioned together with SQL-injections. The reasons for this may not be obvious, but we will try to make it clear throughout this section. We will use a couple of examples to illustrate our points.

2.2.3.1   The problem

As stated by [31], cross-site scripting is

> ... about tricking a web server into presenting malicious HTML code, typically script code, to a user. The intention is often to steal session information, and thus to be able to contact the site on behalf of the victim.

You may recognize that this is almost the same problem as SQL-injection except that the cross-site scripting problems occur when data leave the application code and enter a web browser whereas SQL-injection issues occur when data enter a database system. We are talking about two variants of the same problem and that is most of the explanation why these pitfalls are mentioned in the same paragraphs.

---

[2]Actually, MySQL AB has incorporated some piece of security both in their Java Database Connectivity (JDBC) driver and database itself. More details on these topics is found in appendix F.

When looking for web sites vulnerable to cross-site scripting attacks, the first thing to do is to locate a HTML form which accepts user input and a page that shows the inputted data. In the following we continue to use the examples found in appendix A and B.

If we assume that a malicious user inputs

```
leifode<script>alert("CrAckinG ThAh PaGe!")</script>
```

as username and

```
seCreTPassWoRD
```

as password on the page called `registeruser.jsp`. The he/she access the page where the details are printed (e.g. loginform.jsp) to check whether there is a message box popping up showing the string "CrAckinG ThAh PaGe!".

If the message box is displayed, the web site is vulnerable to cross-site scripting attacks because the web browser executed our script. Based on this knowledge, the attacker may continue the attack and inject more complicated scripts that e.g. steal session information.

2.2.3.2 Stealing a session

The first half of this section discusses what a session is and why an attacker want to get access to a user's session identifier. The last part uses a complete example to show the steps in a session stealing attack. The example is a somewhat extended version of an example found in [31].

To understand why we want to steal an HyperText Transfer Protocol (HTTP) session, we need some basic understanding of HTTP [33, 34], and what a session is. HTTP is a protocol used by web browsers (e.g. Firefox [10]) to access web pages, images and other files found on a web server (e.g. Apache HTTP Server[15]). This protocol is stateless which means that the server does not save any information about which web pages that was accessed in the past. The client passes only a request to the server and the server returns the requested resource if it exists and is accessible.

To maintain state information between requests, HTTP sessions were introduced. A HTTP session is a collection of variables saved on the server side, identified by a non-guessable session identifier. Each time a client access the web server, it also passes the session identifier. The server validates the identifier and uses the saved session information if the identifier is valid (e.g. not expired or invalidated). If an attacker is able to guess or get the session identifier, he or she may impersonate the user and access the web page on behalf of the user. If the vulnerable application is a web shop where credit card information is stored, we all understand the consequences.

Let us use a concrete example to illustrate our points. Suppose that we have the cross-site scripting vulnerable page given in appendix C. Each user entering this page is associated with a session identifier and some other information. This page is just a "dummy" page accepting comments. In real life this could have been a web shop allowing the users to comment on the products they have bought. Everything works well if comments like "A product worth buying" and "Do not buy this product!" are inputted. But what happens if a cracker enters something like:

```
<script>
if(document.cookie.indexOf("stolen") < 0)
{
   document.cookie = "stolen=true";
   document.location.replace(
```

```
    "http://localhost:8080/cookie/stealsession.jsp?what="
    +document.cookie
    +"&whatnext=http://localhost:8080/cookie/cookiepage.jsp")
}
</script>
```

To understand the intention of the cracker, we need to understand some simple lines of JavaScript. The script operations are:

1. Check whether the cookie connected to the current user is already stolen. If it is, do nothing. Otherwise continue to the next JavaScript operation.

2. Add a small piece of information (e.g. stolen=true) to the cookie to ensure that we are not entering an infinite redirection loop.

3. Then replace the current page, with a page on a server controlled by the attacker and post both the user cookie and the address of the page to which we will return after the session identifier hijacking.

So, each time a new user accesses this page, he/she is redirected to a file (e.g. stealsession.jsp found in appendix D) on a server controlled by the attacker. The operation of this JavaServer Page (JSP) script is quite simple, it only saves the posted session information before redirecting the user back the original page. By installing this stolen information in his/her browser, the attacker may be able to impersonate a user and the misuse available personal data (e.g. credit card information).

Stealing session information is only one possible attack that uses the cross-site scripting techniques. [31] explains some other attacks, but like all other software attacks they are only limited by the creativity of the attacker.

2.2.3.3   A metacharacter problem?

Absolutely. However, the characteristics of cross-site scripting problems are somewhat different than the characteristics of SQL-injections. While SQL-injections occur when metacharacters enter a database subsystem, the cross-site scripting problems begin when HTML metacharacters enter a browser. There is a quite elegant solution to cross-site scripting problems in general, that is discussed in section 2.4.

2.3   OTHER THREATS

Even if metacharacter problems form a large set of web application vulnerabilities, there are other problem worth mentioning as well. In this section we will focus on weak authentication routines, access control flaws, and what the dangers of leaking too much information are.

2.3.1   Weak authentication

Weak authentication is major threat to web application security. Custom authentication services often fail to identify all the treats that they face. If there is a single bug in the authentication that allows an attacker to bypass e.g. a password condition, the authentication is useless. There is also an underlying assumption in access control mechanisms that blindly trusts the authenticated subject. If the authentication fails, so does the access control.

Authentication in web applications today often means an HTML form requiring a username and a password. To keep this authentication as secure as possible, several precautions should be taken:

Use Secure Sockets Layer (SSL)
If Secure Sockets Layer (SSL) is enabled before any password is submitted, no attacker can gain access to a plain text password by listening to the communication channel.

Do not store passwords in plain text
No password should ever be stored in plain text. If there is a vulnerability in the application that allows an attacker to print out stored data, it would be an easy task to compromise the authentication procedure. It makes sense to practice defense in depth, even when we are storing password.

Use password requirements
To reduce the likelihood of dictionary attacks, it is good practice to implement password restrictions. This means that we should reject passwords e.g. shorter that six characters, not containing at least two upper-case ones.

2.3.2 Access control

A failing access control is almost as bad as a failing authentication. We know who the user is, but is not able to deny or grant access based on this information. There are at least a couple of well-known access control schemes:

- Role-based access control

- Access Control Lists (ACLs)

Role-based access control is the simplest model of these two and flaws are somewhat less likely to occur when it is used instead of Access Control List (ACL). Each user is assigned to one or more roles and access is granted if this role is allowed to access the requested resource. Otherwise, it is denied.

There are some threats related to role-based access control:

- A user is assigned to a wrong role, possibly an administrator role.

- The access control fails such that a user is granted access to an object that is usually protected.

ACLs are slightly more complex because they require more configuration than roles. For each object, an ACL lists users and their permissions. However, the threats are the same as the ones we listed in the previous paragraph, but their likelihood is somewhat larger.

2.3.3 Leaking information to the user

Leaking information imposes a significant threat in all sorts of situations and not only in web applications. Suppose that a security guard should be giving up alarms codes and other security routines that a bank uses to all his/her friends, just because he/she wants to point out how important his/her job is. It would be a fairly easy job for a thief to get his/her hands on this information to compromise the security systems and rob the bank.

We find this situation in web applications too. If our application fails, we should not return complete stack traces to the user. First of all, this does not make any sense to users other than developers or attackers. To make our web application user friendly, a message describing the problem should be returned. The second point is that printing the stack traces give away a lot of information about the internal structures of the system that may be used to identify weak points.

2.4 HOW TO HANDLE METACHARACTERS?

After a couple of sections demonstrating the power of metacharacters, it seems quite clear that these characters need special treatment. In this section we will discuss several ways to remove their magic.

2.4.1   Avoiding SQL-injection

As discussed in section 2.2.2, the SQL-injection problem occurs when the data is leaving the application code and entering a database. This means that we need to handle the metacharacters before they are used by the Database Management System (DBMS). There are at least two different solutions to this problem:

1. Using `PreparedStatements`

2. Handle each metacharacter manually

2.4.1.1   Using prepared statements

To avoid SQL-injections, the easiest solution is to use a construction called `Prepared-Statement`. According to the Java Application Programming Interface (API) documentation [35], a `PreparedStatement` is an

> . . . object that represents a precompiled SQL statement. This object can then be used to efficiently execute this statement multiple times.

But how does this ensure that all SQL-injection attempts are avoided? Let us look at the example given in appendix E.

In this example we create a `PreparedStatement` object that contains our SQL-query. We insert placeholders (e.g. '?') where we want to append user input later on. This object now represents the control information of the query. Then, to include the data given the user in the query, we use the setters (e.g. setString(int parameterIndex, String string)) of the `PreparedStatement` object. By using such an object we have separated the control information and the user data, and any SQL metacharacter (e.g. ';') given by the user will always be treated as just plain text by the database system.

2.4.1.2   Handle each metacharacter manually

Instead of using `PreparedStatement`, we may choose or be forced to handle the metacharacters of a system manually. This is the case if

1. We are using a programming language that does not support `PreparedStatements`

2. We are talking to a database that does not support `PreparedStatements`

First of all, we need to read the documentation thoroughly through to identify all the metacharacters used by the system we are sending our data to. Then, we have to develop a strategy to handle every single one of them. This does usually mean some kind of escaping. However, we recommend that `PreparedStatements` are always used when they are available because it is quite easy to forget only a single metacharacter unhandled, leaving the application vulnerable to attackers.

2.4.2 Avoiding Cross-site scripting

In the same way as SQL-injections (section 2.2.2), cross-site scripting problems occur when user input leaves the web application code and enter another software subsystem. This time, the problem is not related to DBMSs and SQL-queries, but to web browsers accepting scripts. However, the solution to the problem is more or less the same as the one we discussed in section 2.4.1.2. We have to identify all the metacharacters used by the web browser and then escape them somehow to remove their "magic". In HTML this escaping is called HTML encoding.

[31] sketches a quite simple algorithm for HTML encoding:

1. Map every occurrence of `&` to `&amp`.

2. Then replace every `"` with `&quot`.

3. Then every `<` with `&lt`.

4. And finally replace every `>` with `&gt`.

It ensures that all the HTML- or scripting code given by the user is only represented as plain text and nothing more by the web browser. If we want to allow our users to format their input by using a restricted set of HTML tags, selective filtering is possible. Sverre H. Huseby writes more about this in [31].

2.4.3 Avoiding metacharacter problems in general

Metacharacter problems are closely related to software systems that use metacharacters. Earlier in this chapter, we have discussed two types of such problems - SQL-injections and cross-site scripting problems - and how they should be handled. But what do we do if our web application accesses a subsystem that defines a custom set of metacharacters?

Based on [31]'s elaborations on metacharacter handling we suggest the following approach:

1. Identify all metacharacters that the system uses. The system manual may be a good place to start.

2. Then for each metacharacter:

    (a) Escape it if it makes sense as a plain character

    (b) Otherwise remove it.

As we illustrate in figure 2.2, this algorithm should be run every time data is passed from our application to a subsystem to avoid metacharacter problems.

2.5 HANDLING OTHER THREATS

Handling weak authentication, access control and information leaks is somewhat more difficult than dealing with metacharacters. They form limited sets that may be identified and escaped properly. Detecting weak authentication procedures is not that straightforward.

[65] discusses several attacks (e.g. replay attacks, persistent logins)towards authentication and access control functionality and suggest how a developer may implement PHP functionality to counter these treats. These principles are not language-specific and may be used in Java web applications as well. However, the
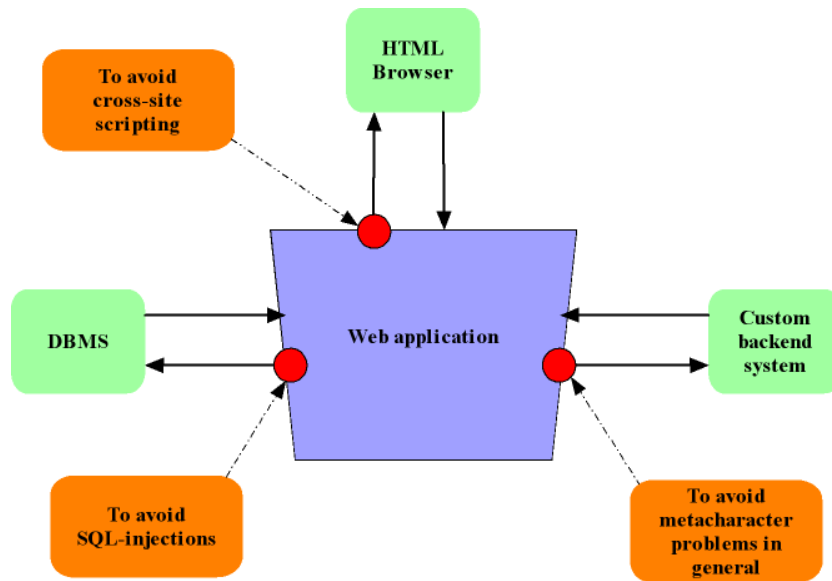
Figure 2.2: Handling metacharacters when they leave the application code

best solution to these problems is probably to develop and use well tested authentication and access control features implemented by web application frameworks. These modules tend to contain less bugs and be more secure than custom solutions.

Leaking information is another fuzzy threat, but it can easily be countered by web application frameworks if they dump stack traces to log files instead of in browser windows. However, it should be possible to override this setting because it could lead to more efficient development.

2.6 SUMMARY

In this chapter we have discussed several security pitfalls found in web applications through both explanations and examples. We have focused on metacharacter problems like SQL-injections and cross-site scripting problems, but also weak authentication, bad access control and information leaks. The last couple of sections concentrate on how these problems may be solved, or at least reduced.

<div align="right">

CHAPTER 3

</div>

WEB APPLICATION FRAMEWORKS

In this chapter we will discuss what a framework is, and how frameworks differ. Then, we look at three frameworks and the architectural patterns they use. As we stated in the introduction, we focus on Spring, Struts and JavaServer Faces, which are three popular and widely used Java/Java EE software frameworks.

3.1 WHAT IS A FRAMEWORK?

[32] defines a framework as

> ... a reusable, semi-complete application that can be specialized to produce custom applications.

The idea is to provide the developers with a common and reusable structure that may serve as a foundation for their software applications. Additionally, [32] states three important framework characteristics:

1. A framework is known to work well in several applications.

2. A framework is ready to use in the next project.

3. A framework can be used by other developer teams in the organization.

Application frameworks may be divided into several categories according to their functionality and architecture. The Spring framework (section 3.3) is, as they state at their homepage a:

> ... leading full-stack Java/Java Enterprise Edition (Java EE) application framework.

This means that Spring itself is not only designed to be used when developing web applications, but also other software applications. In this setting, full-stack points to the application stack. Spring provides support for all the layers in a typical web application including database access layer, business logic layer and the web layer.

Struts (3.4) and JavaServer Faces (JSF) (3.5) represent another category of application frameworks. In contrast to Spring, these frameworks do not try be a full-stack frameworks. This means that they do not include support for e.g. database persistence. Instead they focus on developing abstractions on top of the Servlet API only. More details on each of these frameworks are found in the following sections.

3.2 MVC/MODEL 2

Model View Controller (MVC) is a well-known design pattern. It is widely used in a lot of software applications (e.g. Java Swing [58]) and is an acronym for:

Model
> A model is representation of the data for an application.

View
> A view is the visual representation of the data.

Controller
> A controller ties the model and the view together by translating view changes into model changes.

This pattern does not fit the request-response behaviour of HTTP and web application in general, so it is somewhat changed into a new pattern called *Model 2*.

An illustration of the *Model 2* pattern [45] is given in figure 3.1. From this figure
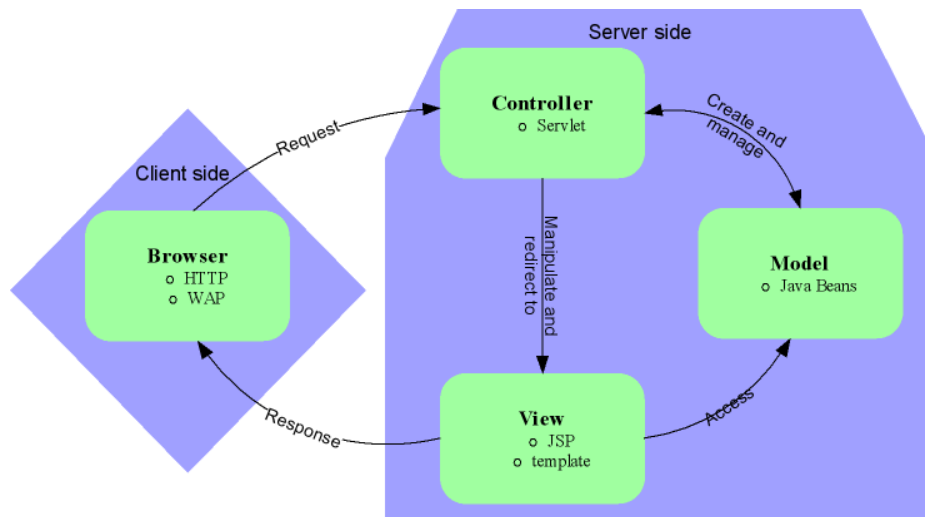
Figure 3.1: The Model 2 pattern

we see that there is still a model, a view and a controller and that they are all placed on the server side of the web application. The controller receives the requests made by the web browsers. Then, it manipulates the model according to the request, before redirecting to the view which is returned to the user.

One of the consequences of this architectural pattern is that an error in a JSP does not affect the model or the controller. On the other side, if there is an error in the model, this error does not affect the application code or the JSP. This architecture also allows unit testing of each component and different developers may work at the different MVC-components simultaneously.

3.3   SPRING

As we quoted in 3.1, Spring [73] tries to play the role of the leading full-stack Java/Java Enterprise Edition (Java EE) framework. In this section we will discuss Spring's architecture and some important patterns it uses to play this role. We will show how these patterns influence the security level, when discussing Spring security in chapter 4.

3.3.1 Spring architecture

Spring's architecture is discussed in a lot of books (e.g. [73]) and articles (e.g. [7]). It consists of seven clearly separated modules that are build on top of the Spring Core. This is shown in figure 3.2. There are no dependencies between the
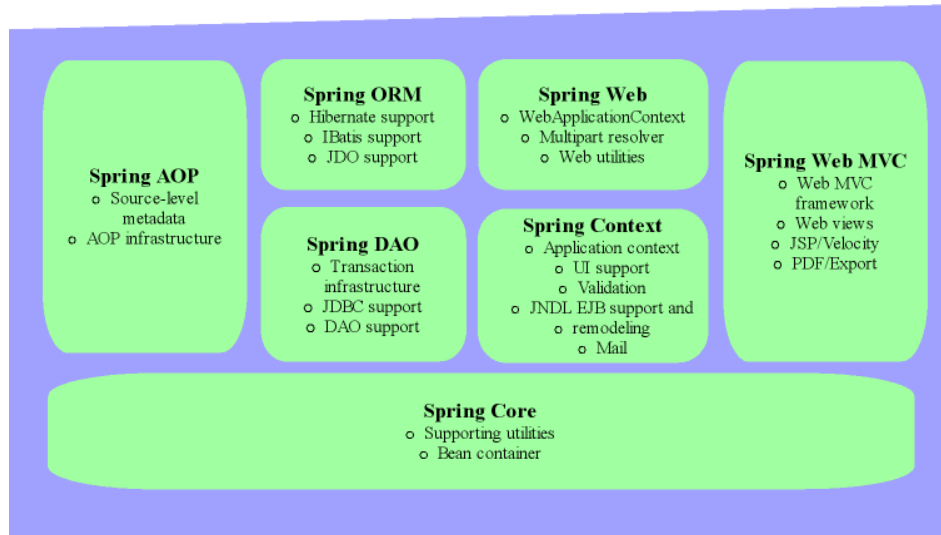
Figure 3.2: Spring architecture

illustrated modules, so each developer may freely choose the modules that fits the current project. The responsibility of each module is described by [73] as:

Spring Core

The functionality found in Spring core, forms the heart of every Spring application. Within this module there is a `BeanFactory` that instantiates the beans described in the application's configuration files and then, injecting the dependencies into each bean by using the IoC pattern. More details on this process are found in section 3.3.2.

Spring AOP

Spring AOP provides support for aspect-oriented programming in Spring applications. The classes found in this module, is based on the API defined by AOP Alliance [6] to ensure that Spring AOP is compatible with other AOP implementations. A developer may base his or her custom aspects on the functionality found in this module. AOP is discussed in more details in section 3.3.3.

Spring DAO

Spring DAO offers a Java Database Connectivity (JDBC) abstraction that takes care of the JDBC code needed to perform database queries. This includes opening `Connections`, creating `Statements`, iterating over `ResultSets` and, finally, closing `Connections`. This means that the developer may focus on the business logic instead of boilerplate JDBC code.

Spring ORM

This module is included in Spring to support object/relational mapping tools (e.g. Hibernate [39]) if such a tool is preferred instead of plain JDBC.

Spring Context

Because of the BeanFactory found in the Spring Core module, we may think

of Spring as a container. The Spring Context module expands this container into a framework by adding support for internationalization, application life cycle events and validation. This module also adds support for Enterprise Java Beans (EJB) integration, remoting and template frameworks like Velocity.

Spring Web

The Spring Web module is deployed on the top of the Spring Context module offering a context that is suitable for web applications. For instance, this includes tasks like binding request parameters to domain objects. This module also offers support for MVC frameworks like Struts, Tapestry, JSF and Web-Work.

Spring Web MVC

The Spring Web MVC module offers a full-featured MVC framework for web application development. Unlike Struts, JSF and other MVC web frameworks, Spring MVC relies heavily on Inversion of Control. Additionally, Spring MVC is closely integrated with other Spring services like internationalization and validation. More details on Spring MVC is found in section 4.2.2.

As you can see, Spring offers integration with several different other frameworks at different levels. At the bottom of the application stack we find Spring's persistence layer which is often represented by the Spring ORM or the Spring DAO modules. Spring ORM offers hooks for popular object/relational mapping frameworks like Hibernate [39], Java Data Objects (JDO) and iBATIS SQL Maps [14]. At the other end of the stack we find the web- and user interface layers. Spring Web MVC provides a complete module for these two layers, but, as stated earlier, Spring Web also contains integration with other frameworks.

When we discuss Spring security in chapter 4, we will concentrate on a plain Spring application using Spring modules only. This means our discussion is centered around the Spring DAO and the Spring Web MVC modules. The details are found in sections 4.2.1 and 4.2.2, respectively.

We have now explained the seven Spring modules in short. In the next section we discuss Inversion of Control (IoC), which is a very important design pattern in Spring.

3.3.2 Inversion of Control

Inversion of Control or dependency injection, as the pattern is named by [24], is a key pattern in Spring. Any non-trivial application consists of several classes that depend on each other to perform some business logic. When implementing applications, we usually let the objects themselves obtain references to the objects it collaborates with. This often leads to highly connected code that is difficult to test. Martin Fowler, uses a simple example called `MovieLister` to illustrate these points [24]. The `MovieLister` class and the related classes are given in listing 3.1.

Listing 3.1: MovieLister

```
1  package moviefinder;
2
3  import java.util.Iterator;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  public class MovieLister
8  {
```

```
 9    private MovieFinder movieFinder;

10

11    public MovieLister()
12    {
13      movieFinder = new ColonDelimitedMovieFinder("movies.txt"
          );
14    }

15

16      public Movie[] moviesDirectedBy(String arg) {
17          List allMovies = movieFinder.findAll();
18          for (Iterator it = allMovies.iterator(); it.hasNext
              ();) {
19            Movie movie = (Movie) it.next();
20            if (!movie.getDirector().equals(arg)) it.remove
                ();
21          }
22          return (Movie[]) allMovies.toArray(new Movie[
              allMovies.size()]);
23      }

24

25      public interface MovieFinder
26      {
27        public List findAll();
28      }

29

30      public class ColonDelimitedMovieFinder implements
          MovieFinder
31      {
32        private String filename;

33

34        public ColonDelimitedMovieFinder(String filename)
35        {
36          this.filename = filename;
37        }

38

39        public List findAll()
40        {
41          List l = new ArrayList();
42          /*
43           * Map the contents of the file named filename
44           * to movie objects and add them to l.
45           */
46          return l;
47        }
48      }
49  }
```

In this example `MovieFinder` is an interface that is implemented by the class
called `ColonDelimitedMovieFinder`. This interface decouples the `MovieFinder`
from the `MovieLister` and may only change to a different implementation of `MovieFinder`
to read data from another source, e.g. an eXtensible Markup Language (XML)-file
or a database. The most important line in listing 3.1 is line 13. This line illustrates
the exact opposite to IoC - the `MovieLister` itself is instantiating a `MovieFinder`
object.

Listing 3.2: MovieLister using the IoC pattern

```java
package moviefinder;

import java.util.Iterator;
import java.util.ArrayList;
import java.util.List;

public class MovieListerIoC
{
  private MovieFinder movieFinder;

  public void setMovieFinder(MovieFinder movieFinder)
  {
    this.movieFinder = movieFinder;
  }

    public Movie[] moviesDirectedBy(String arg) {
        List allMovies = movieFinder.findAll();
        for (Iterator it = allMovies.iterator(); it.hasNext
            ();) {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg)) it.remove
                ();
        }
        return (Movie[]) allMovies.toArray(new Movie[
            allMovies.size()]);
    }

    public interface MovieFinder
    {
      public List findAll();
    }

    public class ColonDelimitedMovieFinder implements
        MovieFinder
    {
      private String filename;

      public void setFilename(String filename)
      {
        this.filename = filename;
      }

      public List findAll()
      {
        List l = new ArrayList();
        /*
         * Map the contents of the file named filename
         * to movie objects and add them to l.
         */
        return l;
      }
    }
```

```
49   }
```

Listing 3.2 shows an IoC implementation of the functionality found in listing 3.1. Now there are no line stating `new ColonDelimitedMovieFinder("movies.txt")`, but instead we need to provide the container with an XML file describing the relationship between the objects. In Spring applications, this XML file may look like listing 3.3.

Listing 3.3: Wiring Spring beans
```xml
1  <?xml version="1.0" encoding="UTF−8"?>
2  <!DOCTYPE beans PUBLIC "−//SPRING//DTD BEAN//EN"
3          "http://www.springframework.org/dtd/spring−beans.dtd
              ">
4
5  <beans>
6    <!−− Business objects start −−>
7      <bean id="movieLister" class="moviefinder.MovieLister">
8          <property name="movieFinder">
9              <ref bean="movieFinder"/>
10         </property>
11     </bean>
12     <bean id="movieFinder" class="moviefinder.
          ColonDelimitedMovieFinder">
13         <property name="filename">
14             <value>movies.txt</value>
15         </property>
16     </bean>
17     <!−− Business objects end−−>
18 </beans>
```

When this example is about to be deployed, the container resolves that `movieLister` bean depends on the `movieFinder` bean. This means that the container have to instantiate the `movieFinder` before the `movieLister`. The execution steps are:

1. Instantiate the Java Bean named `movieFinder`.

2. Inject the filename property by calling `setFilename("movies.txt")`.

3. Instantiate the Java Bean named `movieLister`.

4. Inject a reference to the `movieFinder` bean by calling `setMovieFinder()` and give a reference to the movieFinder as parameter.

In this revised example we see that the `MovieLister` class does not look up references to their dependencies themselves. Instead the container inject the dependencies as they are defined in the XML configuration file and this is exactly what IoC is about.

3.3.3 Aspect-Oriented Programming

[73] states that Aspect-Oriented Programming (AOP) is often defined as:

> ...a programming technique that promotes separation of concerns within a software system.

The background for introducing aspect-oriented programming is quite clear. Each software system often consists of several components where are responsible for carrying out a limited set of functions. As the application evolves, the responsibility tends to increase beyond their core functionality. Functionality like logging, transaction management and security is often included in business objects that have other concerns simply because there no other way to make this functionality available to the objects. AOP allows this functionality to be separated from the business objects in independent modules.

To understand how Spring implements AOP, there are a couple of terms that need further explanation [73]:

Aspect

An aspect is the functionality that we are implementing. The most common example is logging. Such an aspect allows us to do extensive logging throughout our application without weaving logging code into business objects.

Advise

An advise is the implementation of the aspect. If we use our logging example, the advise is the code that e.g. writes the logging information to a file. Advises are plugged into our applications at joinpoints.

Joinpoint

Joinpoints are places in our code where advises may be plugged in. This may be a method being called, an exception being thrown or a field being modified. However, Spring only supports method joinpoints which means that an advise run before, after or before and after a specified method has been run.

Pointcut

A pointcut defines at which joinpoints the advise should be executed. We may choose freely among the available joinpoints.

To illustrate these concepts, we will give an example. We will extend the example given in section 3.3.2 to illustrate how a logging module may archived through AOP.

To develop a logging aspect we first need to decide at what joinpoint(s) we want our advice to be plugged in, and then write the advice implementation itself. One trivial implementation is given in listing 3.4.

Listing 3.4: LogMovieAdvice

```
1  package moviefinder;
2
3  import java.lang.reflect.Method;
4
5  import org.springframework.aop.MethodBeforeAdvice;
6
7  public class LogMovieAdvice implements MethodBeforeAdvice
8  {
9    public void before(Method arg0, Object[] arg1, Object arg2
         ) throws Throwable
10   {
11     System.out.println("*** Logging start ***");
12     System.out.println("*** Intercepted method:"+arg0+" ***"
           );
13     System.out.println("*** Logging end ***");
14   }
15 }
```

This class implements the `MethodBeforeAdvice` interface which means that we want our advice to be executed before the intercepted methods execute (e.g. the before-joinpoint).

Then, to wire the aspect into our application, we have to change our XML configuration file. The revised version is given in listing 3.5.

Listing 3.5: Wiring a logging aspect into our application

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd
            ">

<beans>
  <!-- Business objects start -->
    <bean id="movieLister" class="moviefinder.MovieListerIoC
        ">
        <property name="movieFinder">
            <ref bean="movieFinder"/>
        </property>
    </bean>
    <bean id="movieFinderTarget" class="moviefinder.
        ColonDelimitedMovieFinder">
        <property name="filename">
            <value>movies.txt</value>
        </property>
    </bean>
  <!-- Business objects end -->

  <!-- Advices start -->
    <bean id="logMovieAdvice" class="moviefinder.
        LogMovieAdvice"/>
  <!-- Advices end -->

  <!-- Advisor pointcut definition for before advice start
          -->
    <bean id="movieFinder" class="org.springframework.aop.
        framework.ProxyFactoryBean">
      <property name="proxyInterfaces">
        <value>moviefinder.MovieFinder</value>
      </property>
      <property name="interceptorNames">
        <list>
          <value>logMovieAdvice</value>
        </list>
      </property>
      <property name="target">
        <ref bean="movieFinderTarget"/>
      </property>

    </bean>
  <!-- Advisor pointcut definition for before advice end--
        >
</beans>
```

The most important changes are the advice- and advisor pointcut beans we have introduced. The `logMovieAdvice` bean is the advice itself, doing nothing special. However, notice that we have renamed to original `movieFinder` bean to `movieFinder Target` and then created a new `ProxyFactoryBean` named `movieFinder`. This is the bean that wires our logging aspect into the application and it contains three important properties:

proxyInterfaces
> A proxy interface is an interface that Spring's `ProxyFactoryBean` will masquerade as. This allows us to get the proxy bean from the application context and casting it to a `MovieFinder` object even if it is an instance of the `ProxyFactoryBean` found in the org.springframework.aop.framework package.

interceptorNames
> A interceptor name is the name of a advice bean that is executed when a method is intercepted.

target
> A target is the class that is being adviced. This is usually a custom class written by the developer or a third-party class that is extended by e.g. a logging aspect.

We have now discussed two of the most important patterns found in the Spring framework. They form the basis of all Spring applications including Acegi Security System that we will discuss in section 4.3.

3.4 STRUTS

Struts is another software framework designed to support Java web application development. Like Spring MVC and JSF, it is based on the well-known MVC pattern variant called Model 2 (see section 3.2). But this is almost everything these frameworks have in common. Whereas Spring tries to fill the role as a full-stack Java application framework, Struts is only concerned with the web-layer of the applications [32]. This leads to somewhat different security concerns in Struts, than in Spring. We will come back to the security details of Struts applications in chapter 5. In the following sections we will discuss the Struts model, view and controller in more detail.

3.4.1 Struts Model

The model in the MVC pattern represents the data. Struts does not provide any classes or interfaces that may be used as part of the model. It is up to the developer to implement a model that is based on data from a data source like a file or a database. There are several frameworks available that may support a model implementation and among them we find:

Hibernate ORM
> Hibernate is an object/relational mapping that tries to resolve the mapping mismatch between the object-oriented paradigm of the today's programming languages and the relational databases. Hibernate recognizes typical tasks that developers are required to implement in every project which involves some sort of DBMS, and tries to automate these. This includes creating,

reading, updating and deleting domain objects and handling their dependencies. These properties complement the functionality found in Struts and make Hibernate, but also other Object Relational Mapping (ORM) mapping tools, good partners to Struts when implementing a data model. More details are found in [8, 39].

Apache Commons Scaffold

Apache Commons Scaffold is not an ORM mapping tool. It is rather a set of reusable software components that often are needed in web applications. Commons Scaffold provides several classes that are designed for Struts applications, and among these we also find classes that eases the job when implementing a model based on data from one or several databases. This solution is far less sophisticated than ORM mapping tools, but it is still an alternative. More details are found on the project home page [19].

JDBC

Java Database Connectivity (JDBC) is the simplest and most demanding way of implementing a model based on data stored in a database. However, we do not recommend using plain JDBC in larger projects both because there are more secure solutions with high performance available and because plain JDBC is more resource demanding.

These are only some of the available solutions. We have mentioned these three to indicate that is up to the software architects to choose what is best suited in every project. The important point to notice is that Struts does not place any restrictions on which technique to choose.

3.4.2   Struts View

The view in the MVC pattern is responsible for rendering the data stored in the model. When the model is changed, the view should be rerendered. Because Struts is a web framework based on Java, JSP taglibs are usually used in the view components. Velocity templates [21] and JavaServer Pages Standard Tag Library (JSTL) are also supported, but we will concentrate on the taglibs provided in the Struts distribution.

The taglibs provided by Struts are divided into four distinct libraries:

1. **bean** taglib

2. **html** taglib

3. **logic** taglib

4. **nested** taglib

where each taglib is responsible for a limited set of tags.

The first taglib is called **bean** mainly because it provides tags that allow bean manipulation in JSPs. This taglib allows a developer to create variables based on HTTP headers, request parameters or cookies, create new JavaBeans based on requests, determining the number of elements in a Collection or Map object, localizing labels and, finally, print property values of beans found in the request, the session or the application scope.

The **html** taglib provides a set a tags that is needed to implement dynamic web pages based on JSP. This include tags like checkboxes, radio buttons, text fields, text areas, hidden fields and so on.

27

The Struts taglib called **logic**, contains tags that represents logic operations. This include testing if values are equal, less then, greater than, empty or present, but also tags representing iteration over data structures and tags for redirection and request forwarding.

The last taglib is called **nested**. It was introduced in Struts 1.1 to provide better support for nested properties. This means that if we have a Blog bean containing a List of BlogEntry beans, we may iterate and access the blog entries by using the . notation (e.g. blog.blogentries) which leads to more easy-to-follow JSP code.

This is the main components of the view provided with Struts. We will discuss some security details of these taglibs in chapter 5.

3.4.3 Struts Controller

A large part of the code found in the Struts framework, is related to the controller. The main reason for this is that a lot controller tasks (e.g. client request processing, form submission) require almost the same processing. Struts recognizes the differences and allow the developer to customize through XML configuration and by extending different Struts framework classes.

To implement its functionality, Struts divides the controller into several components which are responsible for their own limited set of tasks:

ActionServlet
    The `ActionServlet` receives state changes through interactions and passes these changes to other components of the controller.

ActionMappings
    The actual state change event.

ActionForms
    The data for the state change.

Actions
    The `Actions` interact with the model to process state changes and point out the next view that the `ActionServlet` should select.

ActionForwards
    `ActionForwards` form the set of the states that a user may select from.

Let us look at an example to illustrate how these components interact to create and send a response to a client request.

When we are deploying and loading a Struts application in a Servlet container (e.g. Apache Tomcat), the container looks for a file called `web.xml`. It describes the Servlet class that the container should pass the client requests to, the Uniform Resource Locator (URL) patterns that should be interpreted by Struts's ActionServlet and the path of the Struts configuration files. A simple example file is given in listing 3.6.

Listing 3.6: web.xml for a simple Struts application

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
       Application 2.3//EN'
3    'http://java.sun.com/dtd/web-app_2_3.dtd'>
4
5
6  <web-app>
```

```
7      <servlet>
8      <servlet−name>blog−struts</servlet−name>
9      <servlet−class>org.apache.struts.action.ActionServlet</
           servlet−class>
10
11     <init−param>
12          <param−name>config</param−name>
13          <param−value>/WEB−INF/conf/struts−config.xml</param−
               value>
14     </init−param>
15
16     <load−on−startup>1</load−on−startup>
17     </servlet>
18
19     <servlet−mapping>
20     <servlet−name>blog−struts</servlet−name>
21     <url−pattern>∗.do</url−pattern>
22     </servlet−mapping>
23 </web−app>
```

Struts is triggered by a request sent by a client. If this request matches the pattern defined in web.xml (i.e. *.do), it is forwarded to ActionServlet defined by Struts. Then, the ActionServlet looks up the mapping of ActionMapping (e.g. login) defined in struts-config.xml (see listing 3.7).

Listing 3.7: struts-config.xml for a simple Struts application

```
1  <?xml version="1.0" encoding="UTF−8"?>
2  <!DOCTYPE struts−config PUBLIC
3     "−//Apache Software Foundation//DTD Struts Configuration
           1.3//EN"
4     "http://struts.apache.org/dtds/struts−config_1_3.dtd">
5
6  <struts−config>
7      <form−beans>
8      <form−bean name="loginForm" type="org.apache.struts.
           validator.DynaValidatorForm">
9          <form−property name="username" type="java.lang.
               String"/>
10         <form−property name="password" type="java.lang.
               String"/>
11      </form−bean>
12     </form−beans>
13
14     <global−forwards>
15     <forward name="/login" path="login.do" />
16     </global−forwards>
17
18     <action−mappings>
19     <action path="/login"
20          type="blog.actions.LoginAction"
21          validate="true"
22          input="/WEB−INF/jsp/login.jsp"
23          />
24     </action−mappings>
```

29

```
25
26     <global−exceptions>
27       <exception    type="java.lang.NumberFormatException"
28             path="/WEB−INF/jsp/errors/numberformat.jsp"/>
29
30       <exception    type="java.lang.Exception"
31             path="/WEB−INF/jsp/errors/exception.jsp"/>
32     </global−exceptions>
33   </struts−config>
```

Before the execution continues, several ActionMapping properties are investigated by the ActionServlet:

1. If a form bean is specified in name attribute of the ActionMapping, this bean is instantiated.

2. If the validate property is set to true, then the ActionServlet calls `validate()` on the form bean.

3. Then, if validate() returns at least one ActionError, the ActionServlet forwards the user to the path specified by the input attribute (e.g. `login.jsp` in listing 3.7) and the request-response sequence is ended.

4. Otherwise, if `validate()` does not return ActionErrors, the ActionServlet looks up the Action specified by the ActionMapping. If it is not already instantiated, the ActionServlet creates the Action object this point.

5. The next step is to call `execute()` on the Action object. This method is often overridden by the developer and it typically creates, read, updates or delete business objects.

6. When the `execute()` method has completed, it returns an ActionForward to the ActionServlet. If the ActionForward indicates a new URL that should be processed by Struts, we start all over again. Otherwise, if it is a JSP, then this page is rendered and returned to the user. This ends the request-response sequence.

These are the basic steps in Struts when processing a client request. We will give some more details connected to this framework when we are discussing how security is ensured in chapter 5.

3.5   JAVASERVER FACES

JSF is a component-based Java/Java EE web application framework specification developed by Sun Microsystems [37, 45]. The process of developing this specification was started back in 2001 through Java Specification Request (JSR) 127. Several major software vendors have participated in this process, hopefully leading to a web application framework that is widely adopted.

At an early stage of the specification process three key design goals were stated:

1. JSF should use the MVC pattern variant called Model 2 (see section 3.2).

2. The JSF view component should not rely on any particular display technology (e.g. JSP).

3. JSF should be based on components using events and listeners to send and receive events.

These goals form the basis of what we today know as JSF.

3.5.1 Components, components and components

Like we mentioned in the introduction to this section, JSF is component-based. If we think of links, forms and other HTML entities, we see that they are all represented as components in JSF. Depending on its goal, a HTML <a href="" /> tag may be represented as both a HtmlCommandLink or a HtmlOutputLink JSF component. Each view rendered by JSF is formed by a tree of components where the root is always an instance of UIViewRoot.

For each component to complete its tasks, it is associated with a set of objects where each of them is delegated more specific tasks. A component usually depends on:

a Renderer
>   The `Renderer` is responsible for displaying a component and translating input from users into component values. Several `Renderers` form a `RenderKit`. JSF is shipped with a default `RenderKit` for HTML 4.01

a Validator
>   The validator is responsible for ensuring that values given by users are acceptable.

a Converter
>   The `Converter` is responsible for converting an object to a string when the object is viewed, and from a string to an object when user input is processed.

one or more backing beans
>   The backing beans collect values from JSF components and implement event listeners.

3.5.2 Event management

In opposite to Spring and Struts, a developer is not required to deal with plain form submissions in JSF. Instead the framework defines an event- listener model similar to the model found in Swing [58]. When a JSF button or link is clicked, an event is generated. Such events are then distributed to the registered listeners, and appropriate actions are executed. JSF defines four groups of events:

Value-change events
>   Value-change events are executed when values in input components are changed. A ValueChangeListener should implement a method with a signature like `public void <<name>>(ValueChangeEvent e){}`.

Action events
>   Action events may be divided into two subgroups: actions events that implicate navigation to another view and action events that do not. The first group of events requires a method signature like `public String <<name>>(){}` while the latter requires a signature on form `public void <<name>>(ActionEvent e){}`.

Data model events
>   Data model events are fired when "a data-aware component processes a row". This means that these events are generated when e.g. a row is selected in a HtmlDataTable instance.

Phase events

As we discuss in section 3.5.5, each request that is passed to the JSF Servlet is processed in, at most, six phases. Phase events are generated by JSF before and after each of these phases. Phase events may be used if we e.g. instantiate backing beans before a view is displayed.

The reason for mentioning the method signatures and not the interfaces they are defined in, may not be obvious for developers new to JSF. JSF does not require the listeners to implement the listener interfaces. They are just markers to indicate the correct method signature. Instead, JSF uses the reflection API (module found in the Java Software Development Kit (SDK)) to check if the signatures are on the correct form. This leads to a slightly looser coupling between JSF and the application code.

3.5.3 Navigation

Like we stated in the previous section, navigation is JSF web applications is closely related to actions. When a button or a link is clicked, an ActionEvent is generated. JSF refers to different groups of actions used in navigation:

1. Static navigation

2. Dynamic navigation

3.5.3.1 Static navigation

When a user clicks a JSF button or link that is associated with an action, an event is generated. If the action attribute refers to a string constant defined in the JSFconfig file (i.e. faces-config.xml), JSF refers to this as static navigation. An example is given in listing 3.8.

Listing 3.8: Navigation case when using static navigation

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
      1.0/EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/blogentry/list.jsp</to-view-id>
      <redirect/>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

This listing specifies a navigation case of the JSP called `login.jsp` found at the root of the web application. If we assume that there is a JSF button or link on this page setting its action attribute to "success", the navigation case ensures that the user is redirected (because of the redirect tag) to `list.jsp` in the blogentry directory. As long as the action attribute is not changed, the user is always forwarded to the same JSP without regard to the system state.

3.5.3.2 Dynamic navigation

Dynamic navigation is somewhat different than static navigation. Instead of specifying a constant string as action attribute, we use a reference to an object method available in the application context. This is illustrated in listing 3.9.

Listing 3.9: Navigation case when using dynamic navigation

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
      1.0/EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>*</from-view-id>
    <navigation-case>
      <from-action>#{createBlogEntryBean.create}</from-
          action>
      <from-outcome>success</from-outcome>
      <to-view-id>/blogentry/new.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-action>#{createBlogEntryBean.create}</from-
          action>
      <from-outcome>failure</from-outcome>
      <to-view-id>/error/blogentry_error.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

In this example we are able to use navigation based the overall system state. If createBlogEntryBean.create() returns the string "success", the user is forwarded to /blogentry/new.jsp. Otherwise, if the outcome is "failure", /error/blogentry_error.jsp shows an appropriate error message. Because JSF places no restrictions on the number of allowed navigation cases, we may specify as many as we need.

3.5.4 Backing beans

If we compare the backing beans found in JSF to the ActionForms and Actions found in Struts, we see that the backing beans are some sort of hybrids of these objects. The backing beans used in JSF provides both data and operations, while ActionForms represent the data in Struts and the Actions provide the operations. An example backing bean instantiated as a managed bean is illustrated in listing 3.10.

Listing 3.10: Declaring a backing bean as a managed bean in faces-config.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
      1.0/EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

```

```
 6  <faces−config>
 7     <managed−bean>
 8        <description>Used to implement login and logout</
             description>
 9        <managed−bean−name>authenticationBean</managed−bean−name
             >
10        <managed−bean−class>beans.AuthenticationBean</managed−
             bean−class>
11        <managed−bean−scope>session</managed−bean−scope>
12        <managed−property>
13           <property−name>visit</property−name>
14           <value>#{sessionScope.visit}</value>
15        </managed−property>
16     </managed−bean>
17  </faces−config>
```

In this listing a bean named `authenticationBean` is instantiated based on the beans.AuthenticationBean class. This bean is then placed in the session scope, which means that this bean is available as long as the session is valid. Further on, this configuration ensures that the visit property of the `authenticationBean` is properly initialized with a reference to an object named `visit` found in the session context.

3.5.5   Request processing

Each request that is passed from the Servlet container to the JSF Servlet, is processed in at most six phases. We say at most because it is possible to render and send a response in every of these phases, thus bypassing the rest of the processing steps. [45] discusses this phase bypassing in more details.

The processing phase found in JSF are:

1. **Restore view**

2. **Apply request values**

3. **Process validations**

4. **Update model values**

5. **Invoke application**

6. **Render response**

The execution of these phases is also illustrated in figure 3.3.

3.5.5.1   Restore view

As we discussed in section 3.5.1, each view in JSF is a tree of JSF components. In the restore view phase, JSF creates a component tree at the server-side based on the requested view identifier. No action is taken if the requested view is the same as the current view of JSF because then, the component tree is already generated and cached. If the view involves validators, managed beans or listeners, these are restored as well.
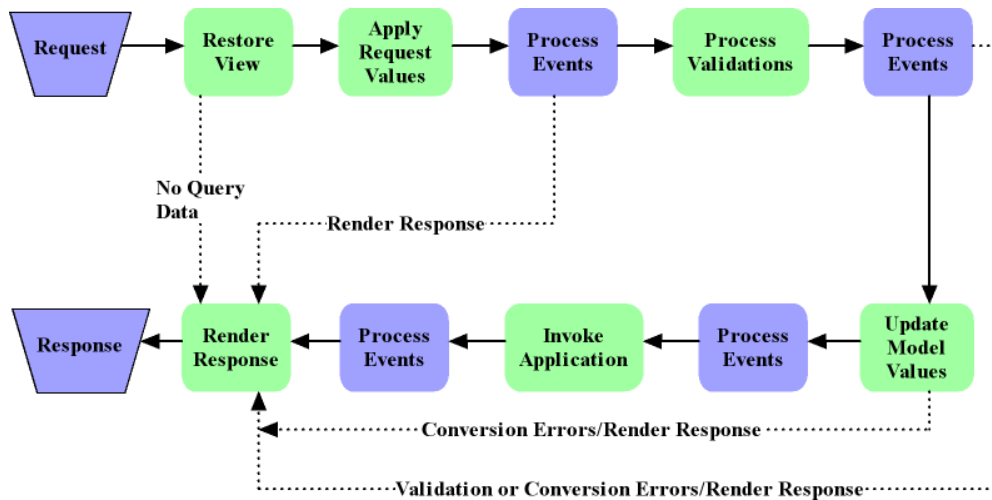
Figure 3.3: JSF request processing

3.5.5.2 Apply request values

The processing in this phase is pretty well described by the phase name. In this phase, JSF iterates over all the components in the component tree and updates their values according to the values found in the request submitted by the user. Let us the JSP source code in listing 3.11 as an example.

Listing 3.11: JSP accessing an AuthenticationBean

```
1  ...
2  <h:inputText  id="usernameInput"
3          value="#{authenticationBean.username}"
4          required="true" />
5  ...
6  <h:commandButton  id="submitButton"
7             action="#{authenticationBean.login}"
8             value="Login" />
9  ...
```

When the apply request values phase for a component tree containing these components is executed, the component associated with the id "usernameInput" is associated with the value given by the user. As we discuss in section 3.5.5.4, all the backing beans properties (e.g. authenticationBean.username) are updated only when the values have passed the validation.

The other main responsibility of this phase, is to add ActionEvents to the event queue if a command button or a command link has been clicked [26]. These events are processed in the invoke application phase (see section 3.5.5.5).

3.5.5.3 Process validations

In the process validations phase, JSF asks every component in the component tree whether they have acceptable values or not. Based on the architecture we sketched in section 3.5.1, each component's validator is main responsible for this validation. If all the given values are acceptable, the processing continues to the next phase. Otherwise JSF jumps to the last phase called render response. Notice that ValueChangedEvents are generated and consumed before the next phase is executed, but after the validation is completed.

3.5.5.4   Update model values

When the JSF execution enters the update model values phase, we know that the user has given acceptable values and JSF is ready to update the properties of the backing beans. To look up references to the backing bean properties, JSF uses the value attribute associated with each component. If we look back to the example given in listing 3.11, this means that setters similar to authentication-Bean.setUsername(...) is called during the update model values phase. Finally, data model events are generated and distributed to its listeners before the invoke application phase is initiated.

3.5.5.5   Invoke application

When entering this phase, JSF knows that its model- and component values are up-to-date, and this is the time when the business logic should be executed. In this phase, JSF consumes the ActionEvents generated in the Apply request values phase. Usually, the receiving listeners often call methods on other Java objects to do access control, update database entries or do other sorts of processing. Based on the processing results, the value of the action attribute is finally reduced to a string value indicating the next page to load.

3.5.5.6   Render response

The last phase is called render response because of quite obvious reasons. Its main responsibility is to generate and send a response back to the user based on updated component- and model values and the application execution. Additionally, the render response ensures that the view is saved before the response is sent to the client. This increases the performance of the web application if the user requests it again due to e.g. invalid input.

3.6   SUMMARY

We have now discussed the architectural patterns used in Spring, Struts and JSF. Our focus has been placed on how different components in frameworks cooperate and how they process incoming request. These discussions form our basis when we, in the next chapter, discuss how each of them perform in terms of typical web application vulnerabilities like the ones discussed in chapter 2.

4

SPRING SECURITY

This chapter touches important topics connected to the security level of Spring web applications. The first sections explains the functionality found in Spring itself in details and what issues Spring does not address. In the last one we discuss the Acegi Security System and how this package integrates with Spring and addresses some of its issues.

4.1 ERROR HANDLING

Errors happen all the time in all sorts of software including web applications. All these errors should be taken care of in a secure manner because:

1. A software application returning error details is not very user-friendly. If the user entered something that the application cannot handle, this should be reported. Based on this, the user may change his or her input to something that is accepted.

2. An application printing the complete stack trace when failing, is leaking a lot of information that may be useful to an attacker. We do not want to provide the him/her with this information because it may describe the internal architecture of our software, making it a lot easier to an attacker to identify weak points. Not leaking error details is also support by an important security principle stated by Viega and McGraw in [70]: "Fail securely".

In Java/Java EE generally, and Spring specially, error handling often means catching exceptions. The Spring Web modules provides a class called `SimpleMapping ExceptionResolver` that is used to catch unhandled exceptions. When instantiating this bean, the `exceptionMappings` property is set. Each of these mappings state what `Exceptions` it handles and to what view the user should be redirected when this exception is thrown. Given this approach, it is possible to redirect to error pages explaining the error to the user without giving away too much details to possible attackers. Thus, Spring addresses both issues mentioned in the beginning of this section.

4.2 SECURITY IN SPRING MODULES

In this section we will discuss how security is ensured in two of the seven Spring modules. These modules are Spring Data Access Objects (DAO) and Spring MVC. We have chosen to concentrate on these modules because they are responsible for passing data to other subsystems (e.g. DBMS, web browser) when developing pure Spring application. If we take a look back to section 2.2.1, we see that this is exactly where metacharacter problems occur and where they need to be taken care of. We

will discuss each module in some detail, before we investigate what they do to avoid metacharacter problems.

4.2.1 Spring DAO

In general, a DAO is an object-oriented way to read data from and write data to a database. The DAO provides an interface through which an application can access a database without exposing what technologies it uses to get the data. This means that we can change data source from an XML-file to a DBMS without changing our application. We only need to switch from one DAO implementation to another. Besides, this architecture allows mockup implementations of the interfaces to speed up testing.

Spring DAO uses Java interfaces, a consistent exception hierarchy and an object called `JdbcTemplate` to create a data access layer that the business objects can access without knowing anything about the underlying technologies. We will discuss the Spring DAO details throughout this section, starting with the exceptions.

4.2.1.1 Exceptions

Spring's first step to create a data access layer is to provide an exception hierarchy. These exceptions encapsulate all technology specific exception into a limited set of Spring DAO exceptions which extends the `DataAccessException`. According to [73], these objects have some important properties:

1. All `DataAccessExceptions` are `RuntimeExceptions`. This means that we are not forced to handle these exceptions by means of `try - catch` blocks.

2. All `DataAccessExceptions` are `NestedExceptions`. This allows us to access the cause of the problem by calling `getCause()` on the `DataAccess Exception`.

As long as we handle the limited set of Spring exceptions by e.g. using the approach sketched in section 4.1, we may use several different data sources in our applications and even add a new data sources without changing a single line of our application code. This makes it easier not to leak information to attackers when changing or introducing new data sources because no new technology-specific exceptions are introduced.

4.2.1.2 Spring's JdbcTemplate

Java Database Connectivity (JDBC) is the traditional way to access database from within Java. It is a clear interface defined in the Java API by Sun Microsystems that makes it easy for database vendors to develop JDBC drivers supporting their products. However, there are several problems occurring when using plain JDBC:

1. Each developer using plain JDBC works closely with the database, but they also have to take care of a lot resource handling (e.g. establishing connections, iterating over `ResultSets`) and exception handling.

2. Plain JDBC requires a lot of boilerplate code each time it is used. In examples given by [73], 70-80% of the code needed to execute a database query is boilerplate code.

3. In plain JDBC, security issues occur more often because the developers are tempted to build SQL-queries dynamically instead of using safer constructs like `PreparedStatements`.

The `JdbcTemplate` found in Spring recognizes these issues and try to provide classes and other abstractions to clean up the JDBC code. The first improvement implemented by means of the `JdbcTemplate`, is the resource administration. When the `JdbcTemplate` is instantiated in an XML-file it is given a reference to a `DataSource` bean. This `DataSource` contains the database connection details such as driver name, URL, username and password. By using this approach, the developers are allowed to focus on writing the application specific queries because resource handling is taken care by the Spring framework.

Spring provides several classes and interfaces that represent traditional database operations like INSERT, UPDATE and DELETE. Some places in our code we execute unique queries and in other modules we execute the same queries over and over again. In the first case, we may simply use the methods (e.g. update(String sql, Object[] params)) defined by the `JdbcTemplate` while in the latter we define the query in a standalone class. Spring offers several ways to execute queries and it is up to the developer to choose the best suited one.

In the light of security, the way the queries are represented and executed is quite important. When the queries are executed, the data leaves our application code and all the metacharacters should be taken care of to ensure that no SQL-injections are possible (see section 2.2.2). Two common ways to execute queries in Spring using the `JdbcTemplate` approach are:

Using jdbcTemplate.query(..) and jdbcTemplate.update(..)
> These methods do not use `PreparedStatements` and, in some contexts (see e.g. appendix F), these methods may be vulnerable to SQL-injections.

Extending the SqlQuery and SqlUpdate classes
> This approach is normally used if the same queries are needed in several parts of an application. However, the approach should always be used because, in opposite to the `jdbcTemplate` methods, they use `PreparedStatements` that avoid SQL-injections. Thus, they lead to a higher level of application security.

The third and last improvement that Spring implements, is connected to the way we usually handle `ResultSets`. In plain JDBC we always need to iterate through all the rows of a `ResultSet`. Spring recognizes this and introduces an interface called `RowCallbackHandler` containing a method named `processRow(java.sql.ResultSet rs)`. A developer only needs to implement this method, stating how to map the data found in each row of the `ResultSet` to domain objects. Then, when a query is executed, a suitable `RowCallbackHandler` implementation is passed a parameter and the corresponding domain object is returned.

Spring also defines some other interfaces that may be suitable when processing `ResultSets`. Some of them are `RowMapper`, `ResultReader`, and `RowMapper ResultReader`. We consider the details of these interface to be out the scope of this discussion, because they do not affect the security of Spring applications in general.

4.2.2 Spring MVC

We use the Spring MVC module when we are developing Spring-based web applications. As the name suggests, Spring MVC puts a MVC layer on top of the request-response centered HTTP. In this section we will focus on the MVC pattern, how incoming request are managed by the Spring framework, some view layer security topics and, finally, how form validation is implemented by Spring.

4.2.2.1 Requests in Spring MVC

In typical web applications, several events happen when a web browser sends a request to a web server. If the user clicked a link to add a new user in a web application, Spring is likely to execute the following steps [73]:

1. The `DispatcherServlet` receives a request containing an URL like "/user/registernew.html". This `DispatcherServlet` is configured in `web.xml`, just like we illustrate in listing 4.1.

2. Then, the `DispatcherServlet` consults the `HandlerMapping` in the application config file to look up a controller whose bean name is `RegisterNewController`. Both the `HandlerMapping`, the controllers, and the `ViewResolver` is defined in a Spring config file. An example is given in listing 4.2.

3. The `DispatcherServlet` dispatches the request to the `RegisterNewController`.

4. The `RegisterNewController` returns a `ModelAndView` object which includes a logical view named `registernew`.

5. The `DispatcherServlet` then consults the `ViewResolver` to find a view whose logical name is `registernew`. This *ViewResolver* usually returns something like `/WEB-INF/jsp/user/registernew.jsp`.

6. At the last step, the `DispatcherServlet` forwards the request to the JSP at `/WEB-INF/jsp/user/registernew.jsp`. This file is rendered and returned to the user.

Listing 4.1: web.xml for a Spring application

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
       Application 2.3//EN'
4   'http://java.sun.com/dtd/web-app_2_3.dtd'>
5
6  <web-app>
7    <servlet>
8        <servlet-name>blog</servlet-name>
9        <servlet-class>org.springframework.web.servlet.
           DispatcherServlet</servlet-class>
10       <load-on-startup>1</load-on-startup>
11     </servlet>
12     ...
13  </web-app>
```

Listing 4.2: Simple Spring config file

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
3   "http://www.springframework.org/dtd/spring-beans.dtd">
4
5  <beans>
6    ...
7    <bean id="simpleUrlMapping" class="org.springframework.web
       .servlet.handler.SimpleUrlHandlerMapping">
8      <property name="mappings">
```

```
 9        <props>
10          <prop key="/user/registernew.html">
              registerNewController</prop>
11          ...
12        </props>
13      </property>
14    </bean>

16    <bean id="registerNewController" class="web.
        RegisterNewController">
17      <!-- Dependency injections... --!>
18      <property name="userService">
19        <ref bean="userService" />
20      </property>
21      <property name="formView">
22        <value>edituser</value>
23      </property>
24      <property name="successView">
25        <value>editusersuccess</value>
26      </property>
27    </bean>

29    <bean id="viewResolver" class="org.springframework.web.
        servlet.view.InternalResourceViewResolver">
30        <property name="viewClass">
31          <value>org.springframework.web.servlet.view.
              JstlView</value>
32        </property>
33        <property name="prefix">
34          <value>/WEB-INF/jsp/</value>
35        </property>
36        <property name="suffix">
37          <value>.jsp</value>
38        </property>
39      </bean>
40      ...
41 </beans>
```

If the origin of the request represents a form submission, then some validation is needed to ensure that the input is within the domain and range expected by our application. This validation takes place before the DispatcherServlet consults the ViewResolver. If the validation fails, the DispatcherServlet looks for the failureView of the controller. Otherwise the successView is fetched. More details on Spring's validation approach is found in section 4.2.2.3.

When the user input has passed the validation, the remaining steps described above are executed. When the JSP rendering is completed, the JSP is returned to the user. It is at this point we need to take care of all the metacharacters, to make sure that all web browser metacharacters are only treated as plain data and not as control information. We discuss how Spring handles this challenge in the following section.

4.2.2.2 View layer and security

Even at the top of the application stack, Spring offers the developers to choose between several popular view technologies to fill its role as "the leading full-stack Java/Java EE framework". It supports template engines like Velocity [22] and FreeMarker [61] in addition to plain JSP files. To lay out the application pages, Jakarta Tiles is supported. However, we will concentrate on the JSP and JSP taglibs and their security aspects.

First of all, we need to point out what taglibs are, why they are preferred when developing web pages using JSP technology. According to [56]

> ... tag libraries define declarative, modular functionality that can be reused by any JSP page. Tag libraries reduce the necessity to embed large amounts of Java code in JSP pages by moving the functionality of the tags into tag implementation classes.

Taglibs allow web site developers to embed JSP functionality without using <% ... %>, <%= ... %> or similar constructs to include Java code in JSP pages. Instead, a taglib provides special tags that look like traditional HTML tags, e.g. <c:out value="Hello World!" /> to print values or <c:out value="${name}" /> to print the contents of the variable *name*. This is only one of the tags found the JSTL. It is possible to develop custom taglibs. Spring comes with a custom taglib simply named Spring. Among the tags in this library we find the *spring:bind* tag, which is used to bind form fields to object member variables. Struts and JSF also provides custom taglibs. More details of these are found in chapters 5 and 6, respectively.

In terms of security and metacharacters, the important part of the taglibs is at the points where data leaves our application code and enters another subsystem. This include tags like <c:out />, <sql:query />, <sql:update /> and possibly others. Since the only subsystem we are passing data to through JSTL in simple Spring applications is web browsers, we will focus on <c:out />. However, we will use a paragraph to comment on security aspects of using <sql:query /> and <sql:update /> as well.

According to the specification of JSTL 1.1 [57],

> ... <c:out /> converts the characters <, >, ', ", & to their corresponding character entity codes (e.g. < is converted to <).

This is the exact process we recognize as HTML-encoding described by several sources, e.g. [31]. We have investigated the JSTL implementation by The Apache Software Foundation [20], and conclude that this implementation follows Sun's specification. Hence it is safe to use the <c:out /> tag in this implementation to avoid cross-site scripting problems. Remark that if other tags are passing data to a web browser, they may be vulnerable to cross-site scripting attacks if they do not use HTML-encoding properly.

Even if it not recommended to access a database directly from a JSP, the JSTL specification [57] provides a couple of tags that make such access possible (e.g. <sql:query />, <sql:update />). The specification arguments that such tags may be used in prototype implementations and some other cases. What is important, is that they both use `PreparedStatements` to execute their operations. As we have mentioned several times already, these objects make it almost impossible to do SQL-injections.

4.2.2.3 Validating form input

To ensure that our application is receiving data within the expected domain and range, input validation is needed. In Spring, this validation is implemented by

means of the `Validator` interface found in the `org.springframework.validation` package. A validator class should be implemented for each of the domain objects that may be created or changed using the web interface.

Suppose that we are developing a simplified blog service where users can create their own blogs. To use this service, users only register their details at web page containing a HTML form. Spring maps these details to a domain object (e.g. `User`) that should be validated before it is accepted. If we assume that this `User` object contains member variables for username, password, first name, last name, and email, the `UserValidator` may look like the validator given in listing 4.3.

Listing 4.3: UserValidator

```java
package validators;

import java.util.List;

import org.apache.oro.text.perl.Perl5Util;
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

import data.User;

/**
 * @author leifode
 *
 * This class is responsible for user validation. The
 * user details are validated both when a new user is
 * created and when user details are updated.
 */
public class UserValidator implements Validator
{
  private static final String EMAIL_REGEXP =
    "\^[a-z0-9]+([_\\.-][a-z0-9]+)*@([a-z0-9]+([\\.-][a-z0
      -9]+)*)+\\.[a-z]{2,}$/i";

  /**
   * Indicates just that only the user class is
   * supported.
   *
   * @see org.springframework.validation.Validator#supports(
      java.lang.Class)
   */
  public boolean supports(Class clazz)
  {
    return clazz.equals(User.class);
  }

  /**
   * The method validates the username, password,
   * first name and last name properties of a User object.
   *
   * @see org.springframework.validation.Validator#validate(
      java.lang.Object, org.springframework.validation.
      Errors)
```

43

```
40      */
41    public void validate (Object obj, Errors errors)
42    {
43      User u = (User) obj;

45      ValidationUtils.rejectIfEmptyOrWhitespace (errors, "
            username", "required.username", "Username is
            required.");
46      ValidationUtils.rejectIfEmptyOrWhitespace (errors, "
            password", "required.password", "Password is
            required.");
47      ValidationUtils.rejectIfEmptyOrWhitespace (errors, "
            firstName", "required.firstName", "First name is
            required.");
48      ValidationUtils.rejectIfEmptyOrWhitespace (errors, "
            lastName", "required.lastName", "Last name is
            required.");
49    }

51    /**
52     * Using a regular expression to validate the format of
            the
53     * email address.
54     */
55    private void validateEmail (String email, Errors errors)
56    {
57      ValidationUtils.rejectIfEmptyOrWhitespace (errors, "email
            ", "required.email", "An email address is required."
            );

59      Perl5Util p5u = new Perl5Util ();
60      if (!p5u.match(EMAIL_REGEXP, email))
61        errors.reject("invalid.email", "Email is invalid.");
62    }
63  }
```

As long as this validator is properly wired in through a XML-configuration file, Spring ensures that these validation methods are called when the form is submitted. If the form is not accepted by the validator, the user is returned to the form view showing appropriate error messages written by the developer.

In our `UserValidator` the username, password, first name and last name fields are rejected if they consist of whitespaces only or are empty. We also could have forced the users to select passwords which are at least five characters long consisting of at least two upper-case letters. This make brute-force password attacks less likely to succeed.

The email validation is somewhat more complex. The `final` String object called EMAIL_REGEXP, defines the format of addresses that are accepted. If no email address is given or it is not matching this regular expression, it is not accepted.

These are the principles of the validation functionality that is included in the Spring framework. If we look back to section 2.1, we see that this approach is a combination of whitelisting and blacklisting. Spring uses blacklisting when the `rejectIfEmptyOrWhitespace()` method is called because this method looks for an empty form element or an element only containing whitespace and rejects it if found. On the other side when defining regular expressions, the approach has

changed to whitelisting. All terms that do not match the expression are rejected.

4.3 ACEGI SECURITY SYSTEM

The Acegi Security System is developed to complement the functionality implemented by the Spring framework. The project's homepage states that:

> "The Acegi Security System for Spring provides authentication and authorization capabilities for Spring-powered projects, with optional integration with popular web containers. The security architecture was designed from the ground up using "The Spring Way" of development, which includes using bean contexts, interceptors and interface-driven programming. As a consequence, the Acegi Security System for Spring is useful out-of-the-box for those seeking to secure their Spring-based applications, and can be easily adapted to complex customized requirements."

In other words, Acegi uses AOP, IoC, and JavaBeans to provide security services for Spring applications. In this chapter we will discuss the principles used by these services. We will focus our discussion on important topics like security interceptors, authentication, access control, servlet filtering and how to secure method invocations. Please remark that we will use several sources, [5, 42, 43, 69, 73], throughout this section.

4.3.1 Security Interceptors

To enforce security, Acegi uses security interceptors. Each interceptor is delegated responsibility for a limited set of tasks. Because of the architecture of Acegi, it is quite easy to plug it into web applications. The developer only needs to define a servlet filter in the `web.xml` of the web application that filters all the incoming requests through a class called `FilterToBeanProxy` in the Acegi API. This class delegates all the request processing to Spring beans including the security processing.

The security processing itself is split in several separate filters. In a web application using role-based authorization, the `FilterToBeanProxy` bean ensures that the following filters are applied in order:

1. **HttpSessionContextIntegrationFilter**
   This filter simply holds the HTTP session object between requests if one exists. The HTTP session object is queried for a `SecurityContext` object at the start of request and posted back at the end of the request processing if it is changed.

2. **AuthenticationProcessingFilter**
   This filter is responsible for the authentication. If the `SecurityContext` object looked up by the **HttpSessionContext**IntegrationFilter contains an *Authentication* object, the user is already authenticated and the request is forwarded to the **FilterSecurityInterceptor**. If the user is not previously authenticated, this filter throws an *AuthenticationException*.

3. **ExceptionTranslationFilter**
   This filter takes care of the exceptions that is thrown by the other filters. If an *AuthenticationException* is caught, this filter launches the authenticationEntryPoint. Otherwise if an *AccessDeniedException* has occurred, the filter determines whether the user is an anonymous user or not. If he/she is, the

authenticationEntryPoint is launched. If not, a HTTP FORBIDDEN response is sent.

4. **FilterSecurityInterceptor**
   When the user has successfully authenticated, the rest of the job is taken care of by this filter. Even though, this **FilterSecurityInterceptor** delegate a lot of its tasks to other objects (e.g. an AccessDecisionManager is used to authorize the request.).

Another filter worth mentioning that may be applied, is the **ChannelProcessingFilter**. This filter contains a *ChannelDecisionManager* which supports one or more *ChannelProcessors*. If we define a *SecureChannelProcessor* here and then, in the **ChannelProcessingFilter** definition, state that a secure channel is required to access some resources, a Secure Sockets Layer (SSL) must be set up before we are allowed to access them. By using this filter, we may ensure that no password is sent in plain text through the network.

These are the most important principles when considering request filtering. All the filters except the **ChannelProcessingFilter** are illustrated in figure 4.1.
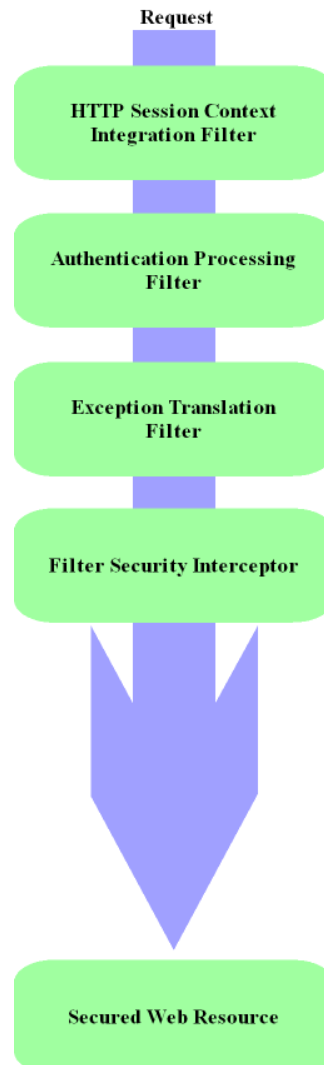
Figure 4.1: Acegi Security Interception Filters

4.3.2 Authentication

As stated in 4.3.1, the **AuthenticationProcessingFilter** is responsible for the authentication. Roughly, this means to check if user credentials are correct and if they are, place a *Authentication* object within the *SecurityContext*. But where are the correct credentials stored?

Acegi provides clear interfaces for credentials storage. It is possible to implement a custom authentication data source by implementing these interfaces. Additionally, Acegi provides four complete implementations that fit most user's needs:

- In-memory user DAO authentication (e.g. user details stored in an XML-file)

- JDBC DAO authentication (e.g. user details stored in a database)

- LDAP authentication (e.g. user details stored in a LDAP repository)

- Single sign-on using Yale Central Authentication Service (CAS) (e.g. user details stored in a CAS)

4.3.3 Access control

When the authentication is taken care of and each user is accepted or rejected, it is time decide whether the current user should be allowed to access the requested resource or not. This is what we call access control or authorization. The Acegi Security System introduces two different authorization principles:

- Role-based authorization

- ACLs

When using role-based authorization, all the users are divided into groups which are associated roles. Administrators may be associated with *ROLE_ADMIN*, while moderators are associated with the *ROLE_MODERATOR* role and regular users get the *ROLE_USER* role. This may be sufficient in some situations, but we run into trouble if each user should be allowed to edit and change their user details. There is no way to distinguish between users, so a logged-in user is allowed to change the details of every other user, theoretically. One solution to this challenge is called Access Control List.

An ACL is a list that:

> for each object, (. . .) lists users and their permitted access rights. The ACL may contain a default, or public entry.

This means we, in Acegi are able to specify who (*Authentication*), where (*Method-Invocation*) and what (e.g. a domain object). Acegi provides a simple set of classes and interfaces that is almost usable out-of-the-box. Additionally, Acegi implements a class named `JdbcDaoImpl`. This class is used when ACL information from a database. More details on how ACL is implemented in Acegi and how to use it, is found in [5].

4.4 SUMMARY

In this chapter we have discussed some important security properties of a couple of important Spring modules including error handling, and how two of the seven Spring modules influence security. Even though, the Spring framework is not addressing important web application security functionality like authentication and access control which is often needed in more advanced application. This is the exact point where the **Acegi Security System** complements Spring. It provides the missing services and is built on the same principles as Spring itself.

CHAPTER 5

STRUTS SECURITY

In this chapter, we will elaborate on Struts functionality and how it influences security. The first section discusses error handling and the following ones focus on important topics like security in MVC components, input validation, and access control.

5.1 ERROR HANDLING

As we saw in Spring, error handling in Struts also means proper exception handling. Based on the same reasons that are mentioned in section 4.1, it is very important that web application frameworks like Struts provide proper functionality to take care of these exceptions securely.

Since Struts 1.1, the tag called <global-exceptions /> has been part of the Struts config file. A sample configuration is given in listing 5.1.

Listing 5.1: Catching global exceptions in a secure manner

```
<struts-config>
  <global-exceptions>
    <exception   type="java.lang.NumberFormatException"
          path="/WEB-INF/jsp/errors/numberformat.jsp"/>
  </global-exceptions>
</struts-config>
```

This setup ensures that all `NumberFormatExceptions` thrown by the Struts application are caught and the user is forwarded to a JSP that describes the error in a user-friendly manner. All exceptions should be treated this way, probably also objects of the `Exception` base class.

5.2 STRUTS MODEL

Since Struts does not provide any classes or interfaces related to the model, the developers are responsible for the security in this module. Anyway, like we mentioned in section 3.4, there are several frameworks available that may ease this task. The important point is that all the metacharacters of the model subsystems are identified and properly escaped. For instance, if we use plain JDBC to pass queries to a database, we have to make sure that '-characters are somehow taken care of.

5.3 STRUTS VIEW

As discussed in section 3.4.2, the view component of Struts web applications mainly consists of JSPs using taglibs. The main task of these tag libraries is to render

49

HTML pages and nothing else. One of the largest security risks related to this page rendering (notice that this is the exact point where the data is leaving our subsystem and entering the browser), is cross-site scripting [23]. Looking back to the solutions sketched in section 2.4, all the tags found in these taglibs that are outputting data, are required to escape all HTML metacharacters.

The most common tag found in Struts's taglibs that is outputting data, is <bean:write />. This tag is capable of printing beans, bean properties, HTTP headers, request parameters and because the data found in these variables if often dictated by the user, all HTML metacharacters should be properly HTML-encoded. After some source code research, we conclude that the Struts bean taglib handles these metacharacters pretty well.

5.4   STRUTS CONTROLLER

When we discussed the Struts controller in section 3.4.3, we stated that it consists of five different modules. In terms of security, we think that the `Actions` are most important because their `execute()` method is dealing directly with the submitted request parameter without any validation. There are at least two methods that may be extended to ensure proper request parameter validation:

1. We can do the validation in the `execute()` method of each `Action` using request parameters.

2. We can overload the `validate()` method of the `ActionForm` and use it validate request parameters.

We prefer the last alternative because it separates that validation from the rest of the business code in `execute()`.

5.5   INPUT VALIDATION

Looking back to section 2.1.2, input validation is about ensuring that our web application is getting the expected input within the expected domain and range. Struts provides input validation through the module called Struts Validator. It includes functionality both for client- and server-side validation, but in this project we concentrate on the server-side since validations at the client side are inherently insecure.

In Struts, the form beans are responsible for the validation. To enable validation, the developers only have to let their form beans extend `ValidatorForm` or `ValidatorActionForm` instead of the plain `ActionForm` class. Then, when the `ActionServlet` calls `validate()` on this form, it looks for a related entry in a file called `validation.xml`. If we continue to use the example given in section 3.4.3, a validator for the form bean named "loginForm" may be configured like we show in listing 5.2.

Listing 5.2: validation.xml for loginForm

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <!DOCTYPE form-validation PUBLIC
4     "-//Apache Software Foundation//DTD Commons Validator
          Rules Configuration 1.0//EN"
5     "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd"
          >
```

```
6
7  <form−validation>
8    <formset>
9       <form name="loginForm">
10          <field property="username" depends="required" >
11            <arg0 key="label.username" />
12          </field>
13          <field property="password" depends="required">
14            <arg0 key="label.password" />
15          </field>
16       </form>
17    </formset>
18  </form−validation>
```

This is a simple validator only stating that the user is required to fill in something in the two textboxes. This is only one of the validators that are provided by Struts Validator module. Some of the others are:

mask
Allows the developer to specify a regular expression that the input should match. If it does not, the validation fails.

range
The range validator is used to check if the given value is within the expected range. If we specify that the minimum value is 1 and the maximum is 4, then the validation succeeds only if 1, 2, 3, or 4 is given by the user.

minLength
minLength is used to specify the minimum number of letters that an inputted string should consist of (e.g. a password must be at least five characters long).

maxLength
maxLength is the opposite of minLength. It specifies the maximum number of letters that are accepted in a string.

date
The date validator checks if the given date corresponds to a legal date format.

In addition to the validators provided by Struts, the developers are encouraged to develop more advanced validators to fill their requirements. In an example given by [32], a validator that checks if two HTML text fields are equal is developed.

We recognize that Struts Validator uses the whitelisting approach we discussed in section 2.1.4. Each validator lists the values, ranges, formats etc. that should be accepted, instead of trying to identify all deniable input. This design decision is one of the Struts Validator advantages compared to e.g. custom solutions.

5.6 ACCESS CONTROL

Since Struts is an extensible framework, it provides extension points for different access control implementations. However, as [28] mentions, there are mainly two ways to secure Struts applications:

- Using container-managed security

- Using application-managed security

Container-managed security means that we are using the functionality found in the Servlet container to enforce proper access control, while application-managed security is related to the functionality implemented by Struts. Since we are focusing on Struts, we will concentrate on access control in terms of application-managed security in the rest of this section.

The access control mechanisms found in Struts is mainly based on roles, also known as role-based access control. Each user is assigned a role or a set of roles where each role is associated with a specific set of privileges. If a user tries to access a page which requires a role that the user is not connected to, access is denied.

When it comes to implementation, the access control may be part of several layers depending on the application's requirements. [28] suggests three different solutions:

- Implement the access control in terms of `execute()` in `Actions` and bean property checks in the JSPs.

- Implement the access control by extending the `RequestProcessor` class found in Struts API.

- Implement the access control by using Servlet filtering.

5.6.1 Access control in Actions and JSPs

Implementing access control in terms of `Actions` and JSPs is a simple way to achieve proper access control in Struts applications. The idea is to save a User object in the session during the login procedure, then, when a page with restricted access is requested, the JSP checks if there is a User object in the session having the required roles. If these checks complete successfully, access is granted. Otherwise an error message or page with limited functionality is returned to the user.

Another variant of this solution is to do all the authorization checks in the `Action` class. This approach scales better than the previous solution and it is easier to get things right.

One of the main drawbacks using this approach, is the that direct access to the JSPs are allowed as long as they are not placed in folder within the WEB-INF directory of the web application.

5.6.2 Extending the RequestProcessor

Every request received by the `ActionServlet` are handed off to the `RequestProcessor`. This class includes a method named `processRoles(...)` which may be overridden in a subclass to implement a proper access control. [28, 32] develops an example using this approach. Additionally, the `RequestProcessor` contains an even more generic method called `processPreprocess(...)` that may be overridden if the role-based access control is insufficient.

This approach still suffers from the same drawback as mention in the previous section, there is no way to deny direct access to the JSPs.

5.6.3 Access control through Servlet filtering

Servlet filters were introduced in the Servlet 2.3 specification. A Servlet filter can provide a quite powerful access control if it is combined with Struts Actions and

JSPs because the filter may be applied to an URL or set of URLs. Notice that Servlet filtering is the main principle used in the Acegi Security System discussed in section 4.3.

A Servlet filter is a Java class that is executed before the request is handled and possibly, after the response is sent back to the client. This allows the web application to do all authentication and authorization before the application code is executed. This organization leads to a loose coupling between important security concerns and the application logic and it also allows the developers to reuse large parts of the authentication- and authorization logic in several projects.

5.7 SUMMARY

In this chapter we have discussed how Struts implements error handling to avoid giving away too much information to attackers (e.g. stack traces), how each of the MVC components is implemented to avoid metacharacter problems and other vulnerabilities, and how the Struts Validator works. The final section discusses three different ways to implement access control in Struts applications.

CHAPTER 6

JAVASERVER FACES SECURITY

JavaServer Faces security is quite a sad story. From the very beginning, JSF has been developed to suit several needs and security was not one of them. This is reflected in a lot of written material, among them [45], which has devoted two out of over 600 pages (!) to security. In this chapter we will discuss the functionality JSF actually implements to improve security.

6.1 ERROR HANDLING

JSF provides only a limited set of error handling functionality. If we look back to the event listener signatures we sketched in section 3.5.2, we see that these methods do not throw exceptions. This means that every listener is required to catch the exceptions that are thrown and handle them properly. JSF uses this approach mainly because it allows each web application to convert every thrown exception into appropriate error messages that are shown to the user when the HTML form is redisplayed. By [45] this should lead to a more user friendly application because it is unnecessary to forward the user to a more generic error page.

From a security point of view, the JSF approach alone is not enough. The Java compiler ensures that there are no unhandled exceptions as long as they are not extending `RuntimeExceptions`. If such an exception is thrown, additional measures are required. However, since JSF is built on top of the Servlet API and the applications are run in web containers, it is possible to use error handling mechanisms provided by this API. If all the related exceptions are identified and specified in `web.xml`, the user is forwarded to a generic error page if these errors occur.

6.2 METACHARACTER HANDLING

As we discussed thoroughly in chapter 2, JSF has to ensure that every metacharacter that is passed to subsystems, is properly escaped. Since JSF only cares about the web layer, only the metacharacters found in this layer should be taken care of by the framework. The responsibility for metacharacters found in other subsystems (e.g. DBMS) is handed over to the developer or to another framework if an Object Relational Mapping (ORM) solution is used.

Web layer metacharacter handling is quite a challenge when using JSF because of the extensive use of components. As we stated in section 3.5.1, each component is associated with at least one `Renderer`. Each `Renderer` is responsible for converting an JSF component into an appropriate format. The default implementation of JSF comes with a HTML 4.01 `RenderKit` (a collection of HTML `Renderers`). Each of the `Renderers` in the `RenderKit` is responsible for proper HTML encoding of user strings to avoid cross-site scripting vulnerabilities. We have done some research on

MyFaces 1.1.2 and it seems to us that the HTML 4.01 `RenderKit` ensures proper HTML encoding.

JSF is developed to be independent of view technology, but the HTML `RenderKit` is the only default implementation provided the framework. [45] encourage developers to write custom `RenderKit` implementations for SVG, XML, etc. However, it is very important that the `Renderers` identify all metacharacters found in these technologies to avoid security vulnerabilities in custom `RenderKits`.

6.3 INPUT VALIDATION

Input validation is one of the areas where JSF do quite a good job if we disregard client side validation. JSF does not support JavaScript validation out of the box, but it is possible to develop custom validators that support client side validation [45]. It is also provided in the JSF-Comp project (section 6.6.2).

The JSF server side validation may be split into two groups, each containing several components:

- **Standard validators**

    - Required
    - Double range validator
    - Length validator
    - Long range validator

- **Converters**

    - Boolean converter
    - Byte converter
    - Character converter
    - Date Time converter
    - Double converter
    - Float converter
    - Integer converter
    - Long converter
    - Number converter
    - Short converter

The standard validators above is called in the JSF life-cycle phase called process validations, discussed in section 3.5.5.3. If the required attribute is set on an input component, the required validator ensures that no empty value is accepted. The other three validators bundled with JSF are quite self-explanatory:

- **Double range validator**
  This validator ensures that the user input may be converted to a double value and that it is respecting the minimum and the maximum values specified by the application.

- **Length validator**
  The length validator ensures that given input is at least as long as the minimum attribute specifies, but not longer than the value specified by the maximum attribute.

- **Long range validator**
  The long range validator is similar to the double range validator. It ensures that the given value may be converted to a long value and that it respects the specified range.

If validation errors occur, these errors are reported and the JSF execution jumps to the render response phase.

If the validation succeeds, the rest of the validation is carried out in the update model values phase (section 3.5.5.4) by converters. JSF calls the correct converter based on the data type of the variable that input is assigned to. JSF provides ten different converters and each developer may create custom ones if needed. If any conversion fails, the JSF continues in the render response phase just like we saw when the validation failed.

6.4 AUTHENTICATION

JSF provides not no direct support for authentication services. However, it is possible to use the authentication and authorization scheme supported by the Servlet API through servlet filtering. This approach is similar to the one used by Acegi Security System, discussed in section 4.3. Since this framework does not depend on Spring, it is possible to use it together with JSF as well. We discuss a custom Acegi Security System for JSF in section 6.6.2.

6.5 ACCESS CONTROL

JSF does not implement any access control. If it is needed, which it often is, the developer has to use the access control mechanisms provided by the Servlet API and the web containers, or design a custom solution. The access control provided by the containers, is illustrated in listing 6.1 [44]. Notice that we referred to this access control model as container-managed security in 5.6.

Listing 6.1: Web container access control

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
    Application 2.3//EN'
   'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
...
  <!-- Define a security constraint on this application -->
  <security-constraint>
    <web-resource-collection>
        <web-resource-name>Admin pages</web-resource-name>
        <url-pattern>/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <!-- This role is not in the default user directory
            -->
        <role-name>admin</role-name>
    </auth-constraint>
  </security-constraint>

```

```
19    <!-- Security  roles  referenced  by  this  web  application  -->
20    <security-role>
21        <description>
22            The  role  that  is  required  to  log  in  to  the  Manager
                  Application
23        </description>
24        <role-name>admin</role-name>
25    </security-role>
26    ...
27  </web-app>
```

This example adds access control to the `admin` directory of the web application. To get access to files within this directory, a user has to be logged in and associated with the admin role. In our example, the admin role is the only registered role.

6.6 JSF SECURITY INITIATIVES

Several people has recognized the security disadvantages and a few initiatives have been started to develop solutions that integrate with JSF, providing proper authentication and authorization. In this section we will discuss a couple of these projects, JSF-Security Components [63] and JSF-Comp [62].

6.6.1 JSF-Security

By default, JSF uses four scopes where variables may be saved. These are:

Application scope
Variables saved in this scope are available for the entire life of the application.

Session scope
Variables saved in this scope are available as long as the user's session is valid.

Request scope
Variables saved in this scope are only available during the current request.

Page scope
Variables saved in this scope are only available on the current page.

To implement its functionality, JSF-Security provides a new scope called `securityScope` where all the related variables are stored. When they are placed in this scope, they are available as long as the session is valid, similar to the session scope. The variables implemented by the JSF-Security project provides important functionality connected role-based access control. The variable expressions provided by this project are :

Table 6.1: JSF-Security variable expressions [63]

Variable expression	Action
#{securityScope.securityEnabled}	Returns false if the user is not authenticated or if no security is installed.
#{securityScope.remoteUser}	If an is user is authenticated, the associated username is returned.
#{securityScope.authType}	Returns a string indicated what type of authentication is being used.

Continued on next page

58

Variable expression	Action
#{securityScope.userInRole['roleList']}	Returns true if the user is assigned to at least one of the roles listed in the comma separated list named 'roleList'.
#{securityScope.userInAllRoles['roleList']}	Returns true if the user is assigned to all the roles listed in 'roleList'.

These expressions may be plugged in the JSPs like we show in listing 6.2.

Listing 6.2: Web container access control

```
1  ...
2    <!-- Only visible if security is not enabled --!>
3    <h:panelGroup rendered="#{!securityScope.securityEnabled}"
         >
4      <h:outputText value="Security is not enabled. Access
           denied." />
5    </h:panelGroup>
6
7    <!-- Only visible if security is enabled --!>
8    <h:panelGroup rendered="#{securityScope.securityEnabled}">
9
10     <!-- This link is only visible if the user is assigned
           the 'user' role. !-->
11     <h:commandLink ... rendered="#{securityScope.userInRole
           ['user']}">
12       <h:outputText value="Edit user details"/>
13     </h:commandLink>
14
15     <!-- This link is only visible if the user is assigned
           the 'admin' role. !-->
16     <h:commandLink ... rendered="#{securityScope.userInRole
           ['admin']}">
17       <h:outputText value="Access admin pages"/>
18     </h:commandLink>
19   </h:panelGroup>
20 ...
```

This example shows how the JSF-Security expressions are used to render only the components that are accessible for the user that is logged in. If security is not enabled, no links are shown. Otherwise if there is a logged in user, one or two links are shown depending on the roles that are assigned to the logged in user.

6.6.2 JSF-Comp

The JSF-Comp project provides several JSF components that extend the default functionality implemented by the framework [62]. These components are:

Chart Creator
    Allows the developer to create JFreeCharts on JSF pages.

Client Validators
    These components add client validation support to JSF.

Acegi-JSF components
> The components found in this package implement a JSF version of the tags found in Acegi Security System, discussed in section 4.3.

Excel Creator
> Excel creator allows JSF to export `dataTables` to Excel with one click.

OnLoad
> Allows JSF actions to be run on page load instead of form posting.

In the following we will concentrate on the Acegi-JSF components, since these are the only components relevant in this security discussion.

The goal of Acegi-JSF components is develop components that integrate JSF with the Acegi Security System with JSF. Through their components, this project provides JSF with functionality similar to the role-based access control model we found in JSF-Security (see section 6.6.1).

One Acegi-JSF component is exemplified in listing 6.3.

Listing 6.3: Authorization component provided by Acegi-JSF

```
1  ...
2  <authz:authorize ifAllGranted="ROLE_SUPERVISOR">
3      Components that are only visible to the users that
           satisfy the requirements here...
4  </authz:authorize>
5  ...
```

This example illustrates how the role-based access control may be enforced in a JSF setting. The `ifAllGranted` attribute may specify a list of roles. If the current user is not assigned to all of these roles, the child components are not rendered.

6.7 SUMMARY

In this chapter we have discussed to what degree security is enforced in JSF web applications. We have pointed out why exception handling in listeners along, is not enough and that metacharacters are handled in the HTML 4.01 `RenderKit`. Next, we have touched validators provided by JSF, authentication, authorization and a couple of projects that extend the default authentication and authorization functionality provided by JSF.

<div align="right">

7
CHAPTER

STRUTS ACL EXTENSION

</div>

In the previous four chapters, we have elaborated on important architectural principles used in three popular Java web application frameworks and how these influence on web application security. In this chapter we will discuss an Internet banking web application to illustrate some of the security shortcomings of these frameworks. In the last section we will sketch how ACLs may be implemented in and integrated with Struts.

7.1 CASE: INTERNET BANKING APPLICATION

In the last ten years, Internet banking applications have become very popular. They allow us to access our bank accounts, transfer money, get updates on our investments, etc. from all over the world. These services only require a browser (e.g. a HTML browser or Wireless Markup Language (WML) browser) and an Internet connection. Even if this has increased the flexibility of banking services, it has also challenged the web application security. No bank want to offer insecure services that allow crackers to get in control of their customer's accounts.

To provide a secure Internet banking service, a lot of security threats at different levels have to be considered. First of all there is a need for strong authentication. The bank has to make sure that each customer may identify themselves uniquely to the services and it should not be possible for users to act on behalf of others. Next, there is a need for a flexible access control mechanism. Each customer is only allowed to transfer money from a limited set of accounts, while the operators in the bank itself should be allowed to access all the accounts in order to execute transactions and get paid for their services.

Companies that use these banking services, often have some additional requirements. Usually, a group of users is granted access to their banking accounts. It seams quite clear that a role-based access control model would be insufficient in these applications. Access to business objects like accounts, should be granted on a per-user basis and not based on their roles.

Several other threats related to Internet banking services are connected to the data transfer itself. If a cracker is listening to the communication link, he or she should not be able use that transferred information to defeat the security of the application. SSL solves some of these problems by introducing cryptography, but web application developers still need to keep attention to replay attacks and other groups of attacks.

In the rest of this chapter, we will concentrate on the access control part of an Internet banking service. We will assume that the access control implemented in the web application, is the only access control used by the application, and that the data is stored in a file or a database. This is a very unrealistic simplification if we think of the large production systems needed to serve thousands of concurrent users, but the access control principles are more or less the same.

7.2 FRAMEWORK EXTENSION REQUIREMENTS

Every developer designing a framework needs proper knowledge of patterns and best practices to ensure that the framework is usable in a wide range of applications. Additionally, he or she needs to have a very good understanding of the technologies the framework uses and is build on top of. In Java, this often means the Servlet API, the reflection API, HTTP, etc.

To design a framework extension the already mentioned requirements are still important, but there are several other general requirements as well:

Framework integration

To convince a developer to use a framework, the framework has to provide some advantages that he or she finds useful. Well integrated modules or increased overall security are two such advantages. This means that a new standalone security framework (e.g. JGuard [40]) is not a good solution for our task. In a well designed framework, it should be possible to configure all the modules using a single interface (e.g. XML file).

Dependencies

Developing a framework that depends on a lot of extern and evolving APIs is not a good approach either. If one of the APIs change, the framework code itself, also has to be rethought. A better approach is to based the framework on stable APIs and program to interfaces. This leads to loose couplings, and it should be a fair task to make the framework work with future versions of external dependencies.

Clearly defined interfaces

We have already mentioned how important stable and clear interfaces are, when choosing framework dependencies. However, we also have to define a clear interface for the functionality provided by our framework. No developer is going to use our framework if he/she has to change their applications every time a framework bug is fixed.

To implement ACLs in a framework, there are some additional requirements:

Deny all access by default

Denying all access by default is a very important principle in software security. It should also be supported in a framework setting because it ensures that no default settings compromise the security of the user web application.

Centralize access control

Centralizing access control decisions is another important principle that eases the job the developers/maintainers. If he or she suspects that there are errors in the access control, there should be possible to resolve it by inspecting a predefined set of files, database tables, etc.

Flexibility

An ACL framework implementation should be quite customizable in order to fit the need of quite different applications. It should support access control at several levels from object access down to object field access.

7.3 WHY STRUTS?

Before we move on to the high level architecture of the ACL implementation/ integration code, we will point out why we chose to work with Struts instead of Spring or JSF.

Spring itself, does not support ACLs at all. However, the Acegi Security System implements both ACLs and a role-based access control model. Since this project uses IoC as well, it should be a fairly easy task to implement a web application based on Spring using ACLs. Hence, we do not think there is a need for another Spring ACL implementation.

On the other side, we have JSF. Even if it has been under development since 2001, it is still changing due to developer requests and the JSF 1.2 specification is still not considered stable. The JSR process is making slow progress, and we think that there is more development needed before JSF is ready to use in enterprise applications. A similar solution to the one we sketch in 7.4 may be possible to use with JSF too, but as long as the APIs are unstable, we think there is no need for advanced and complicated access control mechanisms either.

Based on these considerations, we chose to extend Struts because it has quite a stable API, is widely used and tested and the default implementation does not provide ACL support.

7.4 ASPECT-ORIENTED ACCESS CONTROL IN STRUTS

To remove the access control limitations found in Struts, we suggest implementing ACLs using AOP. An overall architecture diagram is given in figure 7.1.
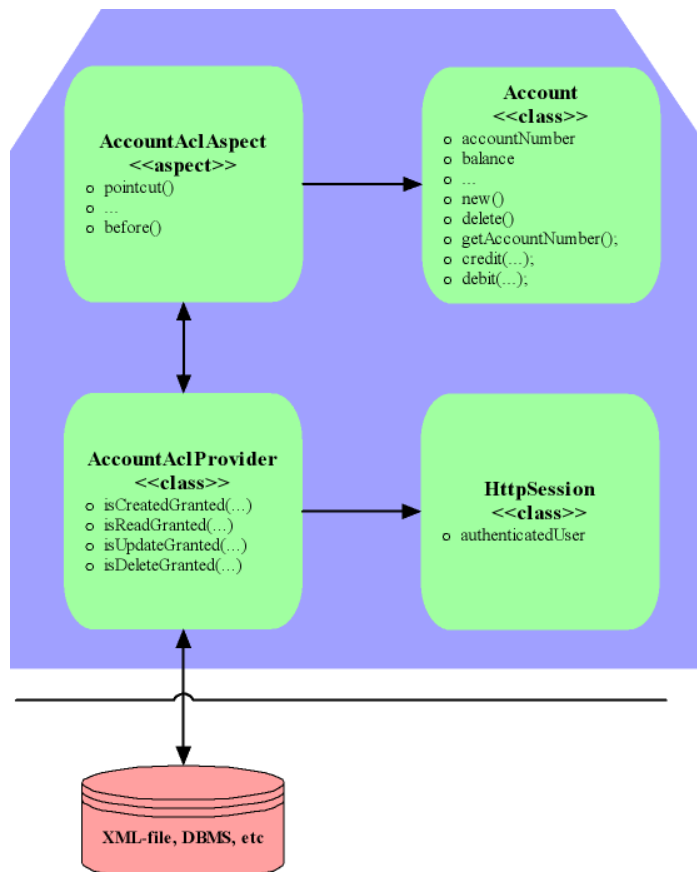

Figure 7.1: ACL architecture overview

In this figure, `Account` represents a domain object, `AccountAclAspect` represents an aspect that is intercepting all the method calls to `Accounts` to enforce

proper access control. The `AccountAclAspect` delegates most of its work to the `AccountAclProvider`. This provider accesses an XML-file, a DBMS or similar to check whether the access should be granted or not.

7.4.1 Framework abstractions

In order to make the ACL functionality more usable and generic, a framework implementation of this access control should provide some well-documented interfaces and some classes that implements them. We suggest that the framework should include a(n):

AclAspect
> The `AclAspect` should provide generic aspect functionality that is needed to intercept all the method calls to the associated domain object. This default aspect should then ask an `AclProvider` whether this access should be allowed or not. If the access is granted, the requested method is invoked, otherwise an `NotGrantedException` should be thrown. This will force the application to take action if access is denied.

AclProvider
> An `AclProvider` is an abstract class providing the interface that every `AclProvider` subclass should support. As we indicated in figure 7.1, the `AclProvider` should at least provide methods for create, read, update, and delete operations.

> > **XmlAclProvider** The `XmlAclProvider` class extends the abstract `AclProvider` class to support ACLs saved in an XML-file.

> > **JdbcAclProvider** The `JdbcAclProvider` class is similar to the `XmlAclProvider`, but instead of supporting ACLs in XML-files, it supports ACLs saved in a DBMS.

To connect the aspect and the domain objects, we suggest that Struts configuration Document Type Definition (DTD) should extended to support tags we illustrate in listing 7.1

Listing 7.1: Extended Struts configuration

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <struts-config>
4  ...
5    <domain-classes>
6      <domain-class>
7        <class-name>beans.Account</class-name>
8        <acl-aspect>
9          <aspect-name>aspects.AccountAclAspect</aspect-name>
10         <!-- All method calls are intercepted by default,
               but it may overridden by the intercepts tag. --!
               >
11         <intercepts>
12           <list>
13             <item>get*</item>
14             <item>set*</item>
15           </list>
16         </intercepts>
```

```
17        </acl−aspect>
18      </domain−class>
19    </domain−classes>
20  ...
21  </struts−config>
```

In this listing we see how an aspect is associated with the `Account` class. It also shows how to let the aspect only intercept method names starting with get and set.

7.4.2 ACL management

One of the ACL challenges, is proper management. If a single entry is wrong in the ACL, it may compromise parts of the system security. Let us assume that we use an access control model in our Internet banking application that combines roles and ACLs. Each account owner has debit and credit rights for his/her accounts. Additionally, the bank staff may be assigned a role called "operator" which allows them to debit and credit a set of accounts. A single error in an ACL e.g. "User A is allowed to debit and credit user B's account", may lead to both bad reputation and economical losses.

In our design, we would like to automate as much as possible of the ACL management process. Every time a new account (or other domain object in the general case) is created, the creator is assigned proper rights which usually include `read` and `update (write)` for all the domain object properties. In our banking example, this means that we have to do some customization because we will not allow the user to change account metadata like account number, account owner, etc.

In a business setting, there are situations where several users should be able to access a single account. If it possible to keep this functionality out the web application and include it in a local application instead, it should because it reduces the risks related to the ACL management. Otherwise, it is possible to associate a role "acl-admin" to every user that is allowed to change ACLs.

7.4.3 Struts ACL advantages

The solution we have discussed in this section, have several advantages over custom access control implementations:

Modularity
Access control based on AOP has several advantages compared to traditional access control that is coded into every method that accesses restricted properties. If we use AOP, all this code may be place in single aspect that is weaved into the application. A default aspect may be configured to intercept all method calls for a class of objects, such that access to these objects is restricted by default.

ACLs are stored in a limited set of tables, files, etc.
When ACLs are stored in a limited a set of files, there is somewhat easier to main them. It is also possible to develop simple ACL maintaining tools that could reduce the risk of introducing configuration errors.

Security by default
Our extension provides security by default as long as at least the default aspect is connected to each of classes in the domain model. This aspect denies all access to object properties of the associated class by default.

Framework support instead of custom solutions
> By integrating ACLs into a framework, there is often a lot more users that use, review, test and debug this functionality than a custom solution developed by the company itself. This often leads to a better architecture and more scalable code and the developers may focus on the task at hand instead of making creative solutions to framework limitations.

7.4.4   Struts ACL disadvantages

Even if we think our design looks quite promising, we have identified several factors that should be taken into consideration before a full scale implementation is planned:

Not using the Java Authentication and Authorization Service (JAAS) API
> This is one of the main disadvantages of our approach: even if the Java Authentication and Authorization Service (JAAS) API is shipped with the Java Development Kit (JDK), we do not consider it in our Struts extension. A Struts ACL implementation should probably use the JAAS API instead of our custom classes simply because it is more flexible and more thoroughly tested.

Testing and Debugging
> Unit tests are usually more difficult write when introducing AOP. AOP may connect objects such that it is impossible to test one object without testing three other simultaneously.

> AOP, in general, may increase the time needed to resolve bugs because method calls are intercepted behind the developer scenes, making it more difficult to follow the application execution.

Does not protect URLs
> We have chosen to place our access control model at the domain object level which means the higher-level elements like URLs are not protected by this model. Our ACL only ensures that the authenticated user will not be able to access or edit restricted values without proper permissions.

> To protect URLs, the default implementation of Struts provides a role-based access control model. It is fairly straightforward task to combine this model with our ACL implementation.

Introduces AOP in a non-AOP framework
> When we use AOP for ACLs in Struts, we introduce a new way of implementing Struts functionality that is quite different from the rest of Struts. This may lead to developer confusion because the architecture is not consistent within the framework. To resolve this problem, it is possible to redesign parts of Struts and introduce other aspects where suitable, e.g. logging.

7.5   SUMMARY

In this chapter we have discussed how ACLs may improve the access control security in Struts web applications. We use an Internet banking application for motivation and provide a general architecture diagram showing and describe how different components in the design interact. In the last two sections we discuss some advantages and disadvantages related to our approach, and suggest some alternatives.

CHAPTER 8

CONCLUSION

The services provided by web applications, has increase a lot in number and complexity the last 5 years. Business critical applications like banking services and shopping portals let us pay our bills and buy food without leaving the computer screen. To get customers, these services have to focus on usability, but also security. Sadly, if both are not feasible, usability is often preferred.

In this project we have focused on security pitfalls found in these web applications and what Java/Java EE frameworks do to avoid them. We have analyzed three widely used frameworks – Spring, Struts, and JavaServer Faces – and suggest a solution to how the overall security may be improved in applications based on Struts and, possibly, JSF.

We have found out that using a web application framework makes sense in terms of security. All three frameworks that we have researched, do implement HTML-encoding such that a lot of potential cross-site scripting problems are avoided. Because Spring tries to fill the role of full-stack application framework, it also ensures that a lot SQL-injections are avoided. However, as we have discussed in section 4.2.1.2, the implementation is not bulletproof so the developer still needs to focus metacharacter handling.

Struts and JSF only concentrate on the web layer leaving all the metacharacter handling, except HTML-encoding, to the developer. Anyway, there are tools available (e.g. Hibernate) that may ease the implementation of the DAO layer.

One major security disadvantage we have identified in Struts and JSF, is the lack of a flexible access control model. They both support a quite limited role-based access control, but fail to provide standardized functionality that is needed in larger web applications where domain objects are tied to users rather than roles.

To provide better support for user-based access control when using web application frameworks, we have designed a Struts extension that supports ACLs at the domain object level. It is based on Aspect-Oriented Programming and is configured in `struts-config.xml`, just like the other parts of an Struts application.

In the last part, we discuss several alternatives that may be worth thinking through before a full-scale design and implementation takes place. One other important decision is whether an implementation should be using the Java Authentication and Authorization Service (JAAS) API provided by Sun Microsystems or not. We think that both these approaches should be evaluated before the final decision is made.

CHAPTER 9

FUTURE WORK

Despite our work in this thesis, there is a lot of loose ends that still need to be taken care of. In this chapter, we point out some of the topics that future work in the web application framework security area should focus on.

9.1 PROTOTYPE(S)

Before designing and implementing a full-scale Struts ACL extension, there is a need for evaluation of design flexibility and how suitable our design is in different web applications. By developing working prototypes, it is also possible to evaluate different designs. We suggest that at least two prototypes are developed:

- One prototype using our design

- One prototype based on JAAS API

These prototypes should focus on different security aspects, of course, but also developer usability.

9.2 JSF CUSTOMIZATION/GENERALIZATION

When Struts ACL extension is developed, the next step could be developing similar functionality that integrates with JSF. JSF is also failing to provide more advanced access control mechanisms like ACLs. AOP may be a good approach here as well because it may plug ACL functionality into existing JSF components, without changing the components themselves. However, a JSF ACL extension requires some more research and thinking before it is designed (we refer to the requirements we stated in section 7.2).

9.3 INVESTIGATE MORE PITFALLS

Even if we have concentrated on several threats mentioned in [23], there are still large groups of flaws remaining. Some of them are:

- application denial of service

- insecure configuration management

- insecure storage

- buffer overflows

Doing research on how web application frameworks can be developed to counter, or at least, reduce the likelihood of denial of service attacks or some of the other threats, seems quite interesting.

9.4 RESEARCH MORE FRAMEWORKS

Adding more frameworks, is an another extension to this project. As we stated in the introduction, [38] lists tens of other Java web application frameworks whose security functionality may be researched.

Additionally, there are other popular web programming languages as well. PHP, .NET, and Ruby are popular competitors to Java and there are frameworks available for each of them. While PHP and .NET are widely known, Ruby on Rails is one of the newest guys in the web application town. It is:

> ... an open-source framework that's optimized for programmer happiness and sustainable productivity. It lets you write beautiful code by favoring convention over configuration [27].

All these three programming languages/frameworks could lead interesting security research. Important topics are how metacharacters are treated, which access control models are supported, etc.

In this appendix we present the source code for a simple web page that is vulnerable SQL-injection (see listing A.1). It prints the query that is executed before executing it. However, this example only works if the MySQL JDBC driver is configured according to appendix F.2.

Listing A.1: Login form

```
1   <%@ page import="java.sql.*" %>
2
3   <html>
4     <head>
5       <title>form.jsp</title>
6     </head>
7     <body>
8       <%@ include file="databaseConnect.jsp" %>
9       <form action="loginform.jsp" method="GET">
10        <table>
11          <tr>
12            <td>
13              Username:
14            </td>
15            <td>
16              <input type="text" name="username"/>
17            </td>
18          </tr>
19          <tr>
20            <td>
21              Password:
22            </td>
23            <td>
24              <input type="password" name="password" />
25            </td>
26          </tr>
27          <tr>
28            <td></td>
29            <td>
30              <input name="submit" value="Submit" type="submit
                    "/>
31              <input name="reset" value="Reset" type="reset"/>
32            </td>
33          </tr>
34
35        </table>
```

```
36        </form>
37
38        <br/>
39        <br/>
40
41        <%
42          String username = request.getParameter("username");
43          String password = request.getParameter("password");
44
45          if(username != null && password != null)
46          {
47            String sqlQuery = "SELECT * FROM users WHERE
                  username='"+username+"' AND password='"+password
                  +"';";
48          out.println("Executing query...:<br/>");
49          out.println(sqlQuery);
50
51          out.println("<br/><br/>");
52          try
53          {
54            rs = statement.executeQuery(sqlQuery);
55            out.println("Iterating over resultset rs...<br/>")
                  ;
56            while(rs.next())
57            {
58              out.println("User: "+rs.getString("username")+"
                    -- Password: "+rs.getString("password")+".<
                    br/>");
59            }
60          }
61          catch(Exception e)
62          {
63            out.println("Query failed because you got a...<br
                  />");
64            out.println(e);
65          }
66
67          }
68        %>
69        <%@ include file="databaseDisconnect.jsp" %>
70      </body>
71  </html>
```

REGISTER USER SOURCE

This appendix illustrates another SQL-injection vulnerability. It is similar to the JSP listed in appendix A and it is possible to inject all sorts of SQL statements. Again, the MySQL JDBC driver has be configured according to appendix F.2.

Listing B.1: Register a new user

```
1  <%@ page import="java.sql.*" %>
2
3  <html>
4    <head>
5      <title>registeruser.jsp</title>
6    </head>
7    <body>
8      <%@ include file="databaseConnect.jsp" %>
9      <form action="registeruser.jsp" method="GET">
10       <table>
11         <tr>
12           <td>
13             Username:
14           </td>
15           <td>
16             <input type="text" name="username"/>
17           </td>
18         </tr>
19         <tr>
20           <td>
21             Password:
22           </td>
23           <td>
24             <input type="password" name="password" />
25           </td>
26         </tr>
27         <tr>
28           <td></td>
29           <td>
30             <input name="submit" value="Submit" type="submit
                 "/>
31             <input name="reset" value="Reset" type="reset"/>
32           </td>
33         </tr>
34
35       </table>
36     </form>
```

```jsp
37
38      <br/>
39      <br/>
40
41      <%
42         String username = request.getParameter("username");
43         String password = request.getParameter("password");
44
45         if(username != null && password != null)
46         {
47            String sqlQuery = "INSERT INTO users (username,
                  password) VALUES ('"+username+"','"+password+"')
                  ";
48         out.println("Executing update...:<br/>");
49         out.println(sqlQuery);
50
51         out.println("<br/><br/>");
52         try
53         {
54            int nofUpdatedRows = statement.executeUpdate(
                  sqlQuery);
55            if(nofUpdatedRows == 1)
56               out.println(nofUpdatedRows+" row were updated in
                     the database.");
57            else
58               out.println(nofUpdatedRows+" rows were updated
                     in the database.");
59
60         }
61         catch(Exception e)
62         {
63            out.println("Query failed because you got a...<br
                  />");
64            out.println(e);
65         }
66
67      }
68      %>
69      <%@ include file="databaseDisconnect.jsp" %>
70      </body>
71  </html>
```

APPENDIX

COOKIE PAGE SOURCE

The example given in listing C.1, is somewhat different than the examples we saw in appendix A and B. It is vulnerable to cross-site scripting, and since it uses a cookie to save session information, it is possible to inject scripts similar to those we illustrated in section 2.2.3 to act on another user's behalf.

Listing C.1: Page creating a cookie

```
1   <%@ page import="java.util.Date" %>
2   <%@ page import="java.net.*" %>
3   <%@ page import="java.sql.*" %>
4
5   <%
6     //Creating a cookie for the user entering the web page...
7     Date now = new Date();
8     String timeStamp = now.toString();
9     Cookie cookie = new Cookie("RedirectTime", timeStamp);
10    cookie.setMaxAge(365*24*60*60);
11    response.addCookie(cookie);
12  %>
13
14  <html>
15    <head>
16      <title>cookiepage.jsp</title>
17    </head>
18
19    <body>
20      <%@ include file="databaseConnect.jsp" %>
21      <%
22        out.println("Welcome to this page...<br/>");
23        out.println("Your associated session identifier is: ")
             ;
24        out.println(cookie.getValue()+"<br/><br/>");
25      %>
26
27      <table>
28        <form method="get" action="cookiepage.jsp">
29          <tr>
30            <td>
31              Comment:
32            </td>
33            <td>
34              <input type="text" name="comment"/>
35            </td>
```

```
36          </tr>
37          <tr>
38            <td></td>
39            <td>
40              <input type="submit" name="submit" value="Submit
                   " />
41              <input type="reset" name="reset" value="Reset" /
                   >
42            </td>
43          </tr>
44        </form>
45      </table>

47      <br/>
48      <br/>

50      <%
51        String comment = request.getParameter("comment");

53        if(comment != null)
54        {
55          comment = comment.trim();
56          if(!comment.equals(""))
57          {
58            String sqlQuery = "INSERT INTO xss(value) VALUES('
                   "+comment+"')";
59            out.println("Executing query...:<br/>");
60            out.println(sqlQuery+"<br/><br/>");

62            try
63            {
64              int nofUpdatedRows = statement.executeUpdate(
                   sqlQuery);
65              if(nofUpdatedRows == 1)
66                out.println(nofUpdatedRows+" row were updated
                     in the database.");
67              else
68                out.println(nofUpdatedRows+" rows were updated
                      in the database.");
69            }
70            catch(Exception e)
71            {
72              out.println("Query failed because you got a...<
                   br/>");
73              out.println(e);
74              out.println("<br/><br/>");
75            }
76          }
77        }
78      %>

80      Comments given so far:<br/>
81      <%
82        String sql = "SELECT * FROM xss";
```

```
83      rs = statement.executeQuery(sql);
84      int cNumber = 0;
85
86      while(rs.next())
87      {
88        String value = rs.getString("value");
89        out.println("<b>Comment "+cNumber+":</b> "+value+"<
            br/>");
90        cNumber++;
91      }
92    %>
93
94    <%@ include file="databaseDisconnect.jsp" %>
95    </body>
96  </html>
```

D

STEAL SESSION SOURCE

When an attack script has been injected, there is a need for an attacker controlled page that can save the stolen information. One simple example page providing this functionality is given in listing D.1. This example stores the information in a database, but there is possible to extend the source code such that an email is sent to the attacker each time fresh session information is hijacked.

Listing D.1: Saving the stolen session information

```jsp
1  <%@ page import="java.sql.*" %>
2  <html>
3    <head>
4      <title>
5        stealsession.jsp
6      </title>
7    </head>
8
9    <body>
10     <%@ include file="databaseConnect.jsp" %>
11     Stealing a user session identifier...<br/><br/>
12
13     <%
14       String fetchedCookie = request.getParameter("what");
15       out.println("The forwarded cookie contains the
            following information...<br/>");
16       out.println(fetchedCookie);
17
18       //Saving session information in the database....
19       String sql = "INSERT INTO stolen_session_info(value)
            VALUES('"+fetchedCookie+"')";
20       int nofUpRows = statement.executeUpdate(sql);
21
22
23       if(nofUpRows == 1)
24         out.println("<br/>"+nofUpRows+" row were updated in
              the database.");
25       else
26         out.println("<br/>"+nofUpRows+" rows were updated in
              the database.");
27
28       //Redirecting back to the original page
29       String whatNext = request.getParameter("whatnext");
30       if(whatNext != null && !whatNext.equals(""))
31       {
```

```
32        out.println("<br/>Redirecting  to  "+whatNext+"...<br
              />");
33        out.println("<script>document.location.replace(\""+
              whatNext+"\")</script>");
34      }
35    else
36    {
37        out.println("No whatnext-parameter given...
              Redirecting  to  google...");
38        out.println("<script>document.location.replace(\"
              http://google.com\")</script>");
39      }

40

41    %>

42

43    <%@  include  file="databaseDisconnect.jsp" %>
44  </body>
45 </html>
```

PREPARED STATEMENT SOURCE

In this appendix we present a somewhat updated version of the JSP listed in appendix B. However, in listing E.1 we use `PreparedStatements` instead of concatenating the query dynamically. This ensures that the SQL metacharacters are handled properly and no SQL-injections are possible.

Listing E.1: Using a prepared statement

```jsp
<%@ page import="java.sql.*" %>

<html>
  <head>
    <title>form.jsp</title>
  </head>
  <body>
    <%@ include file="databaseConnect.jsp" %>

    <form action="prepared.jsp" method="GET">
      Input some text to insert into the database:<br/><br/>
      <input type="text" name="textField"/><br/>

      <input type="submit" name="submit" value="Submit" />
      <input type="reset" name="reset" value="reset" />
    </form>

    <%
      //Insert into the database
      String text = request.getParameter("textField");
      if(text != null && !text.equals(""))
      {
        //Dynamically building the query through string
            concatenation
        //String sql = "INSERT INTO prepared VALUES('"+text+
            "')";

        //Using a prepared statement
        String preparedSql = "INSERT INTO prepared VALUES(?)
            ";
        PreparedStatement ps = connection.prepareStatement(
            preparedSql);

        //Injecting the data into the precompiled query
        ps.setString(1, text);
```

```
33    out.println("Executing query...:<br/>"+preparedSql+"
         <br/>");

34

35    try
36    {
37      //Executing the dynamic query
38      //statement.executeUpdate(sql);

39

40      //Executing the PreparedStatement object
41      ps.executeUpdate();

42

43      out.println("Query completed successfully.<br/>");
44    }
45    catch(Exception e)
46    {
47      out.println("Query failed because you got a...<br
         />");
48      out.println(e);
49    }

50

51    }
52  %>

53

54  <br/>
55  <br/>

56

57  <%
58    //Fetch the data stored in the database
59    out.println("The database now contains the following
         values...:<br/>");

60

61    String sql = "SELECT * from prepared";
62    try
63    {
64      rs = statement.executeQuery(sql);
65      int valNr = 0;
66      while(rs.next())
67      {
68        out.println("Value "+valNr+": "+rs.getString("
             value")+"<br/>");
69        valNr++;
70      }
71    }
72    catch(Exception e)
73    {
74      out.println("Query failed because you got a...<br/>"
             );
75      out.println(e);
76    }
77  %>

78

79

80

81  <%@ include file="databaseDisconnect.jsp" %>
```

```
82      </body>
83  </html>
```

Appendix F

MySQL and JDBC

MySQL is an DBMS licensed under GNU GPL. In this appendix we will discuss some of the security measures MySQL itself and the JDBC driver implements to improve application security.

F.1 General security measures

The most important security measure, is the access control. MySQL uses a fine-grained ACL model where it is possible defined access rights per database table. Some users are allowed to create and delete tables, whereas others are only allowed to read and update the table contents. These access rights should be granted in a very strict manner. Each user should not have access for more operations and tables than needed.

You could ask what the point is using this sort of access control when you are already using some sort of domain object- or other web application oriented access control. The fact is that two levels of access control can reduce the performance of the application. But, if there is a bug in one of them, the other could (probably!) ensure that the application is not vulnerable to attackers. Additionally, practicing defence in depth is a well-known and effective security principle stated by several sources, [65, 70]. Notice that using predictable passwords may compromise this access control completely because an attacker may act on another user's behalf.

F.2 Semicolon as metacharacter

Even if MySQL itself provides proper authentication and access control modules, the company recognized that their DBMS were often involved in SQL-injection attacks. To avoid these problems, they pointed out the importance of semicolon. The JDBC driver looks for semicolon in the queries it is about to send to the DBMS and if one is found, an `Exception` is thrown. This is the default behaviour of the driver, but it is possible to use the allowMultiQueries parameter to disable this functionality [3].

There are several advantages and disadvantages connected to this approach. The most important advantage is that the application security is improve by default. Some of the disadvantages are:

Fail to identify other metacharacters
Metacharacters like # and ' are still allowed. This means that an attacker may still bypass or even add conditions the queries. These attacks may be used to bypass application authentication or other restrictions.

Hide details to the developer
When the JDBC driver implements semicolon detection without pointing out what the real problem is and how it can be solved, the developer will continue

to introduce vulnerabilities. Instead of throwing a generic `MySQLSyntaxErrorException`, it would be better to throw a security exception indicating that the developer should `PreparedStatements` to avoid these and similar problems.

ACRONYMS

Access Control List (ACL)

An ACL is one out of several ways to implement access control in software applications. An ACL lists, for each object, users and their permissions [66].

Aspect-Oriented Programming (AOP)

AOP is a way to integrate a module (e.g. logging) containing a limited set of functionality into a software application without intermingling the module code with the business object. More details on how this is implemented in Spring is found in section 3.3.3.

Application Programming Interface (API)

An API is an interface that is used when developing software application. It allows a developer to focus on using the abstractions defined by API classes and he/she does not have to care about all the implementation details. The Java 2 Platform implements several APIs, among them the Java 2 SDK API [50].

Central Authentication Service (CAS)

CAS is a single sign-on implemention by Yale University's Technology and Planning group. It allows users to authenticate once and then, by using tickets, they are granted access to several applications [73].

Data Access Objects (DAO)

According to [73], "DAOs exist to provide a means to read and write data to the database. They should expose the functionality through an interface by which the rest of the application will access them."

Database Management System (DBMS)

[25] defines a DBMS as "a powerful tool for creating and managing large amounts of data efficiently and allowing it to persist over long periods of time, safely.". A DBMS usually provides capabilities like persistent storage, programming interface, and transaction management.

Document Type Definition (DTD)

A DTD describes the entities allowed in one type of XML documents [59].

Enterprise Java Beans (EJB)

[46] states that an EJB is"the server-side component architecture for the Java 2 Platform, Enterprise Edition (J2EE) platform. EJB technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology."

HyperText Markup Language (HTML)

HTML is a markup language used to create web documents. HTML allows users to produce Web pages that include text, graphics, and pointers to other web pages. HTML is a markup language, a language for describing how documents are to be formatted [68].

HyperText Transfer Protocol (HTTP)

By [68], "HTTP is defined as the transfer protocol used throughout the World Wide Web. It specifies what messages clients may send to servers and what responses they get back in return. Each interaction consists of one ASCII request followed by one RFC 822 MIME-like response. All clients and all servers must obey this protocol. The protocol itself is specified in RFC 2616."

Inversion of Control (IoC)

IoC is way to resolve an object's dependencies. The idea is to let a container inject the dependencies instead giving each object the responsibility to look up their dependencies. More details are found in section 3.3.2.

Java Enterprise Edition (Java EE)

[48] states that Java EE is "Java Platform, Enterprise Edition is the industry standard for developing portable, robust, scalable and secure server-side Java applications. Building on the solid foundation of Java SE, Java EE provides web services, component model, management, and communications APIs that make it the industry standard for implementing enterprise class service-oriented architecture (SOA) and web 2.0 applications."

Java Authentication and Authorization Service (JAAS)

By [47], JAAS is "... a set of APIs that enable services to authenticate and enforce access controls upon users. It implements a Java technology version of the standard Pluggable Authentication Module framework, and supports user-based authorization."

Java Database Connectivity (JDBC)

According to [55], the "JDBC technology is an API (included in both J2SE and J2EE releases) that provides cross-DBMS connectivity to a wide range of SQL databases and access to other tabular data sources, such as spreadsheets or flat files."

Java Development Kit (JDK)

The JDK is a piece of software aimed at Java developers. It contains a compiler, a javadoc generator, a debugger, a runtime environment, etc. [50].

Java Data Objects (JDO)

JDO is a standard interface-based Java model abstraction of persistence developed as JSR 12. An implementation of this by the Apache Software Foundation is called Apache ObJectRelationalBridge [13, 67]

JavaServer Faces (JSF)

As stated by [45], JSF is a component architecture, a standard set of UI widgets, and an application infrastructure. The JSF itself is only specification named JSR-127. There are at least a couple of implementations available (e.g. MyFaces). More details are found in section 3.5

JavaServer Page (JSP)

A JSP is a file consisting of HTML and Java code. A JSP compiler generates a servlet from the JSP file and this servlet is executed by a servlet container (e.g. Apache Tomcat)

Java Specification Request (JSR)

According to [60], a JSR is "a Java Specification Request. This is the document submitted to the Process Management Office by one or more members to propose the development of a new specification or significant revision to an existing specification."

JavaServer Pages Standard Tag Library (JSTL)

As Sun Microsystems state in [54], JSTL "encapsulates as simple tags the core functionality common to many Web applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags."

Model View Controller (MVC)

MVC is a pattern that separates the view implementation from the processing logic and model data. We discuss a MVC variant called Model 2 in section 3.2.

Object Relational Mapping (ORM)

ORM is a mapping tool that tries to remove the paradigm mismatch that exists between today's DBMS and object-oriented programming languages [8].

Software Development Kit (SDK)

A SDK is a set of software tools needed when developing applications. It typically includes a compiler, a debugger, but possibly also other handy tools. The JDK is a SDK for Java.

Structured Query Language (SQL)

SQL is a query language used to fetch and update information in databases.

Secure Sockets Layer (SSL)

SSL is a two-layered protocol that is designed on top of TCP to "provide a reliable end-to-end secure service" [66].

Uniform Resource Locator (URL)

An URL is the address of a single file located on the World Wide Web. Each URL consists of three parts: the protocol (e.g. http://), the DNS name of the server on which the page is located (e.g. www.vg.no) and a unique local filename (e.g. index.html) [68].

Wireless Markup Language (WML)

WML is markup language similar to HTML except that it is designed for wireless applications. [12] provides a WML language reference.

eXtensible Markup Language (XML)

XML is a generic and extensible markup language based on SGML [59].

GLOSSARY

Java Bean

[51] states that the "JavaBeans technology is the component architecture for the Java 2 Platform, Standard Edition (J2SE). Components (JavaBeans) are reusable software programs that you can develop and assemble easily to create sophisticated applications. JavaBeans technology is based on the JavaBeans specification [52]."

Java Servlet Technology

Sun Microsystems [49] claims that the "Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. A servlet can almost be thought of as an applet that runs on the server side–without a face."

BIBLIOGRAPHY

[1] MySQL AB. *MySQL 5.0 Reference Manual*. `http://dev.mysql.com/doc/refman/5.0/en/comments.html`. Accessed March 6, 2006. 9

[2] MySQL AB. *MySQL 5.0 Reference Manual*. `http://dev.mysql.com/doc/refman/5.0/en/drop-table.html`. Accessed March 7, 2006. 10

[3] MySQL AB. *MySQL 5.0 Reference Manual*. `http://dev.mysql.com/doc/refman/5.0/en/cj-configuration-properties.html`. Accessed June 1, 2006. 85

[4] MySQL AB. *MySQL AB*. `http://www.mysql.com`. Accessed May 19, 2006. 3

[5] Ben Alex. *Acegi Security System for Spring - Reference documentation*. `http://acegisecurity.org/docbook/acegi.html`. Accessed April 3, 2006. 45, 47

[6] AOP Alliance. *AOP Alliance*. `http://aopalliance.sourceforge.net/`. Accessed March 20, 2006. 19

[7] Naveen Balani. *The Spring series, Part 1: Introduction to the Spring framework*. `http://www-128.ibm.com/developerworks/library/wa-spring1/`. Accessed March 20, 2006. 19

[8] Christian Bauer and Gavin King. *Hibernate in Action*. Manning Publications Co, 2005. ISBN: 1-932-39415-X. 27, 89

[9] Microsoft Corporation. *Microsoft ActiveX Controls Overview*. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaxctrl/html/msdn_actxcont.asp`. Accessed February 27, 2006. 2

[10] Mozilla Corporation. *Firefox - rediscover the web*. `http://www.mozilla.com/firefox/`. Accessed March 13, 2006. 11

[11] Oracle Corporation. *Oracle ADF Faces Components*. `http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/index.html`. Accessed May 19, 2006. 3

[12] Refnes Data. *WML Reference*. `http://www.w3schools.com/wap/wml_reference.asp`. Accessed June 1, 2006. 89

[13] Apache Software Foundation. *Apache ObJectRelationalBridge*. `http://db.apache.org/ojb/`. Accessed March 22, 2006. 88

[14] Apache Software Foundation. *iBATIS*. `http://ibatis.apache.org/`. Accessed March 22, 2006. 20

[15] The Apache Software Foundation. *Apache HTTP Server Project*. `http://httpd.apache.org`. Accessed March 13, 2006. 11

[16] The Apache Software Foundation. *The Apache MyFaces Project*. `http://myfaces.apache.org/`. Accessed March 16, 2006. 3

[17] The Apache Software Foundation. *Apache Struts Project*. `http://struts.apache.org/`. Accessed March 16, 2006. 3

[18] The Apache Software Foundation. *Apache Tomcat*. `http://tomcat.apache.org/`. Accessed May 19, 2006. 3

[19] The Apache Software Foundation. *Commons Scaffold*. `http://jakarta.apache.org/commons/sandbox/scaffold/`. Accessed May 1, 2006. 27

[20] The Apache Software Foundation. *Taglibs*. `http://jakarta.apache.org/taglibs/`. Accessed March 30, 2006. 42

[21] The Apache Software Foundation. *Velocity*. `http://jakarta.apache.org/velocity/`. Accessed May 1, 2006. 27

[22] The Apache Software Foundation. *Velocity*. `http://jakarta.apache.org/velocity/`. Accessed March 30, 2006. 42

[23] The OWASP Foundation. *Top Ten*. `http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`. Accessed May 3, 2006. 5, 50, 69

[24] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. `http://martinfowler.com/articles/injection.html`. Accessed March 20, 2006. 20

[25] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems – The Complete Book*. Prentice Hall, 2002. ISBN: 0-130-31995-3. 87

[26] David Geary and Cay Horstmann. *Core JavaServer Faces*. Pretience Hall, 2004. ISBN: 0-131-46305-5. 35

[27] David Heinemeier Hansson. *Ruby on Rails*. `http://www.rubyonrails.org/`. Accessed June 6, 2006. 70

[28] James Holmes. *Securing Struts Applications*. `http://www.devarticles.com/c/a/Java/Securing-Struts-Applications/`. Access May 4, 2006. 51, 52

[29] Sverre H. Huseby. *Common Security Problems in the Code of Dynamic Web Applications*. `http://shh.thathost.com/text/common-web-code-vulns.pdf`. Accessed March 2, 2006. 5, 7

[30] Sverre H. Huseby. *PenProxy - a web application pen-test proxy*. `http://shh.thathost.com/pub-java/#PenProxy`. Accessed March 14, 2006. 6

[31] Sverre H. Huseby. *Innocent code - a security wake-up call for web programmers*. John Wiley & Sons, Ltd, 2004. ISBN: 0-470-85744-7. 5, 10, 11, 12, 15, 42

[32] Ted Husted, Cedric Dumoulin, George Franciscus, and David Winterfeldt. *Struts in Action*. Manning Publications Co, 2003. ISBN: 1-930-11050-2. 17, 26, 51, 52

[33] IETF. *Hypertext Transfer Protocol – HTTP/1.0*. `http://www.ietf.org/rfc/rfc1945.txt`. Accessed March 13, 2006. 11

[34] IETF. *Hypertext Transfer Protocol – HTTP/1.1*. `http://www.ietf.org/rfc/rfc2616.txt`. Accessed March 13, 2006. 11

[35] Sun Microsystems Inc. *Java 2 Platform Standard Edition 5.0 API Specification*. `http://java.sun.com/j2se/1.5.0/docs/api/index.html`. Accessed March 14, 2006. 14

[36] Sun Microsystems Inc. *Java Applets*. `http://java.sun.com/applets/`. Accessed February 27, 2006. 2

[37] Sun Microsystems Inc. *JavaServer Faces Technology*. `http://java.sun.com/javaee/javaserverfaces/`. Accessed March 16, 2006. 30

[38] java source.net. *Open Source Web Frameworks in Java*. `http://java-source.net/open-source/web-frameworks`. Accessed March 16, 2006. 2, 70

[39] Inc JBoss. *hibernate.org - Hibernate*. `http://www.hibernate.org/`. Accessed March 20, 2006. 19, 20, 27

[40] jGuard Project. *jGuard*. `http://www.jguard.net`. Accessed May 27, 2006. 62

[41] Jukka "Yucca" Korpela. *Methods GET and POST in HTML forms - what's the difference?* `http://www.cs.tut.fi/~jkorpela/forms/methods.html`. Accessed March 16, 2006. 9

[42] Acegi Technology Pty Limited. *Acegi Security System for Spring*. `http://acegisecurity.org`. Accessed April 3, 2006. 3, 45

[43] Acegi Technology Pty Limited. *Overview (Acegi Security System for Spring*. `http://acegisecurity.org/multiproject/acegi-security/apidocs/index.html`. Accessed April 3, 2006. 45

[44] Cafesoft LLC. *Tomcat Security Overview and Analysis*. `http://www.cafesoft.com/products/cams/tomcat-security.html`. Accessed May 31, 2006. 57

[45] Kito D. Mann. *JavaServer Faces in Action*. Manning Publications Co, 2005. ISBN: 1-932-39412-5. 18, 30, 34, 55, 56, 88

[46] Sun Microsystems. *J2EE - Enterprise JavaBeans Technology*. `http://java.sun.com/products/ejb/`. Accessed March 27, 2006. 87

[47] Sun Microsystems. *Java Authentication and Authorization Service (JAAS)*. `http://java.sun.com/products/jaas/`. Accessed June 1, 2006. 88

[48] Sun Microsystems. *Java Platform, Enterprise Edition*. `http://java.sun.com/javaee/index.jsp`. Accessed March 27, 2006. 88

[49] Sun Microsystems. *Java Servlet Technology*. `http://java.sun.com/products/servlet/`. Accessed March 27, 2006. 91

[50] Sun Microsystems. *Java Technology*. `http://java.sun.com`. Accessed June 1, 2006. 87, 88

[51] Sun Microsystems. *JavaBeans*. `http://java.sun.com/products/javabeans/`. Accessed March 27, 2006. 91

[52] Sun Microsystems. *JavaBeans Spec*. `http://java.sun.com/products/javabeans/docs/spec.html`. Accessed March 27, 2006. 91

[53] Sun Microsystems. *JavaServer Faces Technology Download*. `http://java.sun.com/javaee/javaserverfaces/download.html`. Accessed May 19, 2006. 3

[54] Sun Microsystems. *JavaServer Pages Standard Tag Library*. `http://java.sun.com/products/jsp/jstl/index.jsp`. Accessed March 30, 2006. 89

[55] Sun Microsystems. *JDBC Technology*. `http://java.sun.com/products/jdbc/`. Accessed March 27, 2006. 88

[56] Sun Microsystems. *JSP Tag Libraries*. `http://java.sun.com/products/jsp/taglibraries/index.jsp`. Accessed March 30, 2006. 42

[57] Sun Microsystems. *JSR-000052 A Standard Tag Library for JavaServer Pages*. `http://jcp.org/aboutJava/communityprocess/final/jsr052/index2.html`. Accessed March 30, 2006. 42

[58] Sun Microsystems. *A Swing Architecture Overview*. `http://java.sun.com/products/jfc/tsc/articles/architecture/`. Accessed March 27, 2006. 17, 31

[59] Simon North and Paul Hermans. *Lær XML på 21 dager*. Tano Aschehoug, 2000. ISBN: 8-251-83955-6. 87, 89

[60] Java Community Process. *The Java Community Process Program FAQ*. `http://www.jcp.org/en/introduction/faq`. Accessed March 27, 2006. 89

[61] FreeMarker Project. *FreeMarker*. `http://freemarker.sourceforge.net/`. Accessed March 30, 2006. 42

[62] JSF-Comp Project. *JSF-Comp*. `http://jsf-comp.sourceforge.net/`. Accessed May 22, 2006. 3, 58, 59

[63] JSF-Security Project. *JSF-Security*. `http://jsf-security.sourceforge.net/`. Accessed May 22, 2006. xiii, 3, 58

[64] Matt Raible. *Java Web Frameworks*. `http://www.virtuas.com/files/osl-jwf-01.pdf`. Accessed March 16, 2006. 2

[65] Chris Shiflett. *Essential PHP Security*. O'Reilly, first edition edition, 2006. ISBN: 0-596-00656-X. 5, 15, 85

[66] William Stallings. *Network Security Essentials - Applications and Standards*. Pearson Education International, second edition edition, 2003. ISBN: 0-131-20271-5. 87, 89

[67] Inc Sun Microsystems. *Java Data Objects*. `http://java.sun.com/products/jdo/`. Accessed March 22, 2006. 88

[68] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education International, forth edition edition, 2003. ISBN: 0-130-38488-7. 87, 88, 89

[69] Bart van Riel. *Spring Acegi Tutorial*. `http://home.hccnet.nl/bart.van.riel/SpringAcegiTutorial/PDF/SpringAcegiTutorial.pdf`. Accessed April 3, 2006. 45

[70] John Viega and Gary McGraw. *Building Secure Software - How to Avoid Security Problems the Right Way*. Addison-Wesley, 2005. ISBN: 0-201-72152-X. 37, 85

[71] W3C. *Forms in HTML documents*. `http://www.w3.org/TR/html4/interact/forms.html`. Accessed March 3, 2006. 8

[72] w3Schools.com. *JavaScript Tutorial*. `http://www.w3schools.com/js/default.asp`. Accessed February 27, 2006. 2

[73] Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning Publications Co, 2005. ISBN: 1-932-39435-4. 18, 19, 23, 24, 38, 40, 45, 87

[74] www.springframework.org. *Spring Framework*. `http://www.springframework.org`. Accessed March 16, 2006. 3