AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Electronics, Communications and Automation

Ville Viskari

Improving the architecture of a content management system with Seam

Master´s thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology in the Degree Programme in Communications Engineering.

Espoo, 17.4.2010

Supervisor: Professor Heikki Saikkonen

Instructor: Professor Heikki Saikkonen

AALTO YLIOPISTO Diplomityön tiivistelmä

TEKNILLINEN KORKEAKOULU

Tekijä:	Viskari, Ville
Työn nimi:	Sisällönhallintajärjestelmän arkkitehtuurin parantaminen Seamin avulla
Päivämäärä:	Huhtikuu 2010
Sivumäärä:	57 sivua
Pääaine:	Ohjelmistojärjestelmät
Työn valvoja:	Professori Heikki Saikkonen
Työn ohjaaja:	Professori Heikki Saikkonen
Avainsanat:	JSF, Seam, arkkitehtuuri, sisällönhallintajärjestelmä, verkkosovellus

Tässä diplomityössä tutkitaan, kuinka sisällönhallintajärjestelmän arkkitehtuuria voidaan parantaa Java-pohjaisella sovelluskehyksellä. Sisällönhallintajärjestelmänä on Ambientia Oy:n kehittämä Content Manager ja sovelluskehyksenä on J2EE-tekniikkaan ja JSF standardiin perustuva JBoss Seam. Työn päätavoitteena on lyhentää asiakasprojektien läpimenoaikaa ja toisaalta helpottaa sivupohjien ohjelmointityötä.

Sisällönhallintajärjestelmiä on kehitetty, jotta asiakas voisi muokata verkkosivustonsa sisältöä helposti itse sen sijaan, että muutoksia täytyisi pyytää sivustoa ylläpitävältä yritykseltä. Sisällönhallintajärjestelmä koostuu julkisesta puolesta, joka näyttää sivuston loppukäyttäjille, ja hallintasovelluksesta, jonka kautta julkisen puolen sisältöä hallinnoidaan.

Ambientia Oy käyttää talon sisällä kehitettyä modulaarista sisällönhallintajärjestelmää, joka on monen vuoden kehitystyön tulos. Sovellus perustuu Java Servletteihin, ja sivupohjat on toteutettu Apache Velocity -tekniikalla.

Tässä työssä toteutettiin osa olemassa olevan sisällönhallintajärjestelmän julkisen puolen toiminnallisuudesta Seam sovelluskehyksen avulla. Järjestelmä listaa sisällönhallintajärjestelmän navigaatiopuun ja näyttää ilmoitustaulumodulin viestit verkkosivulla.

Työssä huomattiin, että Seam ei sovellu ajateltuun tarkoitukseen. Ongelmaksi muodostuivat järjestelmien arkkitehtuurissa havaitut suuret erot: Seam on J2EE-lähtöinen malli, kun taas sisällönhallintajärjestelmän komponentit eivät tarjoa J2EE-arkkitehtuurin vaatimia rajapintoja tarpeeksi. Lisäksi, sivupohjien toteuttaminen Seamin tarjoamilla työkaluilla on monimutkaista.

AALTO UNIVERSITY Abstract of Master's Thesis
SCHOOL OF SCIENCE AND TECHNOLOGY

Author:	Viskari, Ville
Title:	Improving the architecture of a content management system with Seam
Date:	April 2010
Faculty:	Faculty of Electronics, Communications and Automation
Pages:	57 pages
Major:	Software systems
Supervisor:	Professor Heikki Saikkonen
Instructor:	Professor Heikki Saikkonen
Keywords:	JSF, Seam, architecture, content management system, web application

This thesis is a case study on improving the architecture of a content management system with a Java application framework. The content management system in question is an in-house developed Content Manager made by Ambentia Ltd, and the framework is JBoss Seam J2EE application framework which is based on the JSF standard. The primary goals in this study were to reduce the time spent in customer projects and to ease the development of Web page templates.

A content management systems have been created so that the customer is able to modify the contents of their web pages without the need to request these changes from the service providing company. A content management system has a public side which contains the public web site, and an administration application from which the public content is modified.

Ambientia Ltd is using a modular content management system, which is the result of many years of in-house development. The application is implemented using Java Servlets and the page templates are written using Apache Velocity templating engine.

In this Thesis a part of the content management system's public application functionality was implemented using Seam framework. The system lists the navigation tree and displays the messages of a bulletin board module on the page.

It was concluded that Seam does not suit the desired purpose. Main problem was the large architectural differences of the old CMS system and Seam. Seam is a J2EE based system, while the CMS application does not provide the interfaces required to be used in an enterprise environment. In addition, the implementation of page templates using the components provided by Seam was found to be too complicated.

FOREWORDS

This Master’s Thesis is written in Helsinki University of Technology for the Faculty of Electronics, Communications and Automation. The writing of this Thesis took place mainly during the winter and spring of 2010 in few intensive sessions.

I'd like to thank Ambientia Ltd. for giving me this opportunity. Especially my thanks goes to Tero Tielinen and Jari Saukkonen who gave me invaluable guidance during the practical phase of the project. Jari introduced me to the technology that was required and Tero explained me their internal processes and how they relate to the work done by me.

I appreciate the feedback and critique given by my supervisor Heikki Saikkonen. Without the critique this work could have taken a whole new direction. With he's guidance I started to understand the requirements of this thesis. He gave me good advice on how to get started with the writing process.

I'd like to thank my wife Pauliina and the rest of my family for supporting me during this period. For several years they have encouraged me to finally graduate.

Espoo 27.4.2010,

Ville Viskari

TABLE OF CONTENTS

TERMS AND ABBREVIATIONS

AJAX

Asynchronous Javascript And XML

API

Application Programming Interface

CMS

Content Management System

Cookie

Small amount of data that is stored in the client browser and sent back to the server with every request.

DOM

Document Object Model

EJB

Enterprise Java Bean

Hibernate

ORM library for Java applications

HTML

HyperText Markup Language

HTTP

HyperText Transfer Protocol

IoC

Inversion of Control, dependency injection

Java Bean

A simple Java class which contains private properties and exposes read and/or write methods for manipulating them.

J2EE

Java 2 Platform Enterprise Edition. Currently known as Java EE (since version 5).

JSF

Java Server Faces. A standard web application framework.

JSP

Java Server Pages

JSTL

Java Server Pages Standard Tag Library

MVC

Model View Controller. A common design pattern used with user interface implementations.

ORM

Object Relational Mapping

POJO

Plain Old Java Object.

Servlet

Java class that executes inside an web application and responds to HTTP requests.

URI

Universal Resource Identifier

URL

Universal Resource Locator

XML

eXtensible Markup Language

1. INTRODUCTION

This thesis is a case study on improving the architecture of a content management system with a Java application framework. The content management system in question is an in-house developed Content Manager made by Ambentia Ltd, and the framework is JBoss Seam J2EE application framework which is based on the JSF standard. The primary goals in this study were to reduce the time spent in customer projects and to ease the development of Web page templates.

Traditionally web pages have been developed in a way that the site implementing company is also responsible for the content that is displayed on the pages. The customer has not been able to update the text and images on the pages or create a new page within the navigation structure. Every time a change is required, the customer has to issue a change request with the service providing company. This is costly for both sides. Customer has to pay for the updates usually in a form of maintenance fee. The providing company has to tie up resources to implement these kinds of most trivial changes. Instead it would be more productive for everyone if the service providing company could concentrate on the development of their core business software.

A content management systems have been created so that this whole process can be handed to the customer. With a content management system the customer is able to create new web pages, change the navigational structure and edit the content on the pages without the need to request these changes from the service providing company every time. There are several advantages in this solution. First there are the cost savings. Secondly the system is less prone to human errors and the customer is able to correct any errors by itself immediately. In the traditional process the message can be altered and misunderstood while it passes through the bureaucracy from the customer to the final implementing developer.

A content management system consists of the public side that will display the web pages that are visible to the end users. Second part of the system is an administration application that is used to manage the content that is displayed on the public side. Additional functionality can and usually is implemented in the system along with these basic features. The most common features are user management and user roles, web form submission and functionality to process the received data and some form of file management. Additional features can be implemented based on customer needs. Depending on the original system, the implementation of custom functionality can be either easy and fast or difficult or time consuming.

This thesis is done for a company named Ambientia Ltd. They have developed a content management system (Content Manager) that is used with several customer projects to design and implement the public web pages for the client. This application was developed originally in a time when there was no alternatives offered in public domain. The application contains all the common functionality of a CMS and this can be extended with modules that provide additional features like form management, news letters and bulletin board messages. There is always a need to customize the basic functionality of the application by designing new modules that are required to meet the customer's needs. The changes affect both the administration side of the application and the page templates on the public side.

1.1.Objectives of the thesis

This Thesis will evaluate if Seam web application framework is suitable future platform for the Content Manager. This evaluation is performed by implementing some features of the public side of Content Manager using the components and tools provided by Seam. The motivation for the evaluation comes from within the company and the effects of the possible changes in the underlying technology would be virtually invisible for the clients that use Content Manager and to the end users that access the public web pages. The administration application of the system is left untouched. The Seam application will run along the side of the old Content Manager as a separate application. After the selected functions are implemented, they are reviewed against the following objectives.

The main objective of this thesis is to decrease the time spent in the customization projects when developing new functionality to the Content Manager based on customer needs. This can be accomplished with modular component design. These components can be added and combined together and moved easily between different projects.

Second objective is to make the implementation phase easier for the developers. This can be accomplished with simple and effective tags that can be used on the pages with no need for additional coding effort by the developer. The development of the tags themselves should be simple and fast so that it is possible to react to changed or new requirements.

The nature of the code and the excessive amount of work that has been invested to existing code base in Content Manager implies that the work done in this thesis has to use existing code for the main business functionality as much as possible. This work forms the foundation for future developments that build on the knowledge gathered here.

This Thesis is structured as follows. Chapter 2 introduces the basic techniques used in a Java web application. Chapter 3 will discuss the more advanced techniques and frameworks used in a more modern web application. Ambientia Content Manager is explained in Chapter 4. Chapter 5 will explain the implementation details and the results are analyzed in chapter 6. Chapter 7 makes the conclusion.

2. JAVA WEB-APPLICATION TECHNIQUES

This chapter will introduce the basics of Java Web application. The information presented in this chapter is based on the J2EE 1.4 Tutorial by Sun ([20], chapter 11).

The purpose of a Web application is to render dynamic content to a Web page. Dynamic content here means that the contents on a displayed web page changes dynamically based on the user interactions with the web site. The opposite of this is a static web site, which contains only a set of HTML pages. Here the contents of a displayed web page will not change unless the source HTML files are modified manually by the site administrator. The most simple Java Web application is a single Servlet that executes in a Web container inside an application server. The server receives a HTTP request and directs the request to the Web application for processing. The application will direct the request to the correct Servlet for execution. The Servlet will return a response based on the information in the request. The contents of the response may be for example a HTML web page.

The Java Servlet API defines the interfaces that are used in a Java Web application. The package name for Java Servlet API is javax.servlet. The basic interfaces defined in this package include Servlet, ServletRequest, ServletResponse and Filter. A sub package called javax.servlet.http introduces a HttpSession interface which enables the application to store some data across several requests which is associated with the clients. The HTTP package also provides a Cookie class which enables the Servlet to send and store HTTP cookies to the client browser.

A Java application container is responsible for the life cycle management of Servlets and provides specific services that may be used by the Servlets that live inside the container.

2.1.Java Servlet

The presentation of the basic classes and interfaces of Java Servlet API is based on my professional knowledge and the J2EE 1.4 Tutorial by Sun ([20], chapter 11).

Java Servlet is a class that implements a Servlet interface defined in a Java Servlet API. Usually the more specific HttpServlet interface is used in conjunction with HTTP requests. Later a class that implements a HttpServlet interface is simply referred to as Servlet. It receives a HttpServletRequest (Request) and writes the contents of the response to a HttpServletResponse (Response). The contents of the response are generated based on the information received in the request.

2.1.1.HttpServlet

HttpServlet class overrides a service() method from Servlet class and implements several new methods, one for each HTTP method[18]. In the *service()* method the request is directed to the corresponding HTTP request specific method. These methods include *doGet(HttpServletRequest, HttpServletResponse)* which is called when the client issues a HTTP GET request. *doPost()* method is called when the client issues a HTTP POST method. Similar servlet methods exist for HEAD, PUT, DELETE, OPTIONS and TRACE requests. The implementations of these methods in HttpServlet class do not have any functionality. The developer will extend this class and creates a subclass of HttpServlet. Then some of the previously mentioned methods are overridden and the actual functionality is written to the overridden methods. Usually only either *doGet()* and/or *doPost()* method is implemented in the subclasses and other methods are not touched. It is possible to override the *service()* method after which all requests are processed by the same code for all HTTP request methods.

2.1.2.HttpServletRequest

The received HTTP request is parsed by the Web container and it constructs a class that implements a HttpServletRequest interface[18]. This class is passed to the Servlet in the method call and it can read information about the request using this class. Depending on the information in the request, the Servlet will render different output to the response. Information that is available in the request include the requested path inside the application, the HTTP request headers and the HTTP request parameters. Along with this information, the request provides information about the client and the receiving server.

The request parameters are read from the HTTP request and they are put in to a Map by the container. The name of the parameter is the key in the Map and the value is a String array of the parameter values. Since HTTP request may contain multiple values with same name, the map value entry is of String array type. The request class provides methods to access these parameters. There is one method for getting the whole parameter map, one that returns the array of values of a request parameter and one convenience method which returns only a single String value of a parameter. In case the parameter name had multiple values associated to it, only the first one is returned by this method.

There are several methods for reading the request path information. These methods can be used to return the complete URL that the client requested, the request URI without host information, the context path, the Servlet path without context path or the query string. Complete URL is the URL that the client used when issuing the request complete with protocol, host and port information and the Servlet path. Any HTTP request parameters are not returned in this path. Request URI is the same as URL, but protocol, server and port information are not returned. Context path is the part of the path that maps to the current web application. Web server may contain several web applications and each of them is mapped to a specific path inside the server. All requests that are received under the specific path are directed to the respective web application. The Servlet path is the part that points to the executing Servlet in the application. A Servlet is mapped under a specific path in the container to serve requests. The Servlet path contains the path information after the context path to the part that points to the executing Servlet. A query string is the part of HTTP GET request that contains the request parameters. This section of the URL comes after all path sections and is separated from the rest of the path with a question mark character.

Request headers are sent by the client Web browser where it explains what kind of content it expects and accepts with the response. This information is available in the HttpServletRequest. The methods for reading this information are similar in functionality to the methods used for reading request parameters.

Another way to pass information in the request is through the use of HTTP Cookies. A HttpServletRequest method named getCookies() returns an array of Cookie classes if any cookies were associated with the request. The cookies may be stored by the browser persistently or they may be used only for the current browsing session.

2.1.3.HttpSession

A browser session is a concept that ties together all requests and responses sent by the client and server in a single session. The session begins with the first request of the client and ends when the client browser exits, explicitly clears its session cookie cache or the session timeouts in the server. A HttpSession provides means to store information between several browser requests[18]. The servlet container will create a new HttpSession object the first time it is requested by the underlying components. Usually it is the Servlet that will request a session through the HttpServletRequest interface. The returned HttpSession object is basically a map that persists over several browser requests. Usually the information stored to the session is related to the requesting client's identity, or has something to do with the business logic of the application. An example could be a shopping cart application where the items of the users shopping cart are stored in the session.

The HttpSession object is stored solely by the web application. No information about the contents in the session is passed to the calling client. Only the session identifier is passed between the client and the server. The identifier is sent to the client and received by the server for each request. This way the application is able to identify the incoming request and attach the associated session with the request. This happens before the execution is passed to the Servlet.

There are two ways to pass the session identifiers between the client and server. First and generally preferable is the use of a session Cookie. The application will generate a session identifier and stores it in a cookie. This cookie is then sent back to the calling client with the first response. Client browser will store this cookie for the duration of the session and sends it back to the server along with each request. This scheme works when the client browser accepts cookies. Some users have denied their browsers to store cookies and in this case the session identifier would get lost with every request. The server would have to recreate a new session for each request.

The second scheme solves this problem. The application will filter the contents of a returned response before the contents are sent to the client. If the response is a HTML page, the application will render the session identifier as a HTTP request parameter to every link that is returned in the response. This way the client will pass the session identifier as one of the request parameters and application is able to fetch the corresponding session from the store and attach it to the request. The received HTTP session parameter is removed from the map of request parameters and it is not passed to the underlying Servlet.

Usually HttpSessions are stored in memory by the application container. Every HttpSession will consume application memory depending on the amount and type of objects that are stored in the session. To prevent the application server from crashing due to running out of memory, the sessions have a specific timeout configured in the application container. After the session times out the container will remove it from the storage and deletes the associated objects along with it freeing the used memory.

2.1.4.HttpServletResponse

The Servlet will send the response to client using a class which implements the HttpServletResponse interface. This class is passed to the Servlet in the method call. Through this class the Servlet is able to set the HTTP response status, the response headers and cookies and write the contents of the response. Text responses are written using a Writer class and binary data is written using a ServletOutputStream class. A reference to these classes is obtained from the HttpServletResponse class. The servlet can not use both of these classes in the same response. A HTTP response type is always either text or binary, not both. An attempt to use both classes

(Writer and OutputStream) for the same response results in an error.

HttpServletResponse contains a buffer for the written data. The response data is written to the buffer and it is flushed periodically to the client. The Servlet may flush the buffer explicitly if needed. Also the Servlet may reset the buffer and clear the data written to it before it is flushed. Data that already has been flushed and written to the client can not be recovered and cleared.

In some cases the Servlet decides that the request has to be redirected to another location. It will then call the sendRedirect(String url) method in the response. This will issue a HTTP 302 response to the client and it will move to the redirected address. After the Servlet calls the redirection method, it is unable to write any further data to the response or it will result in an error.

2.1.5. Filter

A servlet filter is a class that is executed by the container before the request is passed to the servlet[18]. A filter is mapped to a URL pattern or specifically to some servlet. The container decides which filters are applied to the current request and composes a FilterChain in which the filters are executed one after each other. The executing filter is able to modify the incoming request and insert information to the request object or modify the response after the servlet execution has completed. An example code of two filters is presented in Table 1. First filter modifies the request and the second will modify the resulting response.

```
import javax.servlet.*;

// Filter that is executed first by the container
public class Filter1 implements Filter {
    public void destroy(){}
    public void init(FilterConfig filterConfig){}
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain){
        // modify the incoming request
        req.setAttribute(”key”, ”value”);
        chain.doFilter(req,res);
    }
}

// Filter that is executed after filter1
public class Filter2 implements Filter {
    public void destroy(){}
    public void init(FilterConfig filterConfig){}
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain){
        chain.doFilter(req,res);
        // Modify the resulting response after servlet has executed.
        ((HttpServletResponse)res).addHeader(”headerName”, ”headerValue”);
    }
}

```

Table 1: Example code for two filters

A filter is able to block the request processing by not passing the execution further in the filter chain.

2.2.JSP and JSTL

Java Server Pages is a technique that will ease the development of web pages that render dynamic content. JSP is a higher level abstraction of Java Servlets. According to J2EE Tutorial, “a JSP page is a text document that contains two types of text: static data, which can be expressed in any text-based format (such as HTML, SVG, WML, and XML), and JSP elements, which construct dynamic content” [20]. The pages are interpreted by a JSP compiler. The JSP compiler will compile the JSP page to a Java Servlet which is then compiled by standard Java compiler to bytecode and executed by the JVM. So, JSP pages are in fact standard Java Servlets, but the actual Servlet class is hidden by the developer. Formally, a JSP page implements a *JspPage* interface which in turn extends the *Servlet* interface ([18], package javax.servlet.jsp). An example JSP page is presented in Table 2 and it's based on J2EE Tutorial ([20], chapters 12, 14 and 16).

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ page import="com.some.company" %>
<%!
// Code written here initializes instance variables
// so it is available in the page context.
  String instanceVariable = "Hello World!";
%>
<%
// Code written here is written to the executing method
// so it is available for the current request.
  int localVariable = 2;
%>
<html>
<head>
  <title>A simple JSP page</title>
</head>
<body>
<h1>This is a simple JSP page</h1>
<p>
Below is the contents of the instance variable:<br>
<%= instanceVariable %>
<%--
   This is a JSP comment.
   Below JSTL tag c:if is used to test a condition of a variable.
--%>
<c:if test="${localVariable == 1}">
  This text is written if localVariable equals 1.
</c:if>
</body>
</html>
```

Table 2: JSP page example

Based on my experience, the main advantage of JSP pages is that the page is more readable compared to a Servlet. With standard Servlets the contents of the HTML page needs to be written to response using a Writer that is called in between the source code. This leads to a situation where the resulting code is cluttered with println() method calls to the writer. Second advantage of JSP pages is the ability to split the JSP page into different sections and put each section to its own file. These parts can then be included to the main page into the desired positions or some sections can be skipped altogether. Usually the splitting is done so that there is a separate header and footer JSP template which are included to every page. This way the changes made in the header template are effective in all pages at once and there is no need to replicate the changes for each JSP file.

JSP tags are predefined tags recognized by the JSP compiler and they perform some specific task. A tag may be a logical branch tag, iterator tag or it may read and print out a value of a Java bean. The Java Server Pages Standard Tag Library (JSTL) provides tags for the most common tasks. This library is defined in the JSTL specification [17]. Using these tags reduces the need of embedded Java code on the page and makes the page more readable and maintainable.

It is possible to create custom JSP tags that perform some desired functionality. J2EE API defines a set of interfaces for this in package *javax.servlet.jsp.tagext*. By implementing one of these interfaces or extending one of the predefined base classes it is possible to create a new tag. The newly created tag has to be compiled and packaged to the web application and a tag definition file needs to be created. This definition file states the name and namespace of the tag after which it will be available on the JSP pages.

2.3. Web container

According to J2EE spec, "containers provide the runtime support for J2EE application components" [19]. A web container manages the execution of JSP pages and Servlets and other components deployed in the J2EE web application. Also it provides services that are defined in the J2EE specification ([19], chapter J2EE.2 Platform Overview). These services include the life cycle management, transaction management, concurrency, security and JNDI lookup services. The container will perform the lookup and binding of data sources, EJB remote references and JMS queues to the application context so that they are available to the executing components when they are initialized.

The container implementations are supplied by the application server provider. The developer needs to just implement the components that run inside the container and write an XML deployment descriptor file which registers the components in the application.

According to J2EE Tutorial ([20], chapter 11), the Servlet life cycle is controlled by the web container in which the Servlet has been deployed. When a request is mapped to a Servlet, the container performs the following steps. If an instance of the Servlet does not exist the container will load the Servlet class, creates a new instance of it, and initializes it by calling the init() method. When the Servlet class exists, the container will call its service() method passing it the Request and Response objects.

J2EE specification states that the server implementation must transparently support transactional runtime environment ([19], chapter J2EE.4 Transaction Management).

This means that when a component creates a JDBC connection to access the data store, all queries that it makes during the request processing are wrapped inside a transaction by a transaction manager. In case any of these queries fail, the underlying database structure will not be modified. All modifications made by the queries will be committed to the database only after the request processing has completed successfully.

2.4.Execution scopes

The web components need to share information with each others during the application life cycle. It is possible via objects that are maintained as attributes of four scope objects ([20], chapter 11). These scopes are application, session, request and page. Each of these scope objects have setAttribute() and getAttribute() methods that are used to access these shared objects. The difference in these scopes is the visibility and lifetime of the stored information ([2], chapter 10). The general idea about how these scopes relate to each other is presented in Figure 1.

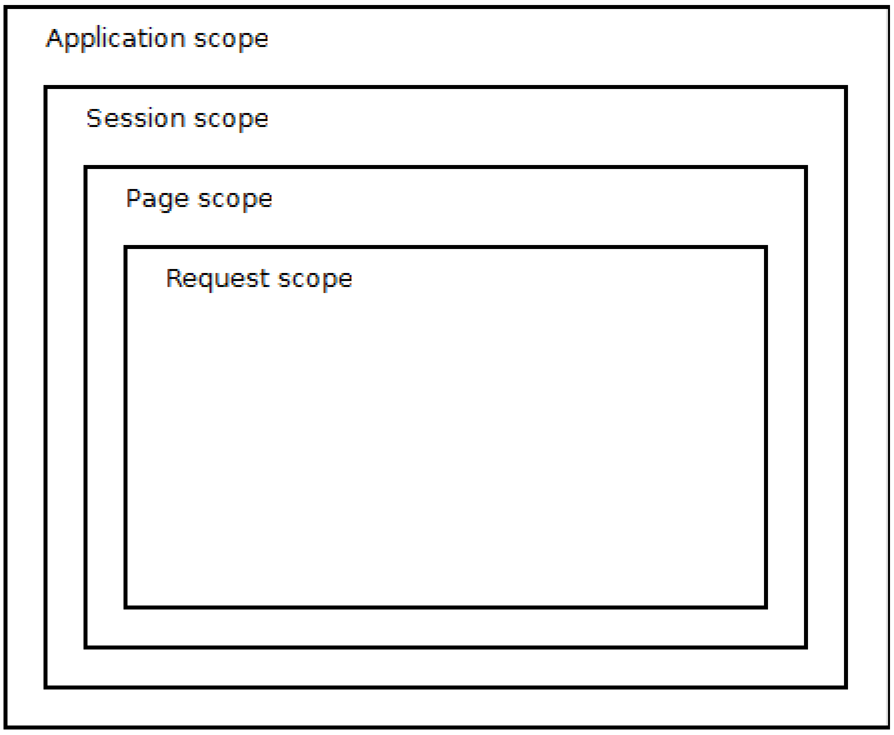


Figure 1: Shared object scopes

An application scope exists as long as the application executes. There is only one application scope in the web application and all information stored in there is seen by any component that executes in the same application.

Session scope is bound to the current request's session. There is one session for each client that accesses the application. Objects that are stored in session scope are only seen by requests that belong to the same session. Requests that belong to another session do not see that information and it is not possible to pass information between sessions directly. Information in the session scope exists as long as the session is valid. The information is destroyed when the session times out in the application or the application exits, though some application servers support persistent sessions where the information state is maintained even over server restarts.

Objects in the page scope are similar to instance variables of a Servlet. A page scope is associated with JSP pages and each page has its own scope. Information stored to the page scope is visible only inside the associated JSP page. The scope exists as long as the JSP page instance itself.

Request scope is associated with the currently executing request. The objects are stored to the request instance as attributes and they are available to all components along the path of the request execution. The request scope exists only for the current request and all objects stored in it are removed after the execution is finished.

3. WEB APPLICATION FRAMEWORKS

3.1.Motivation

According to DocForge ([5]), “a web application framework is a type of framework, or foundation, specifically designed to help developers build web applications. These frameworks typically provide core functionality common to most web applications, such as user session management, data persistence, and templating systems. By using an appropriate framework, a developer can often save a significant amount of time building a web site”. Based on my professional experience and the knowledge presented in the DocForge page (and its related pages), using a web application framework as the basis when developing an application has the following advantages.

The developer doesn't have to implement and test each functionality every time but can count on the components delivered by the framework that they are working and have been tested.

Implementing a web application using only JSP pages very soon leads to code that is hard to read and maintain. Accessing databases directly from the pages or using some other classes that implement some business logic means that the JSP page will be cluttered with Java code. Instead it makes sense to divide the business logic and view rendering layers from each others.

The web application framework has been developed by a large group of people. This has at least following advantages. The framework has gone through several stages of testing before it is released. Errors in the framework are found earlier since it is used by large amount of developers. Developers working with the same framework can help out each others in problematic situations even if they are not working in the same company or even on the same continent.

A web application will display dynamic content on the web pages. This information is stored in a database. A framework can assist in this task by providing an API that can be used to access the underlying storage in a controlled manner. It can validate the inserted data automatically so that all mandatory fields are filled and that the required relations between different tables exist. A framework could provide an automated mechanism that maps rows in the database tables to Java entity classes so that the developer does not need to write any or minimal amount of code that access the database. This sort of functionality is called ORM, Object Relational Mapping.

A framework provides some way to manage the security of the application. There can be a common way to set up user authentication and user management. A framework usually already implements the logic that handles user logins and developer only needs to create the login page itself. Also integrating the user management with some external system like LDAP directory is usually easy using the provided components. The application may have public pages that are available to everyone and there may be sections which require authentication. Detecting the need to direct the user to the authentication page is normally handled automatically by the framework.

A framework can have an effect on the application performance. The web page contents can be cached which increases the request throughput with most commonly accessed pages. The underlying database accesses can be cached and fewer database queries go all the way through to the database itself.

A framework supports a templating engine. Usually the framework supports some specific engine directly but can be set up to work with some other template engine as well if the developer desires to use another one.

Finally, the way the framework should be used usually encourages the developer to implement code that is reusable and easy to maintain. The business logic is clearly separated from the page templates.

3.2.MVC framework components

Model-view-controller is a common design pattern used with user interfaces in all kinds of applications. It is not restricted to web applications only. It enforces the developer to implement the application in a way that clearly separates the business logic of the application (in this case Java classes) from the view layer (the page templates). The business logic is backed up by the model which is a representation of the data that is stored in the system. Usually the data of the model is stored in a relational database. Information presented here is partly based on my professional knowledge and partly on [16], chapter 11.

Practically all web frameworks have been designed to be or include a part that is a model-view-controller. MVC pattern divides the whole application logic strictly into three parts: model, view and the controller. An MVC application is request driven component framework that maps the incoming request to a set of controllers. These controllers access the data storage which is called the model. The controller then gives this data to a template called the view. The view will render the final web page using the data given to it by the controller. The basic stages of the request processing is presented in Figure 2.

First stage in MVC application is to map the incoming request to the correct controller. There are several schemes how this is done. Usually a part of the request URI is mapped directly to some controller in an XML configuration file that is framework specific. The controller can forward the execution to another controller. This way the controllers can be chained together and each of them performs some small part of the processing. A controller may include some other controllers data to the request. This way the main controller will be in charge of which controllers to apply to the request.

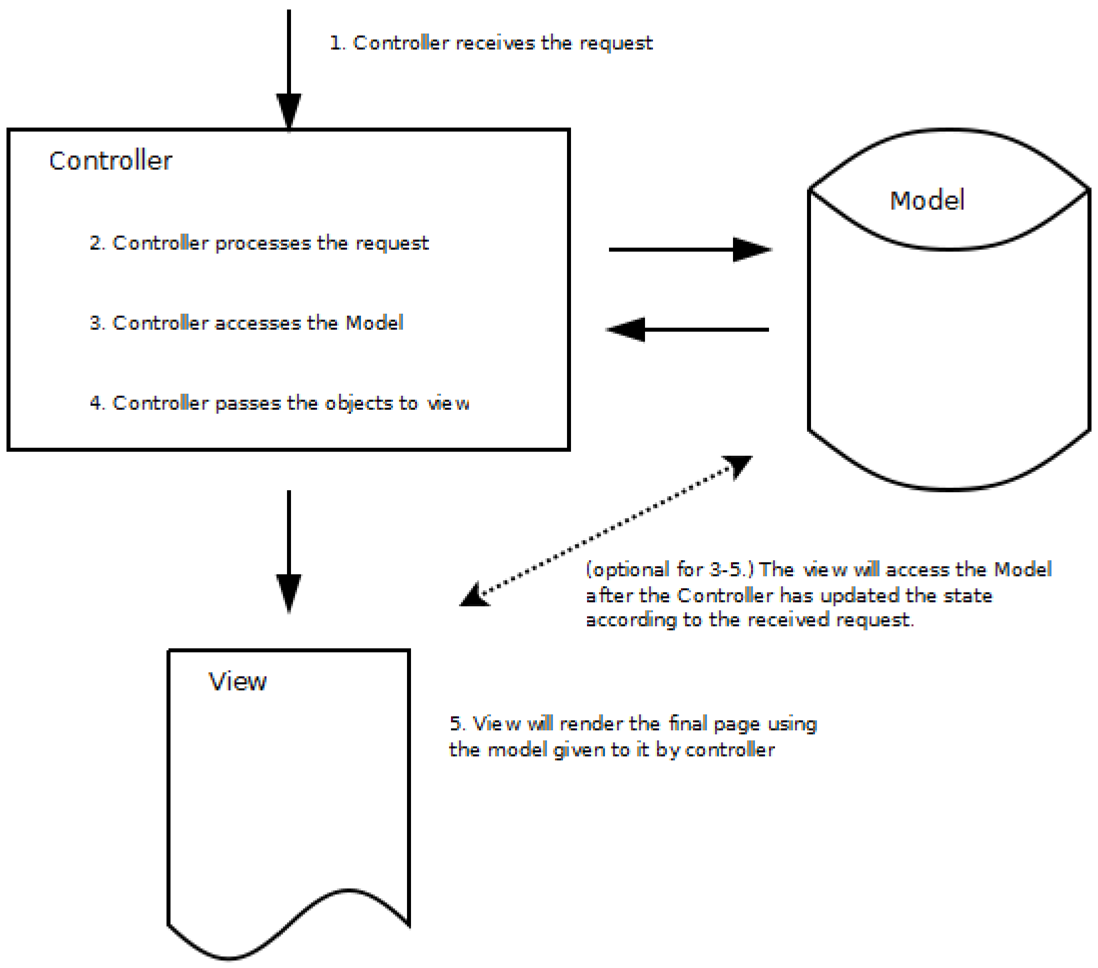

1. Controller receives the request
Controller
2. Controller processes the request
3. Controller accesses the Model
4. Controller passes the objects to view
Model
(optional for 3-5.) The view will access the Model after the Controller has updated the state according to the received request.
View
5. View will render the final page using the model given to it by controller

Figure 2: Basic stages of Model View Controller

The controller will read the information from the request and based on that information it will access the model. Usually the model is a relational database and the controller will read the data stored in the database tables and constructs objects that represent that data. These objects are stored to a map as key-value pairs. The key is a known predefined value that is recognized by the view and the type of the object that is stored as the value needs to match the type expected by the template. This map containing the model objects is added to the request scope.

```
// Example controller code based on Spring MVC.
public class ExampleController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        ModelAccessClass model = ModelAccessClass.getInstance();
        String parameterValue = request.getParameter("parameterName");
        List<MyEntityBean> results = model.list(parameterValue);
        ModelAndView mv = new ModelAndView("exampleView.jsp");
        mv.addObject("results", results);
        return mv;
    }

}
```

Table 3: Controller example

Finally the controller will forward the execution of the request to the view layer. The view is a template file (for example a JSP page) which will read the expected values from the request scope and writes the contents of the model objects onto the page. The template contains static content and the dynamic model objects are inserted between the static content.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
  <title>A simple JSP page</title>
</head>
<body>
<h1>This is a simple JSP page</h1>
<p>
Below are the results<br>
<table>
  <c:forEach var="myEntity" items="${results}">
    <tr>
      <td>Name:</td><td>${myEntity.name}</td>
      <td>Phone:</td><td>${myEntity.phone}</td>
      <td>Address:</td><td>${myEntity.addess}</td>
    </tr>
  </c:forEach>
</table>
</body>
</html>
```

Table 4: View example

Strictly speaking, as presented in [16] chapter 11, in MVC pattern, the View will access the Model after a Controller has updated the Model's state based on the request. But generally, as far as I have experienced, the workflow resembles more the one described in the above paragraphs.

3.3.Spring

Spring is an open-source J2EE application framework. It was created to address the complexity of enterprise application development. Spring is a lightweight inversion of control (IoC) and aspect oriented container framework. It is made up of seven modules that each provides a strict set of functionality ([21], chapter 1).

Core module defines how beans are created, configured and managed. Other modules build on top of the core module and these modules are: Application Context Module, AOP Module, JDBC and DAO Module, Object/relational Mapping Module, Web Module and The Spring MVC Framework. These modules provide everything that is needed to build J2EE applications on top of Spring.

Most of the Spring modules are left out of scope of this thesis but the Spring MVC is of most interest. As are the two principles of Spring framework, IoC and AOP.

In Spring it is possible to work with plain Java beans or POJOs (Plain Old Java Objects) where previously it was necessary to use EJBs and other enterprise Java specific objects. Spring objects are configured in XML files where the Java beans and their relationships are defined outside the Java source code. Developer just creates these basic beans in Java code and then their creation and relations are configured in the XML files ([21], chapter 2).

3.3.1. Inversion of Control

According to Rinat Abdullin[1], "Inversion of Control (IoC) is an approach in software development that favors removing sealed dependencies between classes in order to make code more simple and flexible. We push control of dependency creation outside of the class, while making this dependency explicit. Usage of Inversion of Control generally allows to create applications that are more flexible, unit-testable, simple and maintainable in the long run."

Quote from [21], chapter 1.4, "Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects".

In fact, Inversion of Control is more commonly called dependency injection. Both of these terms are used and they mean the same thing. Personally I have first heard the term dependency injection.

Based on my professional knowledge, IoC works basically as follows. Normally a class will create or find its dependencies by itself when it is created. This means that in the class constructor there is code that will create other classes or will look up a reference to some other class in the system. This leads to hardwired code where class names are hardcoded in the class constructor. Since the class in hardwired to some specific classes in the system, it makes it less portable and reusable. IoC will reverse this thinking. Instead of the class being responsible for creating and finding the dependencies, they are given to it when it is created.

```
public class TraditionalClass {
    private SomeService service;
    private SomeBean bean;

    public TraditionalClass() {
       service = SomeService.getInstance();
       bean = service.createNewBean();
    }
}

public class MyIoCClass {
    private SomeService service;
    private SomeBean bean;

    public MyIoCClass(){
    }

    public void setSomeService(SomeService s) {
        this.service = s;
    }

    public void setSomeBean(SomeBean b) {
        this.bean = b;
    }
}
```

Table 5: Example of Inversion of Control

In the code sample presented in Table 5, the basic idea of IoC is clearly demonstrated. The traditional class will create its dependencies by itself. It has to know where to look for the service and how to create the bean. In the second class the dependencies are injected in to the class with setter methods. The class itself doesn't know anything about where the service and the bean instances come from. This is now the responsibility of the external manager that resolves these dependencies.

In Spring these dependencies are configured in XML files. Below is an example which creates the classes introduced in Table 5 (based on an example in [11], chapter 3).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="someBean" class="com.some.company.SomeBean" />

  <bean id="someService" class="com.some.company.SomeService" />

  <bean id="myIoCBean" class="com.some.company.MyIoCClass">
    <property name="someService">
        <idref bean="someService" />
    </property>
    <property name="someBean">
        <idref bean="someBean" />
    </property>
  </bean>

</beans>
```

Table 6: Example of XML-based bean configuration

3.3.2. Aspect-oriented Programming

Quote from [21], chapter 1: “Aspect-oriented programming (AOP) is often defined as a programming technique that promotes separation of concerns within a software system. Systems are composed of several components, each responsible for a specific piece of functionality. Often these components also carry additional responsibilities beyond their core functionality such as logging, transaction management and security”.

As a result, the component contains a lot of code that is related to managing these responsibilities and which is not part of the component's core business logic. This leads to two problems on the code level: the code that implements these additional tasks is replicated throughout the application. Each component needs to call the logging component to print a message to system logs by themselves. If the logging component changes, the changes propagate to every place where the logging component is called. Second problem is the fact that the component contains a lot of code that doesn't belong there in the first place.

Based on an information found in [21], chapter 1, aspect-oriented programming addresses this problem by modularizing these additional services and then applying them to the components that need them. The AOP services that wrap the application component will intercept the method calls to the component. They will perform their

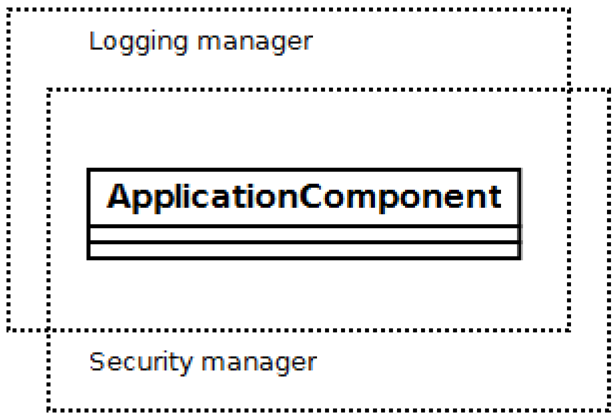


Figure 3: Application component wrapped with AOP services

specific tasks before or after the application component's method is executed. The presence of these AOP services is completely transparent to the application component. It will perform its functions as before without knowing that these AOP services are involved in the process. In Figure 3 the situation is presented when an application component is wrapped with a logger and security manager AOP services.

By wrapping all components with the needed AOP services it is easy to change the functionality of some service. The changes are automatically effective in every part of the application since the same service is used with all components. There is no need to modify the actual component in any way.

3.4.Hibernate

Hibernate is an object-relational mapping (ORM) library for Java applications. Hibernate project was started in 2001 by Gavin King [8]. Hibernate's main feature is mapping Java POJO classes to database tables and from Java data types to SQL data types. Mapping Java classes to database tables is done using XML configuration files and Java annotations [22]. Hibernate also provides a query language (HQL) that resembles SQL but is fully object-oriented. As of 2010 Hibernate 3 is a certified implementation of the Java Persistence API (JPA) specification [22]. In addition to the features defined in JPA, Hibernate provides an nonstandard set of features and extensions through annotations that are specific to Hibernate [8].

Following information is based on my personal experience and thoughts. Storing an object to a relational database table in the traditional way requires that the data from the object's fields is read and inserted to a database table using an SQL query. When reading an object from the database the opposite happens: the object is created based on the data read from an SQL query result. The resulting code contains functionality that is replicated in many places. Transforming primitive Java types to SQL data types in the source code easily leads to run-time errors due to insufficient type checking or incorrect conversion. And because these errors occur only in run-time, they are often fatal to the system stability. ORM solves all of these problems.

Because the application source code does not contain the direct SQL language anymore, changing the underlying database engine has no effect on the application code. Usually different SQL engines may in some circumstances require some tweaks in the executed queries. These small changes (like hints and such) are specific for the one SQL engine and are not supported in another. ORM resolves this problem by implementing support for several common SQL engines. The developer just needs to configure the ORM library to match the underlying database and the application code will run without any changes.

3.4.1.Object-relational mapping

Following information is based on my personal experience with JPA and Hibernate and partly the information and samples obtained from [13], chapter 1 and [3], chapter 1.

Object-relational mapping basically means that an object can be stored to a relational database without writing and filling the values of an SQL query. Correspondingly the read operation does not necessarily require an SQL query and the resulting object instance with the correct values is created and returned by the ORM library. An object instance that is mapped to set of data in the database is called an entity. In order that the mapping would work, the ORM library needs to know how to map the Java classes to the correct database tables. This is achieved with either an XML configuration file or direct annotations in the Java source code. An example of an XML configuration is shown in Table 7. The same can be achieved with annotating the source code directly. An example of an annotated Java class is shown in Table 8.

```xml
<!-- Example based on Hibernate Reference Documentation -->
<hibernate-mapping package="com.some.company">
    <class name="MyItem" table="ITEM">
        <id name="id" column="ITEM_ID">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <property name="description"/>
    </class>
</hibernate-mapping>
```

Table 7: Sample Hibernate XML mapping

```java
/* Example based on Hibernate Annotations Reference Guide */
package com.some.company;

import javax.persistence.*;

@Entity
@Table(name=”ITEM”)
public class MyItem {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String description;

    public MyItem() {}

    // public accessor methods for name and description
    // private setter for id
}
```

Table 8: Sample entity class with JPA annotations

In Hibernate these entities are managed through a *Session* interface. The corresponding interface in JPA is *EntityManager*. The *Session* interface provides methods for storing, updating, reading and removing entities to and from the database. Searching the database for entities can be performed in several ways. A session provides a simple *get()* method for querying a single entity. Second option is to use *Criteria* objects which perform the search and restrict the result set as desired. Third option is to use a HQL query and finally it is possible to use a plain SQL query. Examples of entity queries are shown in Table 9.

```java
// get an item with id 2
Item anItem = session.get(MyItem.class, 2);

// list items that have a description starting with White
List items = session.createCriteria(MyItem.class)
    .add( Restrictions.like("description", "White%") );

// same with HQL
Item anItem = session.createQuery(
    “from Item i where i.description like 'White%' ”).list();
```

Table 9: Example Hibernate queries

3.5.Ajax

Ajax is shorthand for Asynchronous Javascript And XML. The term Ajax was first introduced in 2005 by Jesse James Garrett[7]. Normally in a web application model the client browser issues a HTTP request to the server and starts to wait for the server to respond. Server starts to process the received request and after a while it will send the data back to the client in a response. During the time of request processing added with time taken in the data transmission the client browser is blocked and only displays an indication to the user that the requested page is loading.

An Ajax application eliminates the start-stop-start-stop nature of interaction on the Web by introducing an intermediary — an Ajax engine — between the user and the server. Instead of loading a web page, at the start of the session, the browser loads an Ajax engine — written in JavaScript and usually tucked away in a hidden frame. This engine is responsible for both rendering the interface the user sees and communicating with the server on the user's behalf. The Ajax engine allows the user's interaction with the application to happen asynchronously — independent of communication with the server[7].

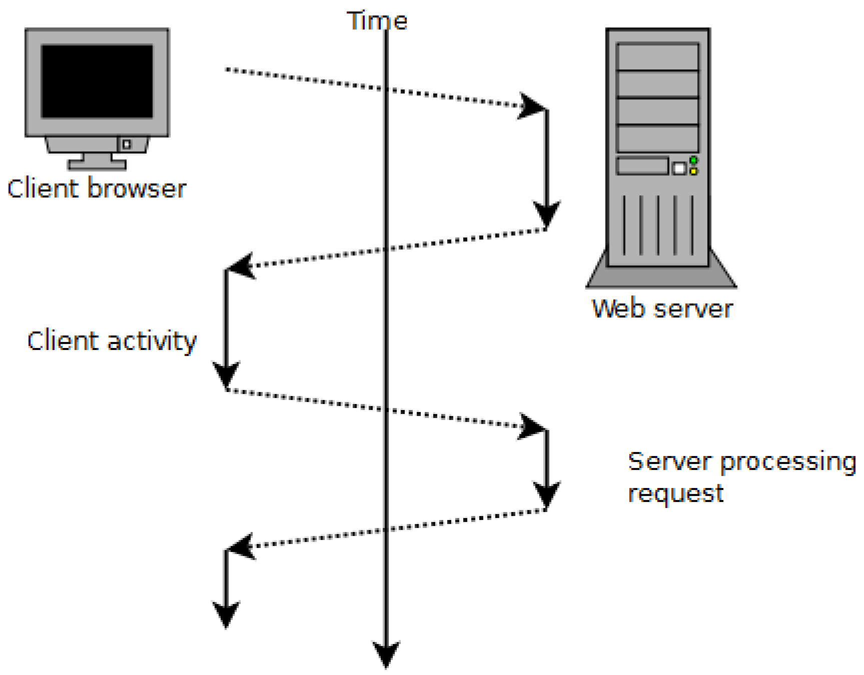


Figure 4: Classic web application model

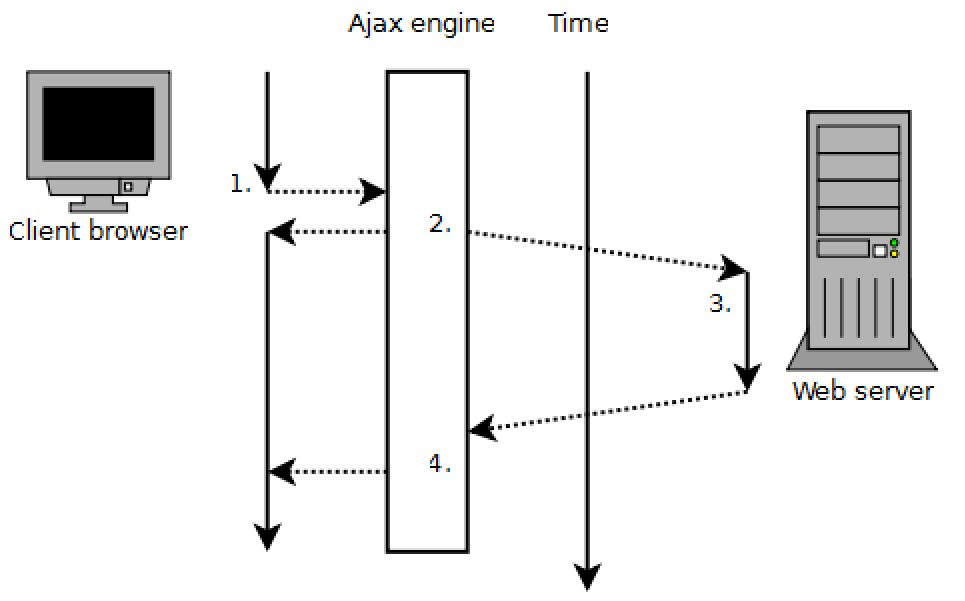
Following is based on my personal experience. The images were inspired by the article in [7].

In a classic web application model (synchronous model) the client browser will send a HTTP request and starts to wait for the response. This is illustrated in Figure 4. The client is not able to perform any other activity while waiting for the response. The whole web page (or frame) is redrawn with each request-response cycle. When the web page is complex and contains many different sections each containing a set of information, this means that the server will have to process all of those sections for each request. This will lead in increased server load. The server will have to reprocess each section of the page even if it is not necessary.

An example of this could be a web store application. The sections on the page in this example are the navigation tree containing the product categories, the user's shopping cart and the list of items in the selected category. User selects a link on the product list in order to view some detailed information about the product. The request is then processed by the server. It has to read the categories from the database (or from memory if they are cached) and print the navigation tree on the page. Next it has to fetch the user's shopping cart from the session scope and print the contents of the cart. Finally it will read the product information from the storage and prints the information of the product on the page on the same position that previously contained the product listing.

In fact two out of three steps illustrated above are unnecessary. The position in the navigation tree has not changed in any way. The contents in the navigation section of the page are the same as before. Also, the user has not added anything in to the shopping cart so the contents of the cart have not changed. All the work that has been done to print these sections on the page is redundant.

In an Ajax web application model (asynchronous model) clicking on an element on the page issues a Javascript event which is processed by the Ajax engine. The engine will construct a HTTP request and sends it to the server in the background. This is illustrated in Figure 5. The control is returned to the browser page practically immediately and this will go unnoticed by the end user. The Ajax engine will bind a callback function on the HTTP request so that when the server returns the response,



1. User clicks on an element on the web page.
   An event is received by Ajax engine.
2. Request is sent by Ajax engine to the server.
   Control is returned to the browser while request is being processed.
3. Server processes the request.
4. Ajax engine receives the response. A part of the web page is updated.

Figure 5: Ajax web application model

that callback method is executed. The function call contains the response contents as a parameter. In the callback function a part of the current web page is replaced with the contents of the response. Rest of the current page is left untouched.

Referring to the previous web shop example, in this case only the section that previously contained the product listing is now replaced with the details of the selected product. The web server had to only perform the last step out of three in the previous example. This will reduce the server load significantly when the number of simultaneous requests increases. Second advantage in this model is that the user interface feels more responsive to the end user. Each click on the web page will not issue a complete refresh of the page and the user does not need to wait for the response. Instead, the user is now free to click some other element on the page which would then trigger a new Ajax call and would be processed by the Ajax engine concurrently with the other ones. For example the user could first select an element to view some product details and while that request is still being processed, in the meantime could empty the shopping cart. Both of these requests would update their respective sections on the page after the response is received independently of each other.

3.5.1. An Ajax example

In practice the updating of different sections of the web page is usually based on the id-attributes of the related HTML elements or the relational position of the elements in the document object model (DOM) tree.

```
<div id=”mainContent”>
  <div id=”productList”>
    <ul>
      <li onclick=”showItem(1);”>item1</li>
      <li onclick=”showItem(2);”>item2</li>
      <li onclick=”showItem(3);”>item3</li>
    </ul>
  </div>
</div>
```

Table 10: Product listing section

The product listing HTML source for the example that was discussed above is listed in Table 10. The page contains a div-element with an id attribute *mainContent*. The elements in the product list have an onclick event handler and clicking on one of the elements will call the *showItem()* function.

```
function showItem(id) {
  new Ajax.Updater('mainContent', '/path/to/showProduct?id='+id, { method: 'get' });
}
```

Table 11: Ajax function

The source of the *showItem()* function is listed in Table 11. In this example a Javascript library Prototype[14] is being used. There are many other Ajax-enabled Javascript libraries but Prototype was selected just as an example. All the *showItem()* function does is that it defines which HTML element to update (here it's the element with id *mainContent*) and an URI where the request should be sent to. The Ajax engine will send the request to the defined address and after receiving the response, updates the contents of the defined element.

```
<div id=”productDetails”>
    <ul>
    <li>item name</li>
    <li>item description</li>
    <li>item price</li>
  </ul>
</div>
```

Table 12: Ajax response

The contents of the Ajax response is listed in Table 12. The response contains only the part of the page that goes inside the element that is being updated.

3.6.Java Server Faces

As defined in JSR 314, “JavaServer Faces (JSF) is a user interface (UI) framework for Java web applications. It is designed to significantly ease the burden of writing and maintaining applications that run on a Java application server and render their UIs back to a target client” ([4], chapter 1).

JSF is unlike other web frameworks because it's not a library, but rather a specification for a library. It's developed in the Java Community Process (JCP) and included in the Java EE platform of specifications. Currently, it's being worked on as Java Specification Request (JSR) 314, though the foundation was laid for JSF in JSR 127 and later improved on in JSR 252[10].

Compared to other common web UI frameworks, JSF is an event driven model. Developing a JSF application resembles more the traditional application UI development than the request-response model used with other frameworks. The JSF components emit events based on user actions. These events are processed by components that have registered themselves as the event listeners. JSF contains a set of basic user interface components and it allows the creation of new components by application developers.

3.6.1.Request life cycle

Following is based on the information in [9], chapter 1 and [4], chapter 2.

The request lifecycle in a JSF application differs from the usual request-response life cycle seen with common web applications. The JSF specification defines six distinct phases in the life cycle. These phases are shown in Figure 6.

Restore View phase retrieves the component tree for the requested page. If they do not exist, they are created. If the page has been requested before by the same client, all components are set to their prior state. This way JSF application will retain information of forms and other web components. If the request has no query data, the execution will skip directly to *Render Request* phase. Otherwise, *Apply Request*

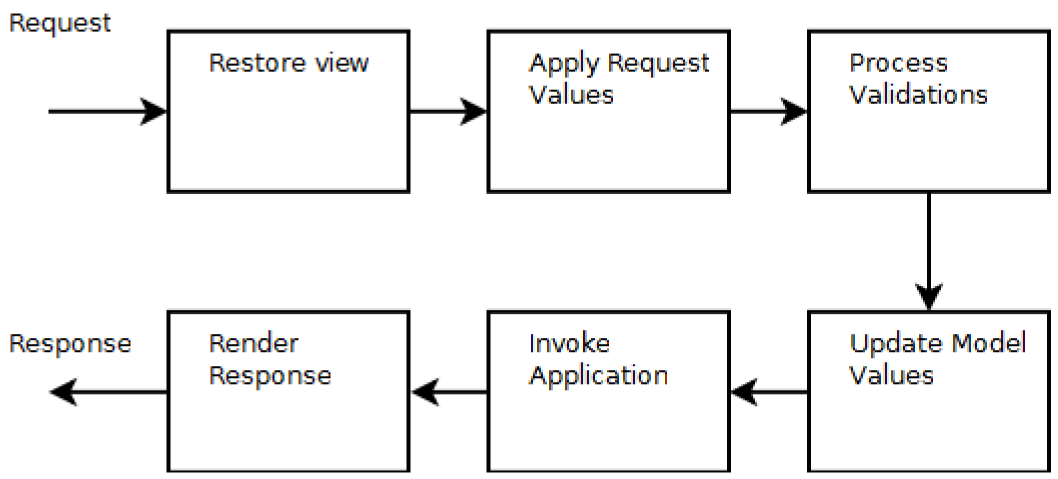


Figure 6: JSF phases

Values phase is executed. Here the request data values are stored to the components where they belong. In *Process Validations* phase the previously assigned request values are validated and converted. The validation phase uses converters to convert the incoming string values to the correct field values and validators to check the sanity of desired fields. For example the validators could check if an email field or a phone number contains a valid value. If validation fails the execution will skip to *Render Response* phase. In *Update Model Values* the converted and validated values are set in the beans that are wired to the components. Next the *Invoke Application* phase will execute the core business logic of the application. This is where the real work is performed. The corresponding components *action()* method is called. Finally the response is rendered in the *Render Response* phase by a JSP page.

3.6.2.Managed beans

All data that is accessible for from the page is done through beans. Beans are the bridge between the display layer and the business logic. All data that is printed on the page is read from a bean and all data that is sent back to the server is bound to a bean. These beans are connected to the JSF application components. The beans are defined in *faces-config.xml* file. When calling the value of such bean on the page using a page component, the bean instance is automatically injected (and created if doesn't exist) to the component. An example of reading and writing the value of a bean is presented in Table 13 ([9], chapter 2).

```
<!-- This will print out the value from user.firstName property -->
<h:outputText value="#{user.firstName}"/>

<!--
  This example prints a HTML input field where the initial value is
  read from the user.firstName property and stored to that same property
  when the form is submitted
-->
<h:inputText value="#{user.firstName}"/>
```

Table 13: Manipulating a bean property

The binding of a submitted value is automatically performed by the JSF framework during the *Update Model Values* phase.

3.6.3.Navigation rules

JSF navigation rules are specified in the *faces-config.xml* file. This file defines the view names and the transitions between different views. An example of the file is shown in Table 14 ([9], chapter 3).

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
<faces-config xmlns="http://java.sun.com/JSF/Configuration">

  <navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>Success</from-outcome>
      <to-view-id>/welcome.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>Error</from-outcome>
      <to-view-id>/login-fail.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/welcome.jsp</from-view-id>
    <navigation-case>
      <from-outcome>Logout</from-outcome>
      <to-view-id>/login.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  …
  <managed-bean>
    <managed-bean-name>user</managed-bean-name/>
    <managed-bean-class>com.some.company.User</managed-bean-class>
  </managed-bean>
</faces-config>
```

Table 14: Sample navigation rules

Here is shown a simple mapping between the login page and the welcome page. As can be seen, the configuration file for just this small example is quite long and editing the configuration manually becomes complicated soon after only a few views are added.

3.7.Seam

Seam is a powerful open source development platform for building rich Internet applications in Java. Seam is developed by the JBoss community. “Seam integrates technologies such as Asynchronous JavaScript and XML (AJAX), JavaServer Faces (JSF), Java Persistence (JPA), Enterprise Java Beans (EJB 3.0) and Business Process Management (BPM) into a unified full-stack solution, complete with sophisticated tooling” [15]. It is another web application framework that combines several other frameworks together and it attempts to simplify the development of a JSF application.

3.7.1.Entity beans as managed beans

Seam makes it possible to use EJB3 beans directly as managed beans on the pages. With plain JSF it was necessary to create the code that glues the EJB3 persistent beans to the backing beans since the managed beans are configured in the *faces-config.xml* file and created by the JSF framework. That is why they can not be directly the entity bean instances. With Seam the EJB3 bean class can be annotated so that it is recognized by the Seam framework and it is available on the page without the additional wrapper code. A sample entity bean with Seam annotation is presented in Table 15 ([6], chapter 3).

```
@Entity
@Table(name=”USERS”)
@Name(“user”)
public class User implements Serializable {
    private String username;
    private String password;

    public void setUsername(String s) { username = s; }
    public String getUsername(){ return username; }
    public void setPassword(String s) { password = s; }
    public String getPassword(){ return password; }
}
```

Table 15: Sample Seam entity bean.

3.7.2.Conversations

In normal Java web application there are five different scopes available as described in Chapter 2.4. Seam adds a new scope called *Conversation* scope. A conversation will group together several requests in a session. Each session may have several concurrent conversations running, and a conversions can be nested inside each others. This enables the web store application to have for example two different shopping carts and processes for the same user. The user may switch between the carts independently of each other and the state of the cart is maintained while user is managing the other. The relation between the session and requests and conversations is shown in Figure 7 ([6], chapter 4).

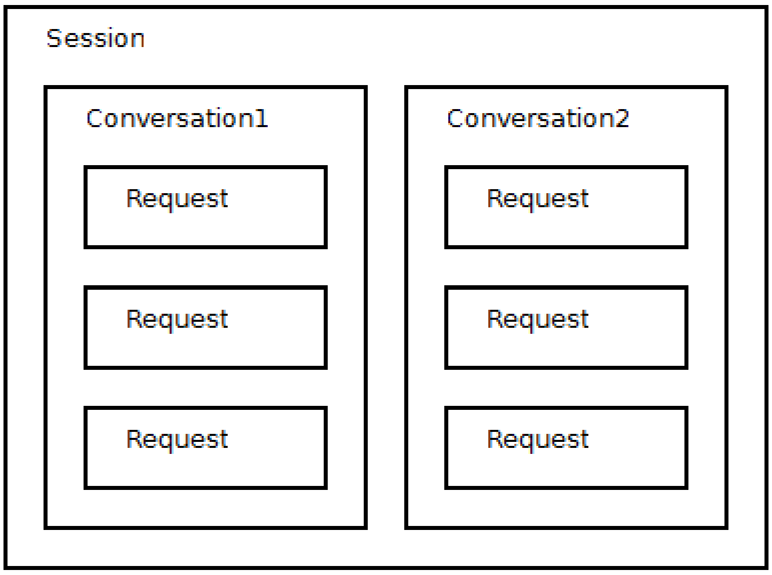


Figure 7: Relation between session, conversation and request

3.7.3.Bijection

Several frameworks (like Spring) make use of a concept called dependency injection. It automatically inserts the property values of a bean. Seam's bijection model expands on this basic injection concept in a few ways. It supports the normal injection as well as what Seam calls an *outjection*. This means that a component may export its value out to the surrounding context and insert to another bean. Second difference is that usually the injection happens only once, before the component is invoked. In Seam the bijection is done on each invocation of the component. This allows the components to be stateful and their state can change during the process. A bijection sample is displayed in Table 16 ([6], chapter 3).

```
@Stateless
@Name(“userBean”)
public class CustomerBean implements ICustomerBean {

    @In(value=”Customer”, required=false)
    @Out(value=”#{order.customer}”, required=false)
    private Customer customer;

    newCustomer(){
        …
    }
}
```

Table 16: Bijection example

In the above example the EJB method *newCustomer()* creates a new customer instance. After it is created the value of the newly created customer is outjected as a value in order bean's customer property.

4. CONTENT MANAGER

4.1.Background

Ambientia Content Manager is an in-house developed CMS application. It has been actively developed for over 8 years while the basic technology on which it is based has not changed. New techniques have been introduced to some parts of the application (like Hibernate) during these years, but on the whole the basic technology has remained. It has been implemented using Java Servlets with Apache Velocity and JSP page templates. The application has not been completely rewritten during its lifetime. Instead, a new version has always been developed from the previous one in an iterative process. This application is offered to clients as a licensed application and it comes as a part of the deal including support and maintenance contracts. There is always a customization project involved and as a result the client gets a modified version of the application which contains the desired additional functionality. Due to the customizations made in project teams to the main product, a later upgrade to new CM version may not be a drop-in replacement but requires a new project altogether.

New stable version of the application is released internally after a development cycle. Major releases are scheduled once every one or two years and minor patch releases are introduced as required.

4.2.Current design

Content Manager contains two main Servlets. Administration Servlet is called the Core and public web-page Servlet is called the FrontEnd. The basic functionality is the same for both of these Servlets. They manage the authorization, navigation and data model and pass the execution to the Velocity templates which render the final web page.

Content Manager Servlets handle the basic request processing. The main business functionality is implemented using customizable modules. Each page in the navigation is handled by some module. For example there is a module for basic single column page (where the text contents of the page is displayed in one column) and another for a page with multiple columns. Form Designer module is used to design simple web-forms and it handles the submitted data of these forms. Bulletin Manager maintains a set of bulletin areas which contain a number of bulletin messages.

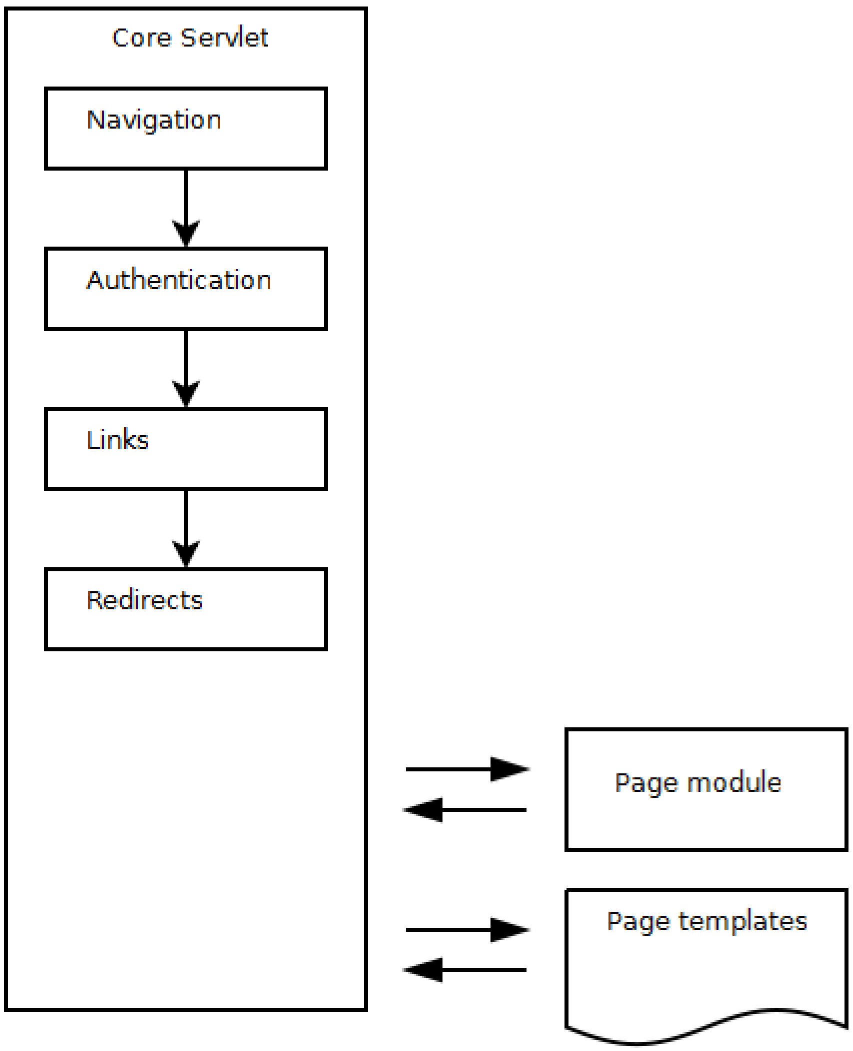


Figure 8: Content Manager core functionality

All these modules can be enabled or disabled from the system using the administration application. Each module provides its own administration views and public views. A module basically reads information from the database according to request parameters and inserts the resulting objects into request context with specific keys. These objects are then available to the Velocity templates that render the resulting page.

Most modules in Content Manager implement a CommandModule interface. This is a module framework that is used as a base for several built-in and custom modules. Basic idea is that the request contains a parameter named *command*. This parameter is mapped to a command class that is managed by the module. This allows the module to have all of the commands cleanly implemented in separate classes. This way the size of the actual module class is minimized. The only code required is the part where all available commands are registered to the CommandModule base class.

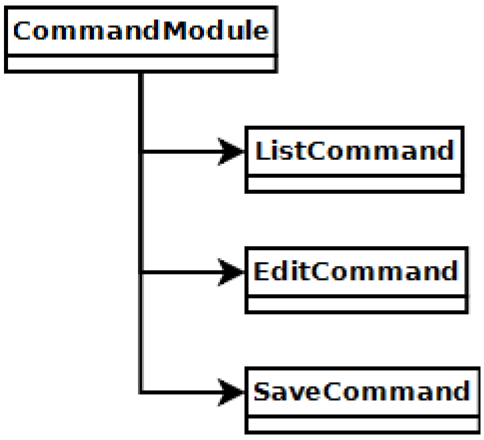


Figure 9: CommandModule structure

After the module has inserted the required objects to the request context the request execution passed to the module specific Velocity template. All page templates use a common layout template which includes a header and a footer sections on the final page. The module specific template inserts its output to the middle of the page to a predefined position.

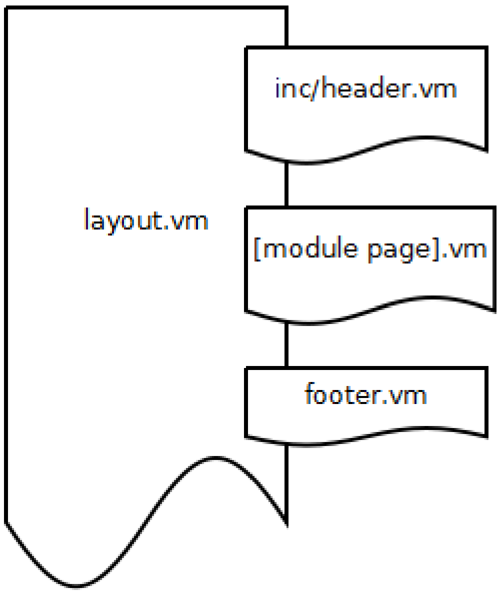


Figure 10: Velocity page template structure

4.3.Problems with current design

Current Content Manager is difficult to maintain and develop since the code it is based on is old. The system has never been completely rewritten and new features have been added on top of old functionality. The two main Servlets handle many tasks from authorization to navigation handling among other request preprosessing before the execution is passed to the modules that implement the page specific tasks. Second major drawback in the current system is the old database management code. Most of it is concentrated in one DataProvider class which executes all database queries and returns their results. The database accessing code is written specifically for MySQL database. Moving to a different database engine is quite impossible since the amount of code in the DataProvider alone is enormous. The DataProvider class has over 3000 lines of code in tens of methods. The whole product is heavily based on this one class and its functionality.

Some of the modules, especially newer ones, use Hibernate to abstract the database procedures from the business logic and therefore they are database engine independent.

In each customer project there is a need to customize the base product. These changes include the page templates, customizing an existing module or creating a completely new module.

The lack of general interfaces in the core of the product leads in to a situation where it is necessary to modify the core directly in customization projects. As a result each customization project and the resulting system is strictly bound to a specific version of the product. Upgrading the base product is not possible without setting up a whole new upgrade project. Depending on the amount of features and the level of customizations done previously, it may be too expensive for the client to upgrade. From business point of view, this may lead to short spanned client relationships with several of the small customers. Bigger clients have larger budgets and may well accept the cost of upgrade.

Since the customized code is bound to a specific version of the base product, the code is not portable. Custom code developed for some client may not fit directly to some other project even if the base product versions are the same. Time is wasted when making the same changes to the base product over and over again. Project throughput is lower when compared to a situation where the code could be reused directly across different projects.

5. USING SEAM WITH CONTENT MANAGER

5.1.Scope of the thesis

Work done in this thesis consists of design and implementation of functionality listed below. The result is a working service which will not be ready for use in production systems, but will be a proof of concept and it will be evaluated whether it is a viable alternative for further development.

The first part of the work is to create the components necessary to enable category navigation within a Seam application. The category hierarchy is managed and maintained by Content Manager administration application and resides in a database. Seam application needs to read that information from the database and print it on the displayed page. The navigation needs to be fully functional i.e. the links in the navigation tree need to be live links rendered by the service and they should reflect the state of the hierarchy in the administration application. Externally the links generated by the service should reflect the navigation path inside the service. The links should contain all parts of the path in clearly readable format just like in the existing system.

The second part of the work is to use some pre-existing code from Content Manager in the new Seam application. Here a Bulletin Manager 2 (BM2) module was selected because it is quite new, well packaged and uses Hibernate object-relational-mapping technology for the persistence functions. The Seam application needs to list the headlines of BM2 messages on the page. From the displayed list it should be possible to select any message and the contents of that message should then be displayed on the page.

The third part of the work is to include Ajax features on the page so that the message list is continuously displayed on the page while different messages can be selected and viewed on the other part of the page. The message should be loaded on the display area asynchronously using the Ajax features provided by Seam and related components.

Fourth part is to implement a tag with given tools (Facelets). The tag should implement some generic task. The tag code must not have relations to any external components but has to be reusable and portable. Here the specific task is to render HTML Meta tag elements to the resulting page. The tag will print out the name and value of the given tag attributes or key-value pairs of a given map.

Fifth part is to come up with a solution so that these components can be used easily in the development process of a customer project. There should be no need for any low level coding as the developer might not be familiar with all the technologies involved in the process. Here the solution was to use Eclipse IDE snippets definitions to include specific code sections with configurable attributes on the resulting page. All the developer needs to know is what the section does and the possible values of its attributes.

5.2.Development environment

The development tools and application servers that were used in the project were:

- JBoss AS 4.2.2 application server
- Seam 2.0.0 JSF framework
- Elipse Europa version 3.3.1.1 integrated development environment
- JBoss Tools plugins for Eclipse IDE

5.3.Implemented components

The Seam application will resolve the correct navigation category based on the information contained in the request. The matching navigation path is searched from the database category structure and the request is forwarded to a page which will render the contents of that category. On the page the category structure is printed as a list of links that represent the navigation tree in the system. The bulletin message that match the selected category is searched from the database and the contents of the message are displayed on the page after the navigation tree. The components that implement this set of functionality are introduced in this section. The detailed inner workings of the system is explained in the next section.

net.ambientia.seamspike.webapp.SeamPathFilter

SeamPathFilter is responsible for translating the external URI representation to internal URI and forward the execution to Seam framework. The filter will search for a category in the navigation database that matches the requested URI path and based on that category the execution is forwarded to the correct page.

This filter has similar methods as a standard Servlet filter (as defined in javax.servlet.Filter interface) but is not managed by the underlying Servlet container. It is created and managed by Seam framework. The filter is not configured in standard web.xml definition file, but the configuration is given as annotations in the class source file. Seam recognizes these annotations and loads the filter as part of the filter chain at application startup.

If the filter was implemented as a standard ServletFilter Hibernate could not be used since the Hibernate transaction does not exist at that stage.

net.ambientia.seamspike.webapp.NavigationBean

NavigationBean handles the loading of navigation hierarchy tree from the database using Hibernate. This bean is used to fetch the root category from the service and using that root it is possible to print out the whole category tree on to the page. This bean will return an instance of CategoryBean. The NavigationBean will cache the contents of the database for one minute so changes made to the navigation structure with the administration application are delayed up to one minute until they are visible in the Seam application.

One bean instance is initialized at startup by Seam and it will reside in APPLICATON scope throughout the whole application life cycle.

net.ambientia.seamspike.webapp.CategoryBean

This is a wrapper class for CMCategory entity bean. This bean will provide methods for creating external navigation link paths that are displayed on the page. This class is used by the SeamPathFilter for resolving the correct path where the execution should be forwarded. All bean instances hold a reference to their parent category (except root which has no parent) and to all their child categories thus enabling category traversal to all directions from any category.

net.ambientia.seamspike.entity.CMCategory
This is an entity bean which represents one entry in the navigation database table. Here only the relevant columns from the database table are gathered as values of the bean. Only a read-only access is provided for the fields so all modifications are performed through the administration application and write operations are denied.

net.ambientia.seamspike.session.BulletinMessageList
This is a Seam specific manager bean class which is responsible for listing Bulletin Manager 2 messages from the persistent storage. The list of messages is searched with EJBQL query. A CategoryBean instance is injected into this bean by Seam and the resulting message list is narrowed with the id of that bean. Only messages that are related to a specific category are returned.

net.ambientia.seamspike.session.BulletinMessageHome
A Seam specific manager bean that will fetch one Message entity bean instance from the database. This class is used when displaying the contents of a message on the page. ID of the selected message is injected into this class by Seam from a request parameter.

net.ambientia.seamspike.webapp.tag.MetaTag
This tag will print out HTML META elements given a map of key-value pairs or two attributes where the first one is the key and second is the value.

5.4.Functional description

This section will describe the different stages when processing a request in the service. The request processing sequence is illustrated in Figure 11. Here only the components relevant to this project are included and the inner workings of Seam, JSF and Hibernate are described only on general level. The figure illustrates a request when the request does not contain a message ID but only lists the navigation tree and message headlines on to the page.

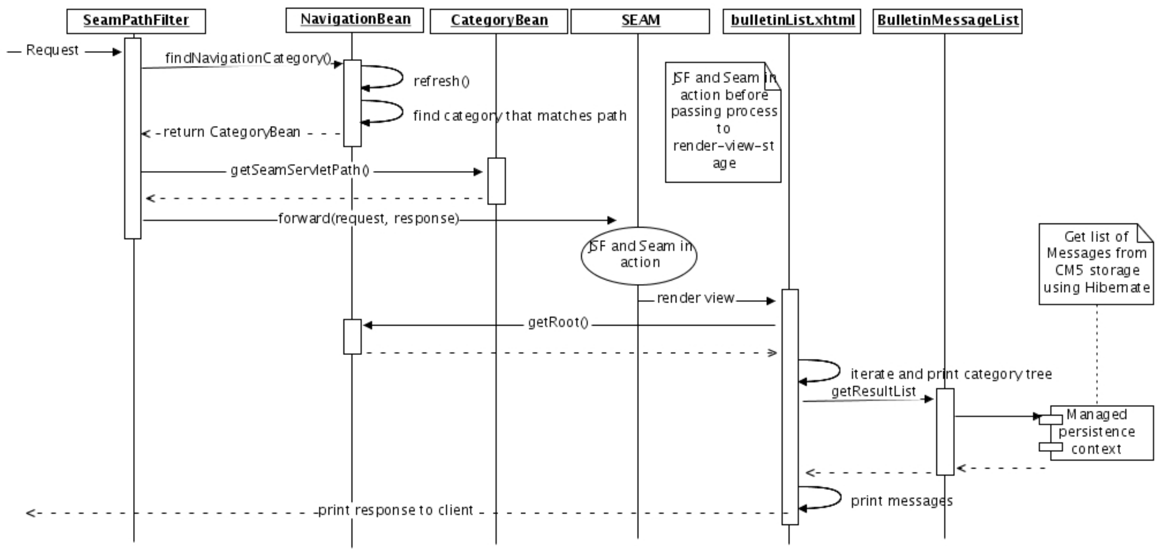


Figure 11: Sequence diagram of request processing

SeamPathFilter
This filter will receive a request that contains a request URI of form /path/to/category. This is called the external navigation path. The filter will search for a matching category from the database and it is inserted to REQUEST context. The category is a CategoryBean instance. The request URI is then modified by the CategoryBean instance so that the request processing is forwarded to the correct page. In this case all requests are forwarded to URI /bulletinList.seam which will make Seam execute a page called bulletinList.xhtml. It is possible to execute different pages for different types of categories by changing the functionality in CategoryBean, but here it is sufficient to forward all requests to the aforementioned page.

Session beans
Session beans extend the classes provided by Seam framework and most of the functionality is handled by the super classes. These bean override only appropriate methods in order to gain control of the returned object types. Some fields in these beans take advantage of the injection properties of Seam framework where Seam will inject a value of a request parameter or instance of bean directly as a value of an instance variable. This functionality is controlled in the class source files with annotations.

Database
This application uses directly the database of ContentManager to read the navigation

hierarcy and Bulletin Manager 2 messages. Care has been taken to provide only read-only access to the database since all modifications are made with the administration application. Also due to the structure of category table in the database, there are columns that are not relevant to this project and are not read.

The database connection is configured in JBoss application server and the connection is wrapped as a Datasource. This is all done by the application server. Then Hibernate is configured to use that data source. Also Hibernate is pointed to use the entity bean configuration found ready in Bulletin Manager 2 libraries. Hibernate is said to have a persistence unit after these are set. Seam needs an EntityManager that uses an EntityManagerFactory that uses the above persistence unit configured in Hibernate. The above process is illustrated in Figure 12.

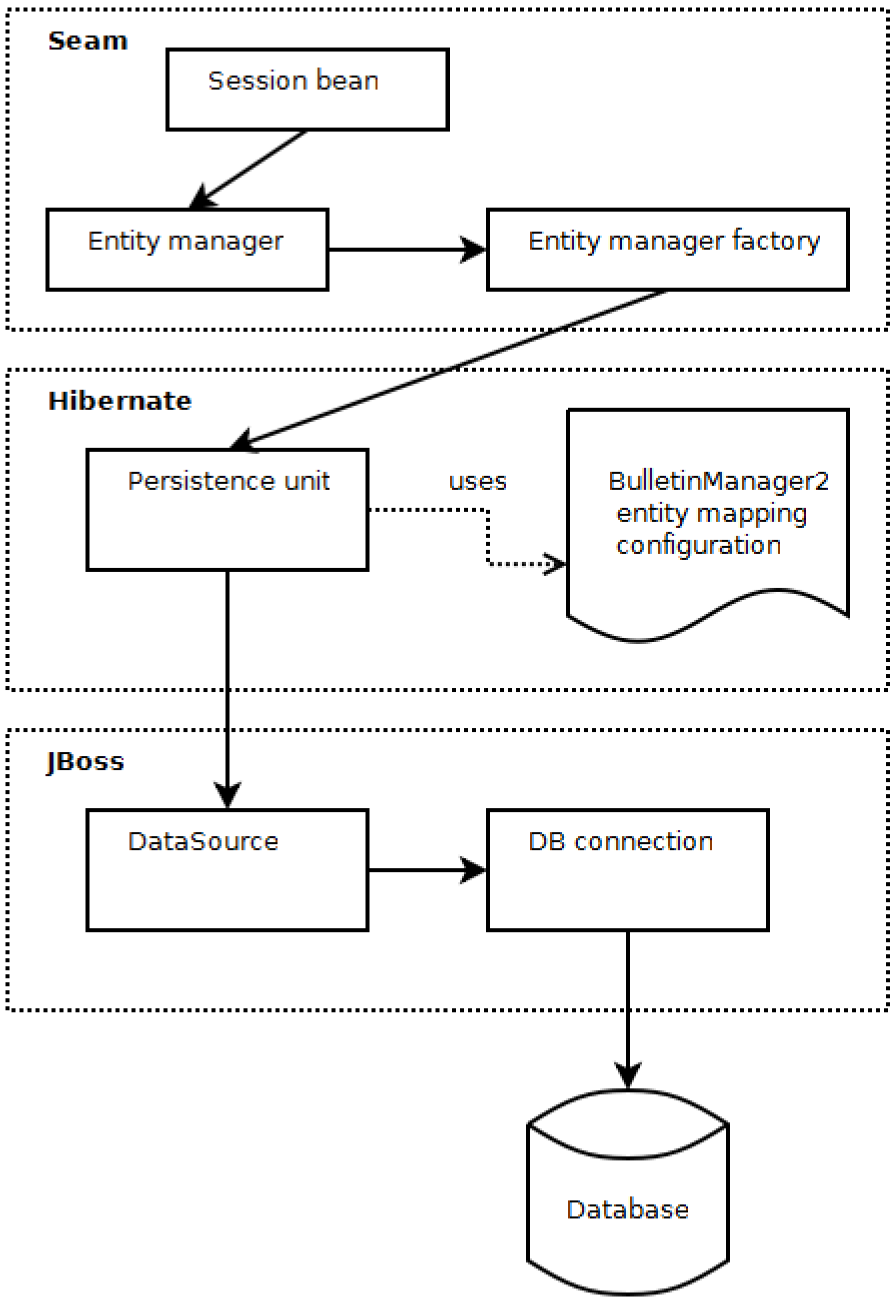


Figure 12: Database connection stack

Page templates

Page templates are implemented using Facelets technology. It is the standard layout language distributed with Seam. Seam provides many JSF tag libraries and they are introduced at the start of the template with xmlns declarations. There is one template file which contains the general page layout. That template file is included to the final page by other templates.

In this project there is only one main template called bulletinMessage.xhtml. It will print out the navigation tree and message headlines. Also the selected message is displayed by this page.

Navigation

Navigation tree is rendered recursively on the template file starting from the root element which is got from NavigationBean. The rendering is performed by Tree-tag, one of tags provided by RichFaces tag library.

Displaying a message

When a link is selected from navigation tree, a request is sent to the server. The request contains the ID of the selected category as a parameter and that value is injected to BulletinMessageList manager bean. List of message headlines that belong to the selected category is printed on the page using the list method in the manager bean.

Selecting a message headline from the list of messages, a request is sent to the server. It contains the ID of the selected message and the value is injected to BulletinMessageHome manager bean. Then the message object is available on the page template by calling the method in the above bean.

AJAX features

The provided RichFaces and Ajax4jsf tag libraries provide AJAX functionality out of the box. There is no need to write any JavaScript code on the client and no code on the server to support these asynchronous calls. All this is handled automatically by the libraries. The functionality is controlled simply by adding the appropriate tags to the page template.

In this project the selected message contents are displayed on the page with Ajax enabled elements. Selecting a message from the headline list triggers an Ajax request on to the server and the response contains the XHTML markup snippet which is inserted as a value on the page element.

Tags

Custom tag was created which will print simple HTML META tags on the page. The implementation of such tag was complicated. First the tag identifier is written on the page by the developer. In this case the tag name was *meta* under the *ss* namespace, hence *ss:meta*. Then the matching class *Metatag* is called which will insert the given attributes to the related component *UIMetatag*. Then a *MetatagRenderer* is used to print the values of the component out to the resulting page. In addition to this complicated class interdependency the component and the renderer have to be defined in *faces-config.xml* file and the tag class has to be defined in the *taglib.tld* file and also in a *taglib.xml* file where the component type is introduced to the Facelet engine. Three classes and three configuration files have to be created or modified for each new tag that is created. Seam does not help in this task and everything is JSF, JSP and Facelet specific and standard way to implement tags.

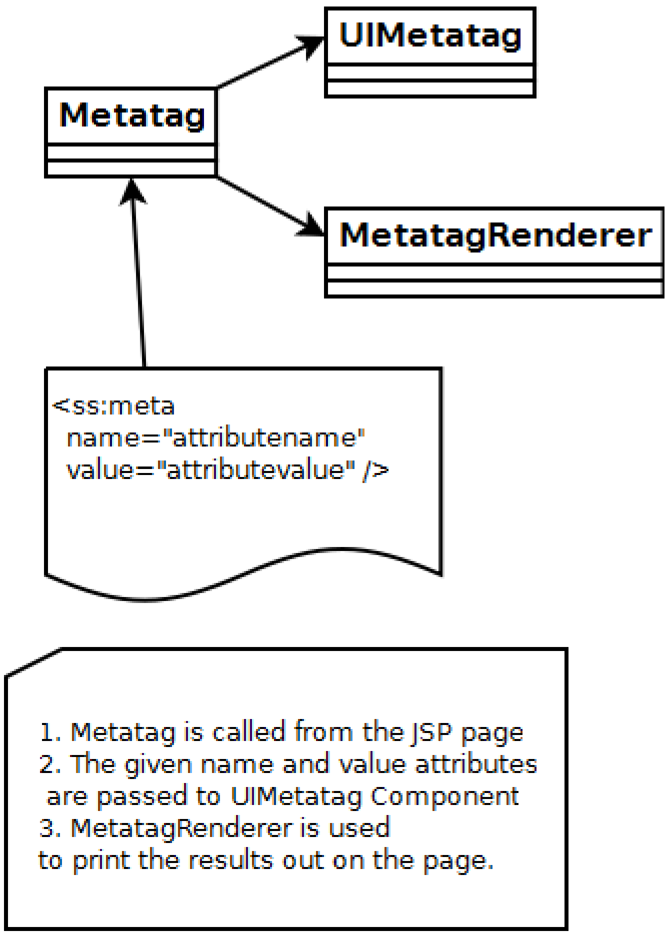

UIMetatag
Metatag
MetatagRenderer
<ss:meta name="attributename" value="attributevalue" />
1. Metatag is called from the JSP page
2. The given name and value attributes are passed to UIMetatag Component
3. MetatagRenderer is used to print the results out on the page.

Figure 13: Class structure of a custom tag

6.ARCHITECTURAL ANALYSIS AND RESULTS

6.1.Architectural analysis

The architecture of the implemented application will be analyzed using ATAM, Architectural Tradeoff Analysis Method. The method is used by creating a scenario that has some effect on the architectural environment. The response of the architecture for this effect is analyzed. After this a risk analysis and an estimation about the cost of moving to this architecture are made. Information in this chapter is based on [12].

First step in ATAM is to generate a Quality Attribute Utility Tree. This tree is represented in table form below.

Quality attribute requirement	Scenario	Importance/ Difficulty (high/ medium/ low)	Solution	Risk analysis
Ease of development	Implement multisite feature	H/M	Implement a multisite service.	Risk: Changes to different components depend on each other. This makes the code hard to maintain.
Cost of development	Move existing customer case to new application.	H/H	Use old custom code with new platform	Risk: Technology is completely new inside company. Developers are inexperienced with this and can not perform the task. Cost of training 20+ developers to new technology is too high. Only a few developers are suited to this task. Workload on these developers would be too high.

In this analysis an assumption is made that the whole ContentManager is already implemented using Seam. The scenarios are based on two different quality attributes. One is the ease of development. The company's business is based on customer cases and the ease to implement new functionality on top of existing code is a key attribute.

Second quality attribute is the cost of development. This includes the cost of training and time spent on customer development projects. Hiring new more experienced developers that can manage these techniques can be seen as a development cost too.

6.1.1.Implementing a multisite feature

Multisite feature means that the same application will serve multiple websites from the same database. The navigational structure in the application is divided for each multisite so that each site has its own section in the structure. The request is directed to the correct site based on the incoming request's Host-header.

This creates changes to the navigation parsing filters. The request processing bean may need to know which site it is serving in the multisite environment. This may affect the localization information that the module needs to attach to the request context. The page templates need to know under which site they operate for the current request. The template file itself is the same for each site and it may be necessary for the page to render some page elements in different order for different site. This creates changes to the data model that is passed to the page template.

Solution to this would be a Multisite Service that will manage this information in a centralized way and different parts of the application may access this service separately.

Due to the amount of changes that are required to different parts of the application, it makes the code hard to maintain. The changes depend on each other and later changing one part of this feature, will affect all other parts of the code as well. The components involved are a Filter, a Module and a Facelet template. Code-wise these components have nothing in common.

Later development of Multisite Service will propagate changes to all components in the application.

6.1.2.Move existing customer case to new application

It is extremely important to be able to transfer existing customers application and its customized functionality to the new platform. Otherwise the customer relationship would end and company would not be able to make further business with the hard earned customer. This longevity in customer relations is a key factor in company's success. The company will make profit if the price paid by customer for an application exceeds the development costs of such application. The development cost is a key requirement when evaluating the architecture.

Moving existing customer case to new application involves implementing all the features made in the old application on top of the new one. There are several risks that can be identified in this case. Only few of the current developers are able to do it. Training all developers to be experts in this technology is almost impossible. Recruiting new developers could be a solution, if there was any to be found. Assigning these tasks to the few current developers will lead to increased work loads and ultimately will risk losing them altogether after they decide to leave the company.

6.2.Results

As a result of the implementation phase, the system was able to read the navigation hierarchy from Content Manager database and display it on the page using a dedicated Tag. The page was managed by Bulletin Manager module. The selected Bulletin Area was displayed along with the messages that belong to that area. The message titles were displayed as a list and clicking on those links opened the message body below on the same page using AJAX. Finally some Eclipse IDE code snippets were created to experiment with easy code insertion to an existing page template.

The implementation was reviewed by the supervising developer from Ambientia. The overall concept was reviewed by company management. They decided that the technological effort in this case was too much for standard developers to adopt. The system is not overall very flexible and relies heavily on XML configuration and definition files. Maintaining these files is hard for inexperienced employees.

The amount of new technology is enormous in this case. Switching from a simple MVC pattern to JSF is hard even for experienced developers not to mention more inexperienced ones. Company of this size can not afford the cost to educate all employees to be experts in this kind of technology. Furthermore it is hard, if not impossible, to find anyone who has enough knowledge about these techniques to be qualified to educate others in the first place. The ease of coding did not meet demands. Creating a simple Facelet tag required several Java source files and multiple XML definition files. In this case only one single Content Manager module was ported under Seam framework. The work was left for several other modules, all of which are at least as demanding to implement than the Bulletin Manager module. Several older modules rely heavily on the core code in the system and are not as easily extracted and ported to other systems.

Also work was left to integrate all other core functionality to the Seam version. These include user authentication, authorization. Even if the missing functionality was integrated to the Seam version, this would lead to a situation where resources are needed to maintain two separate systems. When ever new features would be implemented to the administration side of the system (old Content Manager), the changes propagate to the new Seam public side. All this is extra work and the process is prone to errors. Amount of testing required is doubled.

Eclipse snippets were not accepted as a working solution to easy code insertion mechanism. It was judged as being too clumsy. Also it is not possible to manage the snippets in SVN repository and automatically deploy changes made in the repository version to all employees local environment. The changes have to be inserted manually to the system. This leads to a situation where the developers have different versions of the snippet library on their local machines and may interfere with each other.

7. CONCLUSION

Seam is not the solution when trying to simplify development with Content Manager. It relies too heavily on enterprise Java frameworks. Since the existing Content Manager does not contain for example any EJB3 beans, it is not feasible to start porting existing code to support any of those. The resulting system would be just split in to two completely separate systems that still would depend on each other.

JSF framework is hard to adopt for persons that are accustomed to simple MVC pattern frameworks. The underlying request life cycle has to be understood before any coding takes place. The amount of XML configuration files is too large. It was never clear to me where JSF ends and Seam starts. Facelet page templates did not provide any additional value compared to current Velocity pages in terms of simplicity. Velocity macros accomplish the same as Facelet tags, but require no extra Java source code and there is only one file where the macro is defined.

The world of Java web application frameworks is moving fast. There are tens of different frameworks that are actively developed. Most of them are licensed under open source licensing. They differ from each others in that some are more mature than others. Massive amount of work would be required in order to evaluate them all.

There is a risk involved when changing technology on which the company's product is based on. Prototyping and evaluating these platforms is required in order to determine if a specific framework or technique is appropriate for the given situation. In this thesis Jboss Seam framework was evaluated if it could be used to speed up the development of customer projects that are based on in-house Content Manager product. After implementing a simple application which used existing code base from the Content Manager and reviewing the results, the conclusions were made that Seam does not suit the current system.

Company management will evaluate the options regarding ContentManager and its future. These options are to determine if further evaluations with other frameworks is required. The choice of the framework candidate is made by the lead developers of the core product. One option is to maintain current Content Manager development as before. This will tie up company resources for each iteration when new functionality is developed for the new version. Also this requires new testing rounds for the whole product. Current customers' product upgrades is another time consuming process. Another option is to stop active development of the in-house product and start using some third-party software in future projects. Due to the large amount of support and maintenance contracts with existing clients, it is not possible to completely drop the old Content Manager but the focus could be moved away from it. Problem in this scenario is that the alternative platform has not been found. It should be cheap, preferably open source, customizable and well documented so that it could be taken in to wide scale use throughout the company.

8. REFERENCES

[1] Rinat Abdullin, Inversion of Control – IoC, http://abdullin.com/wiki/inversion-of-control-ioc.html , 10.3.2010

[2] Subrahmanyam Allamaraju, Daniel O´connor et. al., Java Server Programming J2EE 1.3 Edition, 2001, 1-861005-37-7

[3] Emmanuel Bernard, Hibernate Annotations, Reference Guide, http://docs.jboss.org/hibernate/stable/annotations/reference/en/html/index.html, 15.4.2010

[4] Ed Burns, Roger Kitain, JavaServer™ Faces Specification, Version 2.0, JSR-314, 2009

[5] DocForge, An Open Wiki For Software, http://docforge.com/wiki/Web_application_framework , 5.3.2010

[6] Jim Farley, Practical JBoss Seam Projects, 2007, 1-59059-863-6

[7] Jesse James Garrett, Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/ideas/essays/archives/000385.php , 5.3.2010

[8] Hibernate.org, http://www.hibernate.org/ , 15.4.2010

[9] Cay S. Horstmann, David Geary, Core JavaServer Faces, 2007, 978-0131463059

[10] JavaServer Faces.org, http://www.javaserverfaces.org/ , 17.4.2010

[11] Rod Johnson, Arjen Poutsma et. al., Spring Framework – Reference Manual v. 2.5.6, http://static.springsource.org/spring/docs/2.5.x/reference/index.html , 10.3.2010

[12] Kazman, Klein, Clements, ATAM:Method for Acrhitecture Evaluation, 2000

[13] Gavin King et. al., Hibernate Reference Documentation, http://docs.jboss.org/hibernate/stable/core/reference/en/html/, 15.4.2010

[14] Sam Stephenson, Prototype JavaScript Framework, http://www.prototypejs.org/ , 5.3.2010

[15] SeamFramework.org, http://www.seamframework.org/ , 17.4.2010

[16] Inderjeet Singh, Beth Stearns, Mark Johnson, and the Enterprise Team, Designing Enterprise Applications with the J2EE Platform, Second Edition, http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/ , 5.3.2010

[17] Sun Microsystems, JavaServer Pages™ Standard Tag Library, version 1.0, http://www.jcp.org/aboutJava/communityprocess/final/jsr052/, 20.5.2010

[18] Sun Microsystems, J2EE 1.4 API Specification, http://java.sun.com/j2ee/1.4/docs/api/ , 10.3.2010

[19] Sun Microsystems, J2EE 1.4 Specification, http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf , 10.3.2010

[20] Sun Microsystems, The J2EE 1.4 Tutorial, 2005, http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html , 10.3.2010

[21] Walls, Breidenbach, Spring in Action, 2005, 1-932394-35-4

[22] Wikipedia.org, Hibernate (Java), http://en.wikipedia.org/wiki/Hibernate_(Java) , 15.4.2010