INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Softwaretechnik
Oettingenstraße 64 D-80538 München

Ludwig
Maximilians
Universität
München
LMU

A study of java core web technologies

Ferdinand Gollas

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	Okt. 2006
Abgabe der Arbeit:	Sept. 2008
Betreuer:	Prof. Dr. Martin Wirsing
	Andreas Schröder

Zusammenfassung

Der Begriff Web 2.0, der seit einigen Jahren in aller Munde ist und die zweite Generation des Webs einleiten sollte, hat rd bis heute stark geprägt. Überall entstehen hochmoderne und multimediale bis Desktop-ähnliche Anwendungen im Web. Die Palette reicht von Bilder- und Videoplattformen über Blogs, Wikis und viele mehr. Da die Funktionalität und damit auch die Komplexität dieser Anwendungen deutlich angestiegen ist, waren neue Technologien und Frameworks gefragt, die die Entwicklung wieder vereinfachen sollten. Diese ließen auch nicht lange auf sich warten und mittlerweile gibt es sie wie Sand am Meer. Gleichzeitig wird es für den Einzelnen aber immer schwieriger, sich einen Überblick zu verschaffen und die beste Technologie für ein bestimmtes Projekt zu finden. Diese Arbeit soll dies erleichtern, in dem sie einige ausgewählte Technologien kurz vorstellt und den Java Web Technology Stack im Java EE Umfeld detailliert präsentiert. Parallel dazu wurde eine kleine Beispiel-Anwendung entwickelt, anhand derer die relevanten Aspekte der Java Web Technologien veranschaulicht werden. Die Anwendung ist ein Ticket-Billing System, welches es ermöglicht, Probleme (Tickets oder Trouble Tickets) zentral zu verwalten und zu lösen und diese gleichzeitig mit Billing-Informationen zu versehen, um sie anschließend abrechnen zu können und einem Kunden in Rechnung zu stellen. Die Client-Server Anwendung wird mit den aktuellen Java Technologien Java Server Faces und dem Templating Framework Facelets realisiert. Es werden die Vor- und Nachteile der einzelnen Techniken dargestellt und es zeigt sich, dass mit Java's Komponentenbasierter Technologie JSF zusammen mit der Templating Technologie Facelets für Entwickler mit Erfahrung in diesem Bereich sehr einfach komplexere Web Anwendungen mit viel Benutzerinteraktion erstellen lassen, JSF aber keineswegs die bisherigen Java Technologien (Servlet/JSP) komplett ersetzen kann.

Abstract

The term "Web 2.0" which is well known since some years, ought to introduce the 2nd generation of the web and has led to a radical change of it. The amount of modern multimedia services and Desktop-like web applications has drastically increased, including platforms for pictures and videos, blogs, Wikis and much more. As the functionality and thus also the complexity of those applications has grown, the need for new technologies and frameworks which would ease development has arisen. The software market has reacted quickly and issued many technologies which should meet the demands of the developers. Today, the range of web technologies is enormous and it gets more and more difficult to get a general idea and to find the best technology appropriate for one project. Therefore, this work will introduce some meaningful technologies with focus on the Java Web Technology Stack. Additionally, some interesting aspects of web applications are highlighted and exemplified by means of an example web application which was developed in the context of this work. The application represents a "Ticket-Billing System", an effective solution for managing and solving "problems" (Tickets or Trouble Tickets) of users and to discount and bill them to a customer after adding billing relevant data. The client-server application is implemented using the Enterprise Edition of Java and extensive usage of the contained JavaServer Faces (JSF) technology. Finally, the pros and cons of the Java technologies are discussed with the result that JSF is the technology of choice for developers with JavaEE experience for developing complex web applications with much user interaction, but also that JSF will not be able to completely replace the former existing and widespread web technologies for Java (Servlet/JSP).

Table of contents

A study of java core web technologies 1
1. Glossary 4
2. Introduction 6
3. Ticket Billing System 6
3.1. Requirements 7
3.1.1. Security 7
3.1.2. Use cases 7
4. Technologies and Realization 9
4.2. Presentation tier 9
4.3. Business logic tier 10
4.3.1. PHP: Hypertext Pre-processor (PHP) 10
4.3.2. ASP (.NET) 10
4.3.3. ColdFusion 10
4.3.4. Ruby 11
4.3.5. Java 11
4.3.6. Conclusion 11
4.4. Java Platform 12
4.4.1. Localization 13
4.4.2. JavaBeans 13
4.4.3. Java Servlet 13
4.4.4. JavaServer Pages 15
4.4.5. Java Persistence API (JPA) 16
4.4.6. JavaServer Faces (JSF) 18
4.4.6.1. View development 19
4.4.6.2. JSF Managed Beans 20
4.4.6.3. Filter 21
4.4.6.4. Unified Expression language 21
4.4.6.5. Tag handlers and component trees 22
4.4.6.6. Converters 22
4.4.6.7. Event and listener model 23
4.4.6.8. Request Life Cycle 23
4.4.6.9. Navigation 24
4.4.6.10. Security 25
4.4.7. Facelets 26
4.5. Testing 28
5. Conclusion 28

1. Glossary

The first section will introduce some technical terms frequent used in the context of the ticket-billing system which will be described in section 3. The terms come from von the domain model and represent the most important entities of the system. Figure 1 shows the entities and their relations.

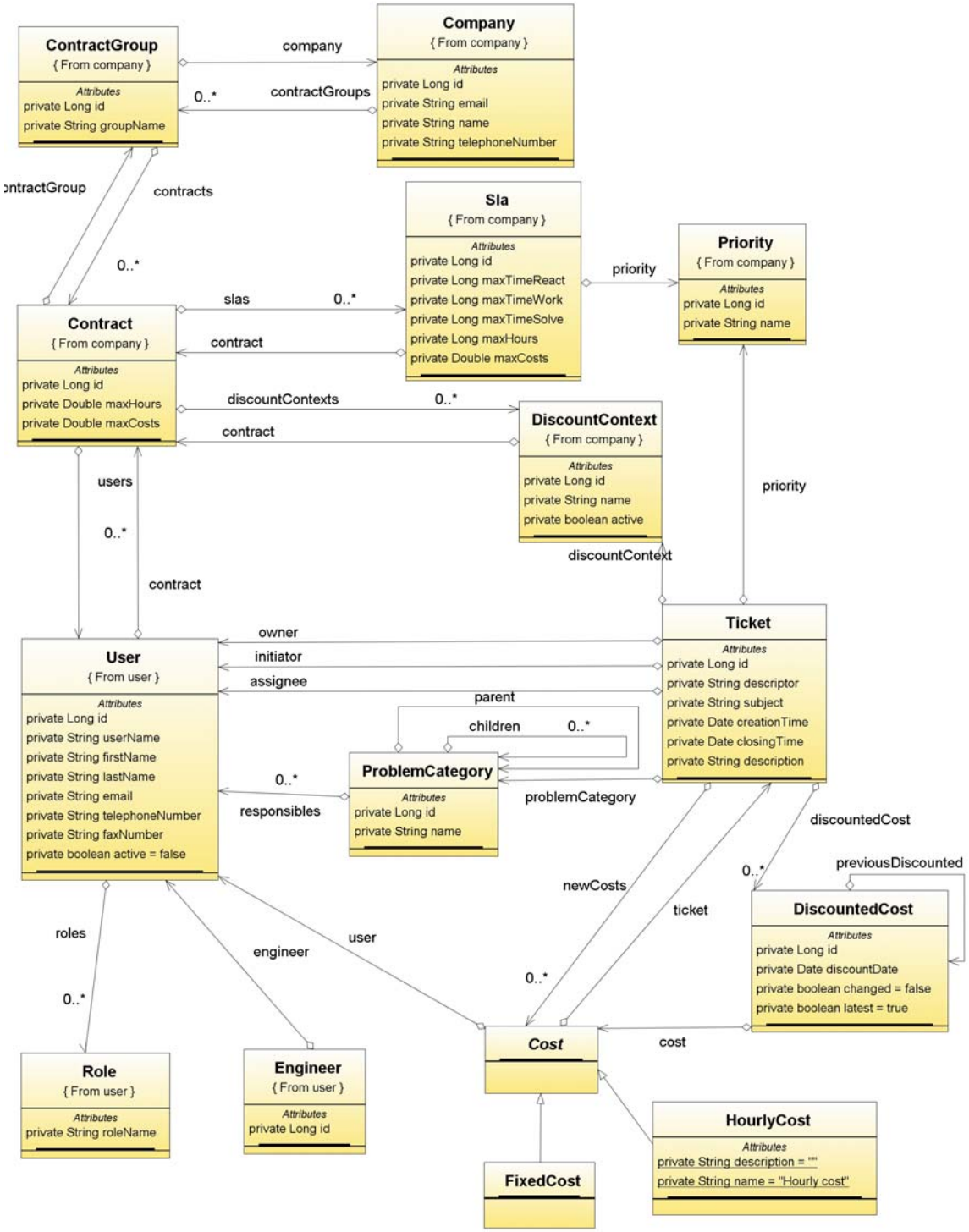

Figure 1: Ticket-Billing System – Entities

The entities are described in short:

Ticket: The Representation of a user's problem which has to be solved. It holds any information describing the problem of the user and additional information occurred while working on the ticket or needed for solving the ticket, including a title, description, notes, priority and optionally a problem category.

Priority: To represent the urgency of the ticket, every ticket has a priority. This priority implicitly defines how fast the ticket has to be solved before service level agreements are broken.

Problem Category: A problem category can optionally be assigned to a ticket. It represents the type of the ticket request. Possible types include general types as "Hardware problem" or "Software problem" but also more specific ones like "Mail service" problems. These problem categories must be organized in a tree structure, meaning that every category has exactly one father and zero to several sons. The root category must hold all possible problem types.

Service Level Agreement (SLA): SLAs are a common instrument to define different quality of service guidelines for the service provider and the resulting costs for the customer within a contract. They can warrant that certain services are completed within a given period of time or define a maximum response time which must not be exceeded when a defined service is requested. Additionally, they define the mostly financial consequences when a SLA is broken. In the ticket billing system, SLAs can be used to define time-frames for different service levels, including the maximum duration for reacting on a ticket and solving it, billed costs and an upper cost limit for all tickets assigned to the given SLA. SLAs are linked with a priority.

Discount Context: Discount contexts are required by the bookkeeper to bill the entries on the name of the context. Setting up different discount contexts for different projects or departments makes it easier for external companies to relate the arisen costs to one project/department, especially when extensively using the ticket system. Thus, when creating the invoice, the total sum can for example be broken down into several items, one for each discount context.

Company: The company holds company related data required for the invoice including the contact person, billing address, telephone number and a list of departments.

Contract and Contract Group: The contract defines billing related data for several SLAs including cost factors and hourly rates. Contracts are organized in groups with each group holding the original contract for one specific customer and all subsequent contracts which result in changing a contract in the group.

Costs: The billing system supports two different cost types: Fixed and hourly cost. Both of them represent raw values which are required for further cost calculations of internal and external (billed) costs. Hourly costs are the result of an engineer working on a ticket. They hold the time the engineer spent working on it. Fixed costs occur when external services or extra charges are required solving a ticket (e.g. some hardware has to be acquired). They represent the raw amount of money which had to be paid for the external service.

DiscountedCost: Discounted cost is not a specialization of Cost but rather indicates that a cost entry is discounted. It holds a single discount date and is linked with one cost entry. Every time a cost entry C is discounted, a discounted cost entry is created and associated with C and containing the discount date. If C was already discounted before, the corresponding discounted cost entry is not overwritten but is added to the current discounted cost entry and marked as previously discounted.

Cost Variables: Cost variables can be seen as modifiers to the raw cost values. They are applied to the raw cost values to obtain the resulting costs of a ticket. To differentiate between the costs which are billed to the customer and the internally arisen costs, each cost variable may occur twice, once for the internal and another for the external price. The variables can be of type Summand or Factor. Applying a Summand variable with value S to a raw value V will return the value of S + V. Analogous, by applying a Factor variable with value F to a raw value V returns F * V. Table 1 shows all available cost factors in the system.

Entity	Property	Description
Contract	Hourly rate	Fallback hourly rate for engineers working on tickets for this contract.
	Cost factor	Fallback cost factor.
SLA	Cost factor	Cost factor which can be applied for tickets with the priority defined in the specific SLA.
	Hourly rate	Defines engineers' hourly rates when working on a ticket with the appropriate priority.
Engineer	Hourly rate	Hourly rate of engineer. Can be overridden by either contract or SLA hourly rate.
Problem category	Cost factor	Factor to apply for tickets assigned to the given problem category.

Table 1: Billing - Cost factors

2. Introduction

Web 2.0 is an expression that has become very well known since some years (Hass, 2008). It means the second generation of the World Wide Web. This is of course very far-fetched and in fact does not mean that there were introduced new and trendsetting technologies that were not available before but rather addresses the way of using the World Wide Web and established web technologies. Nowadays the internet is transitioning from an information to a communication platform where people can do nearly everything one can imagine. Of course, there had to be a change in the way of developing Web Applications, too, otherwise this would not have been possible. But the developers recognized this trend and developed myriads of new web applications that would meet the demands of the people.

The result is a rich offer of applications like Blogs, Wikis, Podcasts, RSS feeds, social networking services and so on, but also some new web services like eBay and Gmail.

Tim O'Reilly defined the Web 2.0 as "business revolution in the computer industry caused by the move to the internet as platform, and an attempt to understand the rules for success on that new platform." (O'Reilly, Web 2.0 Compact Definition - Trying Again, 2006) Economists had realized that the web is an excellent platform for offering new services that had been demanded in the minds of many people since a long time. They satisfied these demands with new and successful business concepts which were based on the web as platform.

Beyond these new web services which were mainly developed to satisfy the general public's need for communication, networking, information retrieving and selve-portraying, many economists have recognized the advantages of web based applications in their company. They see the advantages on the reduced costs of some web applications in contrast to standalone desktop applications.

These financial benefits among other things come from the ability of responding more quickly and cost-effectively to changing market conditions and the possibility of reusing (web) services in many applications. This saves among others maintenance costs, manpower and developing costs.

To summarize, the introduced trend of modern web applications and services which mainly started with the introduction of Web 2.0 has now become a very meaningful sector which affects many parts of the IT industry. It is quite sure that this trend will continue and that more and more companies will jump on the bandwagon and will capitalize the possibilities of web 2.0.

This trend has established a new market in the computer industry. Many new technologies and standards like Java EE 5.0, Enterprise JavaBeans, Servlets or SOAP have been developed or improved to master the growing complexity of modern web applications and to simplify their development. For the same reason, many frameworks have been developed to support the web application developers in nearly all respects. Spring (Wolff, 2007), Ruby on Rails (Tate) or the Google Web Toolkit (Martin Marinschek, 2007) and many others help the developer to easily cope with persistency, security, view development, navigation (among others) of web applications.

As computer science has become an extremely extensive profession with nobody being able to cover every field in detail, it is beneficial to specify oneself to one domain of the computer science. This matters when you have a look at the job offers for computer scientists. Most of the job offers call for detailed knowledge in one domain. So, companies look for database experts, Java experts or also since a while for web application experts.

As web development is a growing branch with excellent future prospects in the IT sector, it is worth having a closer look on it. Therefore, this work will provide deeper insights into enterprise web application development, introducing some selected technologies with focus on the Java Enterprise Edition (Stark, 2006) and demonstrating the main aspects by means of an example application. The example application itself can be regarded as enterprise web application, too, to demonstrate the realization of many enterprise related aspects with the appropriate technology. It is introduced in the next section.

3. Ticket Billing System

Having described the most important terms of the application in the glossary, this section describes the ticket billing system and its requirements including security features and the business case.

The example application is a ticket billing system which is an ordinary (trouble) ticket system combined with billing functionality. Ticket systems are most often used in the IT infrastructure management and support sector for tracking user problems. In this context, a trouble ticket is the representation of a user’s (commonly technical) problem asking for support at the trouble ticket system provider. The provider will then try to solve the ticket, appending all required information to the ticket. Once the ticket is solved, the initiator of the ticket is informed. The advantage of the system is that the information required for solving a problem is stored and can be accessed at any time, the same or a similar problem occurs in the future.

To have more complex application flows than simple create, update and delete functionality and thus to demand more from the technologies, a billing functionality was added to the ticket system. It allows for internally discounting

the costs emerging from solving the ticket and for billing external parties for the costs. The internal and external costs can be controlled by setting cost factors, subject to the ticket initiator's contract, company or Service Level Agreements.

3.1. Requirements

Commonly, every larger software project should start with a requirements analysis where the functional and non-functional requirements are gathered from discussions with stakeholders of the system. Functional requirements may be calculations, technical details, processing or other functionality definitions. They are supported by non-functional requirements which impose constraints on the design or implementation (performance requirements, security, or reliability). Having no real stakeholders for the ticket billing system (TBS), the requirements were defined to point out different aspects of the technologies.

3.1.1. Security

As enterprise applications commonly are protected by security constraints to control the access to critical resources, the TBS was also designed to implement several security levels for different actors including a client, bookkeeper, engineer, administrator and cost controller. They are described in Table 2.

Role	Description
Engineer	Engineers commonly work for the company using the ticket billing system and try to solve the tickets.
Bookkeeper	A bookkeeper has access to the billing related functionality of tickets (discount, export) and customer related data.
Administrator	Administrators have full access to every part of the system.
Client	Clients commonly work for a customer and create tickets which are solved by an engineer. They are allowed to see their own tickets and related data only.
Cost controller	As clients have only very restricted rights and cannot access the billing information, the cost controller was introduced to provide access for billing data of tickets. Cost controllers can see all tickets and external costs of their clients.

Table 2: Security – Roles

It is possible that a real person can act as more than one actor. In this case, he is able to access the aggregated functionality of the roles he belongs to.

3.1.2. Use cases

Here is a list of the interesting use cases. They are described informally because it is sufficient to get a rough overview about the functionality of the system.

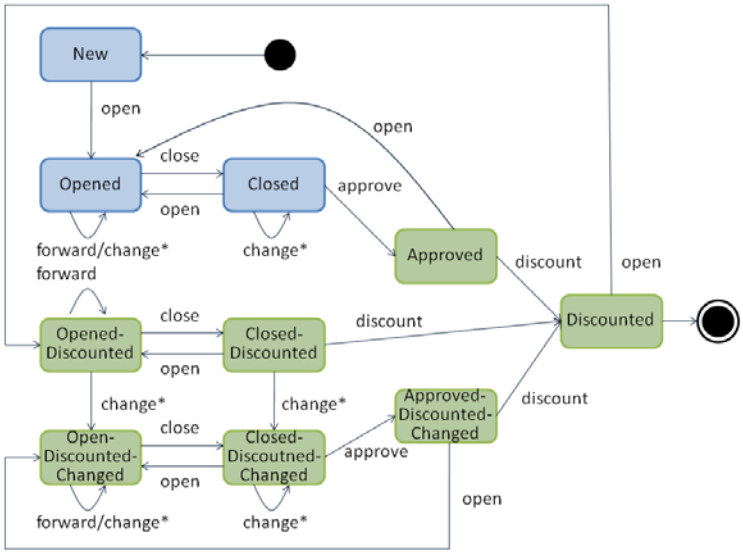

Figure 2: Ticket - State transitions

The basic functionality of the Ticket System includes creation, update and deletion (CRUD) of the described entities (see Domain Model in Figure 1). More complex functionalities are the ticket related ones: Opening, closing and forwarding of a ticket and the auto assignment of new tickets. The performed actions may differ, depending on the current state of the ticket. Figure 2 gives an overview about the possible ticket states and transitions. The blue colored states represent the relevant ones for the ticket system (New, Opened, Closed) whereas the green ones are required for the billing functionality (OpenedDiscounted, OpenedDiscountedChanged, ClosedDiscounted, Approved, ClosedDiscountedChanged, ApprovedDiscountedChanged, Discounted). A state transition can be derived by a user's action (open, close, approve, discount and the change of billing relevant ticket entries).
Here is a description of all ticket concerning use-cases. Some of them were already introduced in Figure 2, others have not yet been introduced (ticket export, auto assignment of tickets).
Open: Tickets can only be opened by engineers, activating the Open button of a ticket. If the ticket has not yet been assigned, it is assigned to the opener. An already assigned ticket can only be opened by the assignee himself or by an admin. The ticket is now in the Opened state. The same processing is done when a ticket is opened in the following states but with eventually different result states: Approved, ClosedDiscountedChanged, Closed, ClosedDiscounted, Discounted and ApprovedDiscountedChanged.
Close: To close a ticket, it must be assigned to an engineer and its discount context must not be empty. Only then, it can be closed and a closing timestamp is saved with the ticket.
Forward: Forwarding a ticket means changing the assignee from one engineer to another. Thus, only tickets that are already assigned can be forwarded. If the new assignee is not an engineer, an error must be thrown.
Approve: To make sure that every engineer having worked on a ticket has entered all billing relevant data before a ticket is discounted, tickets have to be approved before they can be discounted. This has to be done by the engineer having worked most recently on the ticket. He has to assure that every other engineer who has worked on the ticket before, has also entered all billing relevant data.
Discount: When a ticket is discounted, a discount event is created holding the discount date. The ticket's cost entries are marked as discounted and associated with the event. To calculate the discounted costs of a ticket, its cost values and cost variables are restored for a given discount date to obtain the total amount of costs. Figure 3 illustrates the discount of a ticket.
Auto assign: To increase productivity tickets can be assigned automatically immediately after creation. This assignment should not happen randomly but by considering the problem category tree. As described before, problem categories are used to categorize a ticket and to assign engineers to those categories. Thus, the auto assignment logic looks for active engineers who are assigned to the ticket's problem category and select the first one. If there is no engineer available, the first engineer which is assigned to a category, walking through the problem category tree from the ticket's problem category to the root category, is selected. In the second step, the ticket gets assigned to the selected engineer and its status is set to New.

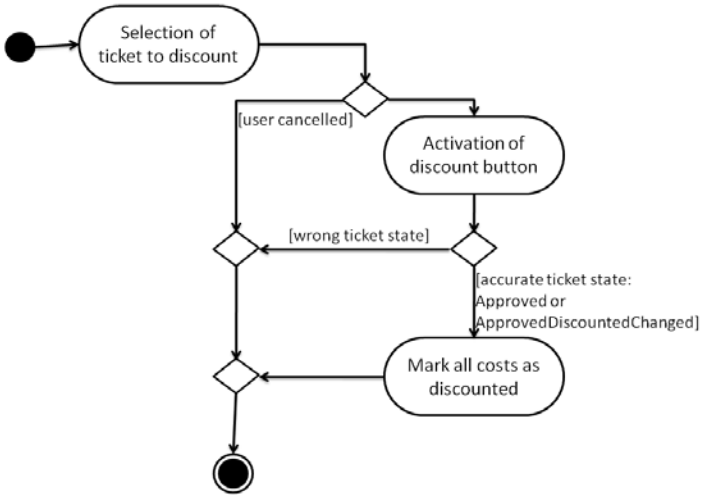

Figure 3: Ticket discount activity diagram

Export: The billing system also supports exporting the billed data to a CSV file which can then be imported from external billing software.

4. Technologies and Realization

In order to develop a client-server application, it is necessary to meet some decisions concerning the structure of the application. The most common one is a three-tiered structure which consists of the following layers (see Figure 4):

1. Presentation tier
2. Application tier or business logic tier
3. Data tier

The presentation tier is the user-interface through which the user can access or invoke the application's logic. The user's requests are forwarded to the application tier which executes the corresponding business logic and renders the response. If it needs to access some persistent data from the model, it interacts with the data tier that contains the model data and controls its access. In the following, some relevant technologies for the presentation, application and data tier are introduced.

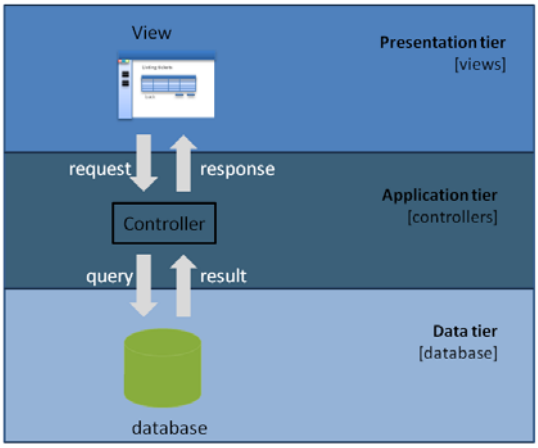

Figure 4: Three tiered architecture pattern

4.2. Presentation tier

In client-server architectures, the first tier is realized by a so-called application client. This client allows the user to interact with the server-side application tier. Most frequently, the browser is used to display the user-interface and to forward the user actions to the server. This has several advantages: It decreases the developing and support costs in contrast to developing a custom application client which had to be updated on every change of the server-sided interface. Due to the wide distribution of web browsers, it is guaranteed that virtually everyone can use the application. Furthermore it makes the server-sided part of the application independent from the user's operating system.

The server-sided application then has to produce "web-documents" for the client requests which can be displayed in the browser. These "web-documents" normally consist of html/xhtml with embedded scripts and other content, which have to be interpreted by plugins, allowing for more complex, dynamic and interactive clients. The TBS does not require any plugins as it would require every user to have the appropriate plugin installed. Sometimes, even company-policies prohibit the installation of plugins. However, JavaScript is nowadays (web 2.0) so widespread that it is not regarded here as plugin, and thus is utilized within the html documents to guarantee the successful operation of some frameworks producing web documents.

Once the decision of using the browser as first-tier is made, the type of the client has to be discussed. Should the client be implemented as a complex application, containing as much application logic as possible (rich client) and also store the data locally or should the application logic be hosted on the server and the client only be used to call this logic and to display the rendered results (thin client) or is a mixture of both the best solution (hybrid client)?

Thick (fat) clients contain (almost) the entire business logic and store most of the data locally. Thus, they do not necessarily require a constant network connection to the server. First, this was not desired for my application as it would have caused huge costs synchronizing the local and server data and secondly there are some disadvantages porting the business logic to a client-sided scripting technology: It cannot be guaranteed that every client interprets the scripts as desired and the changes made on the client have to be cross-checked on the server again which leads to duplicate code. It is rather desirable to hold the application data and business logic centrally on the server to prevent this and to also allow clients with fewer resources to use the application. This means however, that the client is very dependent from the server and has to request every single change from the server and thus causes more traffic. But the increasing traffic is amortized immediately if it is considered that most of the users have broadband access to the

internet and that technologies like AJAX (Asynchronous JavaScript and XML) which make partial page reloading possible can reduce the traffic again. On this accounts, I have decided to develop a thin client.
To summarize, the framework for the first tier is determined. In this case, it is the best solution to utilize a thin client, executed and interpreted in a common browser. It guarantees the nearly unrestricted possibility of using the application through its wide distribution and without being dependent from the underlying software architecture or restrictions towards lacking resources.

4.3. Business logic tier

In a three-tiered architecture, this is the middle tier, holding the company specific business rules and workflows. This section will highlight some technologies which could be used for the implementation of this server-sided tier.

4.3.1. PHP: Hypertext Pre-processor (PHP)

PHP is a scripting technology which was mainly designed for developing the server-side part of web applications but can also be used standalone from the command-line. The language was introduced by the PHP Group with a free licence and has reached version 5.2.0 to this day.
PHP code is embedded into HTML and is interpreted. It is denoted to be an easy to learn, platform independent and dynamic language which provides easy database access, support for structural reflection and object orientation. Although it is interpreted, it is quite fast due to its fast and efficient memory management. In addition to the built-in libraries for verifying user input, serialization and handling ZIP archives, there are many more free and useful extensions provided by the PHP Extension and Application Repository (PEAR).
On the other hand, PHP still lacks some of the features of other current programming languages including namespace support, thread-safety in multi-threading environments, a strong type system and some object oriented paradigms like method overloading. PHP has always been a popular scripting language. Thus, it is not surprising that there exist plenty of frameworks[1] facilitating the development of web applications. They range from Rapid Application Development (RAD) to Database Driven Development frameworks supporting scaffolding, AJAX, localization, etc.

4.3.2. ASP (.NET)

Active Server Pages (ASP) is an active scripting engine from Microsoft. It is an add-on for Microsoft's Internet Information Service (IIS), based on Microsoft Windows. Its aim is to facilitate server-side scripting by providing built-in objects (session, request, server, response, etc.). Although the most common scripting technology is VBScript, it does also support other scripting technologies like PerlScript or Jscript. Similar to PHP, the ASP scripts are embedded into HTML and are interpreted. The successor of ASP 3.0 is called ASP .NET, indicating the affiliation to Microsoft's .NET platform. This means that there is a built-in support for other .NET languages (Visual Basic .NET, C#) which should make the inline-scripting obsolete as developers are encouraged not to write directly to the response object but to use the newly provided APIs to generate the page output or to encapsulate the code into custom tags or components.

4.3.3. ColdFusion

ColdFusion is another technology with distinguishing features concerning web application development. It combines an application server, a markup scripting language and a web application development framework. Thus, ColdFusion cannot be compared with a standalone programming language as a whole but rather has to be divided into three parts, the server, scripting language and the framework whereas each part has to be regarded independent from the others.
Application servers are more complex than conventional web servers (e.g. Apache web server). They provide a runtime environment for the deployed modules and applications and offer several services including transactions, database access, user and role management, authentification, persistency, and much more. Additionally, application servers are scalable and provide functionality for monitoring, logging and software lifecycle management. Thus, they promise data and code integrity, centralized configuration, security and rapid application development (RAD) as the developer must not face with every detail of web application development.
Although most of the Application Servers are currently written for the Java EE platform including WebLogic Server[2] (BEA), JBoss[3] (Red Hat), Sun Java System Application Server[4] (Sun Microsystems), there exist some for other platforms (Appaserver[1], Base4[2], Zope[3]), too.

[1] Web application frameworks for PHP: http://en.wikipedia.org/wiki/List_of_web_application_frameworks#PHP
[2] BEA WebLogic Server: http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/server
[3] Red Hat JBoss Application Server: http://www.jboss.com/products/jbossas
[4] Sun Java System Application Server: http://www.sun.com/software/products/appsrvr/index.jsp

The web application framework appreciates the bundle even more by adding AJAX, PDF, RSS, session, cache and ZIP and JAR file features. This allows dynamic HTML to PDF generation, client side code generation and form validation and much more. It comes with integration for Microsoft Exchange Server, Java EE and .NET.
The ColdFusion Markup Language (CFML) contained in the bundle, is a fully featured tag-oriented scripting language. It includes a wide range of tags, functions, variables and expressions. Custom, reusable tags can be added to define new functions or elements.

4.3.4. Ruby

Although Ruby exists since 1995, it did not attach much attention for many years. This changed with the publishing of the web application framework Ruby on Rails (Tate & Hibbs, 2006) which introduced some new and trendsetting features allowing database driven RAD. It is based on Ruby, a reflective, dynamic and object-oriented scripting language. Like other scripting languages, Ruby must be interpreted and executed inside a virtual machine. Virtual machines exist for C (default implementation), Java (JRuby), .NET (IronRuby) and others. Ruby was designed to be easy to learn and use. The framework tries to continue with this intention by following the philosophies "Convention over Configuration" (CoC) and "Don't repeat yourself (DRY) and by adding object-relational mapping (ORM) and automatic skeleton code generation (scaffolding) which allows for automatic generation of the view from the database model. This reduces programming and configuration overhead as class-names are mapped to database table names by default and views for creating, updating and deleting entities can be created automatically. Configuration is reduced to a minimum and is solely required when the conventions are not kept.

4.3.5. Java

Java is an object-oriented, imperative programming language, developed by Sun Microsystems. The importance of ease of learning and reduced error-proneness is reflected in the syntax, similar to C/C++, the static and strong type system and the suppression of some features like operator overloading and multiple inheritance. Another important design goal was the platform independence. This is achieved by introducing an intermediate-format, the Bytecode at compiling: The code is not directly compiled into platform-specific executables but to Java Bytecode. This Bytecode can then be executed on every platform, having a Java virtual machine installed.
Sun provides three different platforms for the Java programming language which mainly differ in their functional range. The micro edition (J2ME) targeting devices with limited resources, the standard edition (J2SE) which is commonly used for desktop applications and the enterprise edition (J2EE) utilized in the enterprise environment. The latter is the platform of choice developing an enterprise web application with Java and thus will be introduced in the following sections.
Concerning web application frameworks, Sun itself has released some technologies and specifications easing the development of web applications including JavaServer Faces, JavaServer Pages and JavaBeans.
In this context, Sun has invented Servlets, a new interface technology between the server request processing engine and the web application. Similar to CGI, it controls the transmission of necessary client and server information to the web application and instantiates it. But in contrast to CGI, Servlets do not have CGI's performance problems which result among others from the time-consuming instantiation of the interpreter on every request. They are very lightweight, better integrated in a Web server and can be preloaded, avoiding instantiation delays.
Within the current release of JavaServer Pages (JSP), Sun has introduced a XML based view technology on top of the Servlet API which makes HTML generation easier. JSP can be extended with custom tag libraries and is compiled into Java Servlets to be executed from the Web container.
Sun's latest technology is the JavaServer Faces specification, an extension to JSP which brings along a tag library representing all the standard HTML components enriched with additional features (sorting, validation, state-maintenance and much more) and a request processing life-cycle.

4.3.6. Conclusion

For this work, Java (Java EE platform) was chosen to be the technology which will be demonstrated in the following sections. The reasons which have been of prime importance for this decision are the following:
Java is very widespread and thus has become accepted in many domains. According to Google Directory[4], for no other technology there are as much issues, products, books, articles, etc. as for Java (3007 issues), followed by PHP

[1] Appaserver project page: http://timriley.net/index_appaserver.html
[2] Base4 project page: http://www.base4.net/
[3] Zope web page: http://www.zope.org/
[4] Programming languages on Google Directory: http://www.google.com/Top/Computers/Programming/Languages/

(1453 issues) and Perl (950 issues). This great acceptance results in many advantages using and developing with Java: Java technologies are often supported within other products. E.g. the Java Servlet technology is supported by many open-source Web servers including Apache, the leadership in Web server market share[1] and many others[2] and with it enabling the use of Java as server side technology. There is also a great third party support which results in many additional features, components, frameworks or extensions which prevent the developer from reinventing the wheel in many cases. Developers will be supported by a large community of users and developers exchanging their knowledge in miscellaneous forums, tutorials, FAQs or other kind of documentation.
The recent change to the GNU General Public Licence (GPL) for the Java core code and no dependency of any proprietary software will lead to a greater acceptance of the Java Platform and thus will increase the spreading of Java.
With Servlets and JSP, Sun has developed two very high-performance technologies dedicated to the development of web applications. This results from the good integration of Servlets into the web server and the JSP techniques just-in-time compilation and dynamic recompilation.
The principle of Java "Write once, run everywhere" is kept for web applications as far as possible: Web applications can be deployed within every Application Server sticking to the J2EE specification and the two major Application Servers Apache JBoss and Sun Java System Application Server are available for many platforms including Windows, Unix and Macintosh.
Another argument is the ease of development with the Java Platform: With NetBeans[3] and Eclipse, there are two comfortable Integrated Development Environments (IDEs) with support for most of the J2EE features. They hide a good portion of complexity of technologies from the developer, providing templates, default configurations and wizards for many facets.

4.4. Java Platform

After motivating the usage of Java as base technology for my work and bringing up some aspects of the language, it is worth, having a closer look on the interesting facets of the platform. Besides Java's Micro Edition (ME) and Standard Edition (SE), the Enterprise Edition (EE) is Java's answer towards the request of easing web based application development. In contrast to the ME, a subset of the SE, designed for the use in small, resource-constrained devices and to the SE, which is mainly used for creating desktop applications, the Enterprise Edition[4] which is based on the Standard Edition, adds some features designed for web-based enterprise applications. It defines a platform for server programming with the Java programming language. Custom applications are commonly deployed as modules and will be executed inside the Web or EJB Container of the application server which provides a runtime environment for the applications (see Figure 5). The Java EE platform comes with many other technologies included: As Java EE applications are commonly accessed with a browser, with Servlets, JavaServer Pages and JavaServer Faces, the Enterprise Edition includes technologies which ease view development. To set up database connections and access databases, the Java Persistence API (JPA) which is included in the platform can be used. It defines an Entity Manager interface to easily create and modify database objects. The underlying database provider is not forced and can be configured with a XML document.

[1] July 2007 Web Server Survey: http://news.netcraft.com/archives/2007/07/09/july_2007_web_server_survey.html
[2] According to Wikipedia: http://en.wikipedia.org/wiki/Comparison_of_web_server_software
[3] NetBeans IDE: http://www.netbeans.org/
[4] Java EE: This document does not differ between the specification (Java EE 5) and the implementation (Java EE 5 SDK), provided by Sun Microsystems.

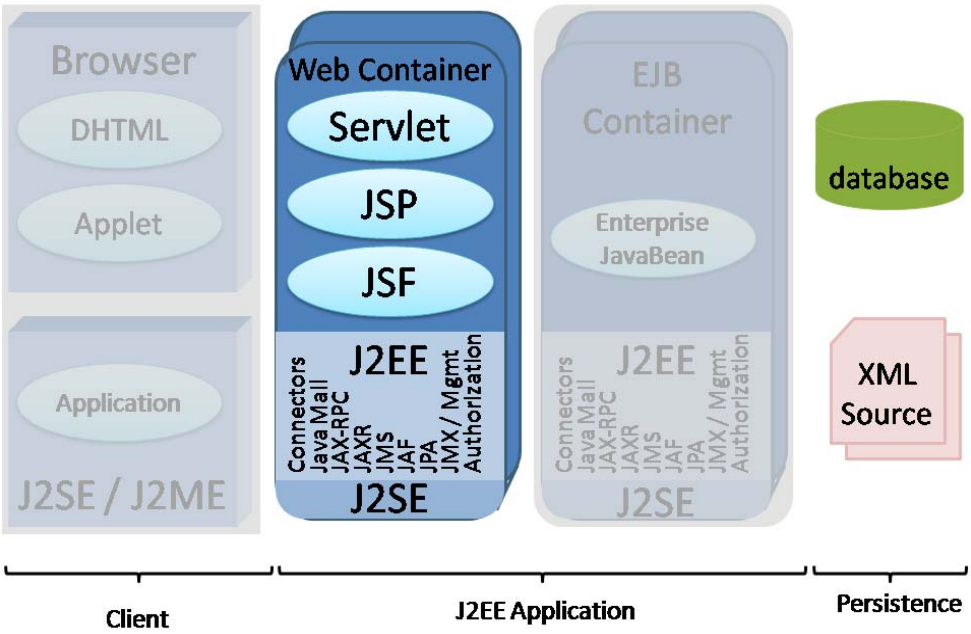

Figure 5: Java EE - Component overview

More on the mentioned technologies can be found in later sections.

4.4.1. Localization

Since SDK 1.1, Java comes with Resource Bundles, meeting the applications' demands for localization and internationalization: Software developers may provide the displayed texts and messages of their application in several languages, letting Java decide which one to choose, depending on the country or geographic region of the user. But also the displayed values of numbers, currencies, dates and times have to be adapted to specific countries.
Therefore, the Locale object was introduced to represent the language and optionally the geographic region and a variant of a user or the underlying operating system. Formatters can now convert the data to be displayed into the language specific format, using the user's or system's Locale. The Resource Bundle loads the localized text from one or more resource bundles (files or Java classes), each of them containing the data for a specific locale and topic.
This approach has several advantages: All the localized data are no more spread over the source code but are encapsulated into several files (resource bundles). This makes the code more clear, the software and resources easier to maintain and translate and changes to displayed text do not require the code to be recompiled.

4.4.2. JavaBeans

JavaBeans (Coffee, 1997) are the counterpart to Microsoft's Component Object Model (COM). They represent platform independent, reusable software components, being composed of Java classes and "conforming to a particular convention. They are used to encapsulate many objects into a single object (the bean), so that the bean can be passed around rather than the individual objects". They were introduced to facilitate the development of components with graphical builder tools. As they do neither have to extend any superclass nor have to implement any interface, they are Plain Old Java Objects (POJOs) which have to stick to naming, construction and behavioural conventions.

4.4.3. Java Servlet

In the Java world, Servlets are the counterpart to other concepts generating dynamic web content (PHP, ASP, etc.). Being implemented in Java, its advantage is to be platform independent in contrast to ASP, which relies on Microsoft Windows and the Internet Information Service and to be able to access any available Java API.

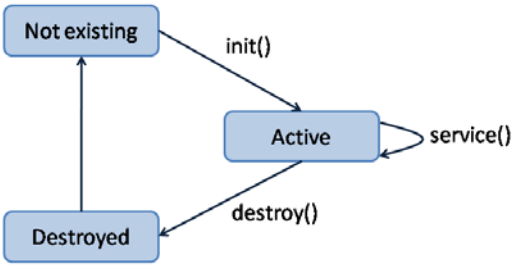

Figure 6: Servlet lifecycle

Servlets are highly integrated in a Servlet Container, which comes with the web server and keeps control over the Servlet life cycle and passing service requests to them.
The Servlet life cycle described in Figure 6 is summarized in short:

1. The Servlet class is loaded by the container during start-up.
2. The container calls the init() method. This method initializes the Servlet and must be called before the Servlet can service any requests. In the entire life of a Servlet, the init method is called only once.
3. After initialization, the Servlet can service client-requests. Each request is serviced in its own separate thread. The container calls the service() method of the Servlet for every request. The service() method determines the kind of HTTP request (GET, POST etc) and accordingly calls the methods doGet(), doPost(), doTrace() etc. The developer of the Servlet must provide implementation for these methods. If an implementation for doPost() has not been provided, it means that the Servlet cannot handle POST requests. In such a situation if a POST request is made, the implementation of the parent class will be invoked. By default, this will throw a BAD HTTP Request exception which will result in a "HTTP 400 - Bad Request" response to the client.
4. Finally, the container calls the destroy() method which takes the Servlet out of service. The destroy() method like init() is called only once in the life-cycle of a Servlet. After the Servlet is destroyed, it will be garbage collected and does not exist anymore.

To access client and request specific data and to be aware of the global context, every Servlet obtains a global and unique ServletContext and a ServletConfig object. The former object can be used by all the Servlets to get application level information or container details whereas the ServletConfig object provides individual initialization parameters (client data, cookies, request data, etc.) for every Servlet.

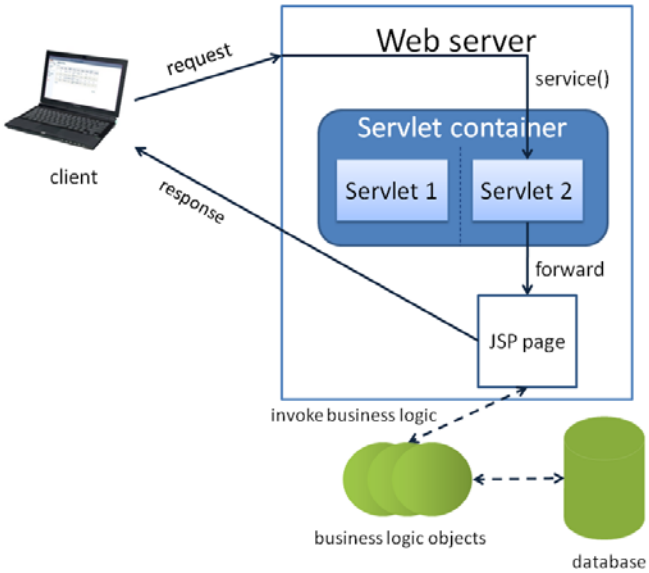

Figure 7: Model 2 architecture with Servlets

Most commonly, the Servlet does not directly create the response but acts as a controller, delegating the rendering to a JSP page which itself calls some business logic in terms of (Enterprise) JavaBeans methods. This architecture is

called Model 2[1], a flavor of the MVC design pattern which separates content from view (see Figure 7). It is propagated among others by the well known Struts (Holmes, 2008) framework.

4.4.4. JavaServer Pages

As the Servlet API provides only low level support for developers, accessing request and response objects, cookies, session and context data, the demand for a higher level technology came up soon. Sun replied with releasing JavaServer Pages[2] (JSP), an abstraction of Servlets by extending the Servlet API. JSP is compiled into Servlet code, a byte code representing the Servlet. JSP pages can consist of static HTML tags combined with JSP specific tag directives, scripting elements, JSP actions and custom tags defined in the corresponding taglib header(s).
The directives allow for including other JSP page fragments which is useful for building headers, footers, menus or other repeating page items. These fragments generally have the extension "jspf".

```
<%@ include file="header.jspf" %>
```

The page directive supports importing of Java packages and setting page specific parameters (e.g. contentType, language, pageEncoding, etc.). To include custom tag libraries under a given prefix, the taglib directive was introduced. To show the number of requests for a specific page, one could import a custom tag library which is located in "taglib/myTags.tld", publishing a counter element and include this element in the JSP page:

```
<%@ taglib prefix="myTag" uri="taglib/myTags.tld" %>
<html><head></head><body>
<!-- cut -->
<myTag:counter />
<!-- cut -->
</body>
</html>
```

Whenever scripting is used inside the JSP page, it must be surrounded with the "<%" and "%>" expressions. Inside these expressions, Java code is located which can reference predefined implicit objects including the JSPWriter out, which writes to the response, and others, representing the session, request, response, Servlet configuration, etc.
The following example uses the implicit objects request and out to print the header of the current request to the JSPWriter:

```
<h3>request</h3>
<%
  Enumeration enumeration = request.getHeaderNames();
    while (enumeration.hasMoreElements()) {
      String key = (String)enumeration.nextElement();
      out.println (request.getHeader(key));
    }
%>
```

The JSP specification[3] also defines some standard actions in the form of JSP tags for including other JSP pages, modifying request parameters, forwarding, and for reading and writing bean properties.
What makes JSP very powerful is the possibility of extending the predefined actions with custom ones. Developers can write collections of custom actions which are organized in so called tag libraries. To develop a custom tag, they have to develop common Java classes implementing one of the Tag interfaces of the JSP Tag Extension API. To tell the JSP compiler what to do when one of the custom tags is to be handled, a XML tag library description (TLD) file which maps the tags to the class implementing it, must be provided within the library.
That is what Sun has done to extend the built-in actions of JSP. Sun published the JSP Standard Tag Library (JSTL) (Bayern, 2002) containing additional functions, support for database access, internationalization, XML processing and flow control.
Since version 2.0 of JSP, the specification defines an Expression Language (EL) which is included in JSP. The EL, which should make the JSP scriplets obsolete, facilitates view development as detailed knowledge of the Java language is no more required. It was designed to easily access application data contained in JavaBeans components: A JavaBean with the name "cart" can be accessed with the statement ${cart}, the nested list-property items (with public getter getItems()) can be accessed with ${cart.items} and the first element is ${cart.items[0]}. To displace scriplets completely, the EL also provides implicit objects similar to scriplets and allows the usage of common

[1] Model 2: http://en.wikipedia.org/wiki/Model_2
[2] JavaServer Pages: http://java.sun.com/products/jsp/
[3] JSP Specification: http://java.sun.com/products/jsp/

logical and mathematical operators as known from Java. Another very interesting feature of the EL is the possibility of calling external, bean independent functions which have to be imported with the taglib directive. With the EL, page authors do no more have to be aware of the Java programming language used in scriplets within the JSP code but can concentrate on their job of realizing the design-specific aspects with JSP. On the other hand, the code is better maintainable because the program logic formerly embedded in the page as scriplets can now be encapsulated with JavaBeans and be called from any EL statement on the pages. For a more detailed description of the EL see the JSP 2.0 specification[1].

4.4.5. Java Persistence API (JPA)

The JPA (Maki, 2007) is a Java standard to access and persist Java objects to relational databases. It should combine the best ideas from the most common persistence frameworks (Hibernate[2], TopLink[3], Java Data Objects[4], etc.) and create a practical, easy to use API for Java developers. It is part of the EJB 3.0 specification and replaces the former EJB 2 Container Managed Persistence (CMP) Entity Beans specification. Unlike the EJB 2 CMP, the JPA managed objects are POJOs which do not have to implement any interfaces or methods.
The standard consists of three parts: The Java Persistence API (JPA), contained in the javax.persistence package, the Java Persistence Query Language (JPQL) and object/relational metadata. The objects managed by the Persistence API are called Entities. An entity is a lightweight persistence domain object which represents a table in a relational database. It is implemented as POJO whose class must be annotated with the javax.persistence.Entity annotation. Each Entity instance represents one row in the corresponding database table with each of its properties mapped to a database field. If an entity has a relationship with another entity, the type of relationship must be specified either using annotations or in a XML descriptor file supplied with the application. Besides, the JPA provides several annotations for configuring the database fields. The following example will clarify this procedure. Example 1 demonstrates how two POJOs, Address and Person, can be annotated to be managed within the JPA. As mentioned before, they have to be annotated with @Entity to indicate being part of the Persistence Unit, which holds all the entities of the application.

```
@Entity
public class Address implements Serializable {

  @Id
  @GeneratedValue(generator=GeneratorType.AUTO)
  private Long id;

  @Column(length=64, nullable=false)
  private String addressLine1;

  @Column(length=64)
  private String addressLine2;

  @Column(length=6)
  private String postalCode;

  private String city;

  // default constructor
  public Address() {}

  // getter and setter methods
  // equals and hashCode method
}
```

Example 1: Entity - Persistence annotations

[1] The JSP 2.0 specification: http://jcp.org/aboutJava/communityprocess/final/jsr152/
[2] Hibernate: http://www.hibernate.org/
[3] TopLink: http://www.oracle.com/technology/products/ias/toplink/index.html
[4] Java Data Objects: http://java.sun.com/jdo/

```
@Entity
public class Person implements Serializable {

  @Id
  @GeneratedValue(generator=GeneratorType.AUTO)
  private Long id;

  @ManyToOne
  private Address address;

  @Temporal(type=TemporalType.DATE)
  private Date dayOfBirth;

  private String telNr;

  public Person() {}

  // getter and setter methods
  // equals and hashCode method
}
```

Every entity has to declare one identifier field which is required for the database management system to join several tables and to distinguish between entities of the same type. This field must be annotated with @Id. Optionally, the @GeneratedValue annotation can be used to let the generation of the id value be handled by the runtime system. The @Column annotation can be used to configure field specific data like the maximum length of Strings or whether null values are allowed or not.
When there is a relationship between entities, its type must be declared. In the example, there is a n:1 (many-to-one) relationship between Person and Address indicating that one person can have only one address but one address can be owned by many people. This information is necessary for the persistence engine to map the entities correctly to the database. In the case of this unidirectional many to one relationship, the persistence engine would by default add a reference field to the person's database table, referencing the id of the person's address (see Figure 8). Other available relationships are bidirectional many-to-one and uni- or bidirectional one-to-one or many-to-many relations.

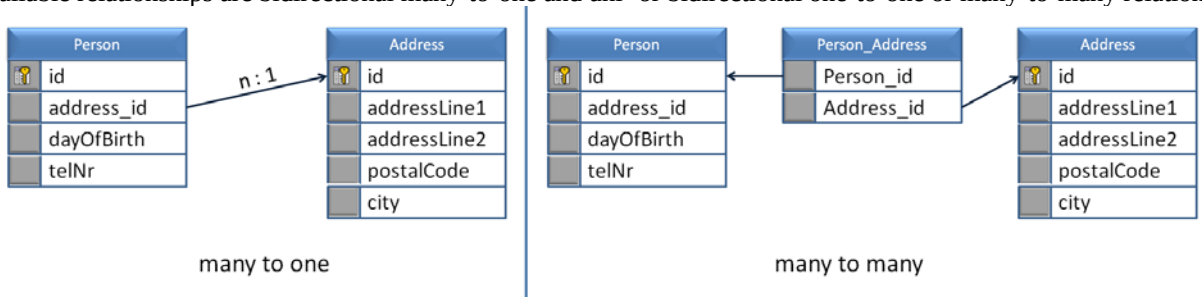

Figure 8: Persistence Context - Entity relationships

If one would allow one person having more than one address, the relationship would be many-to-many and would have to be annotated with @ManyToMany. The persistence engine would then create a join table by default, mapping the address items to person items (see Figure 8).
The JPA also provides an EntityManager which can be used to store, update, merge and find entities by their ID in the database.
During its life-time, an entity can have several states (see Figure 9): New, Managed, Detached and Removed. When a new entity is created, it automatically has the "New" state. Persisting an entity leads to the "Managed" state. It is then controlled by the Entity Manager. In this state, a change on the object's property, mapped to a database field will automatically cause the change of the corresponding database value. Sending the managed entity back to the client will result in its detachment. Changing properties of the object in the detached state will have no immediate effect on the database. To achieve this, the entity must be transferred to the managed state again. This is done by merging the entity with the Entity Manager. After removing an entity in the managed state, it becomes "Removed". This state can be left again by persisting the removed entity.

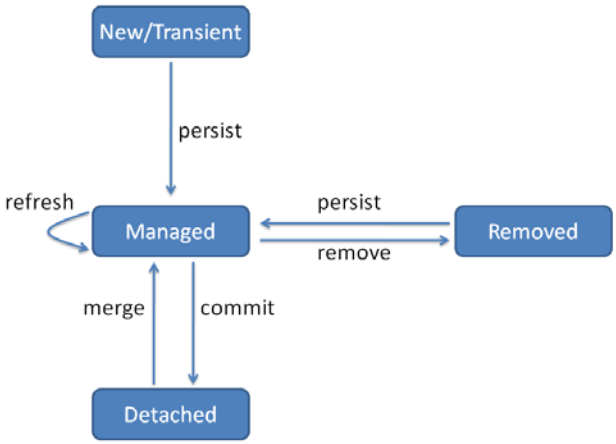

Figure 9: Persistence Context - Entity state transitions

4.4.6. JavaServer Faces (JSF)

According to Sun, JavaServer Faces is the "new standard" in web application view development. It is a component based view development framework for Java technology-based web applications which should supersede Apache's Struts technology. JSF includes a specification (Burns & Kitain, 2006) and a reference implementation. Publishing the specification should allow other competitors to develop their own implementation. With Apache MyFaces, Apache has published an alternative implementation which provides many additional useful components including calendars, color chooser, breadcrumbs, navigation panes, and much more[1]. Unfortunately, Apache MyFaces often lacks behind the current version of the specification because it takes some time until a new version of the specification is implemented.

From the technical point of view, JavaServer Faces is an abstraction from the Servlet API (see Figure 10). It is built on top of the Servlet API and keeps the developer from dealing with the low level Servlet API and HTML code. In fact, the developer has to handle with components that provide much functionality like browsing, sorting, validating, etc. out of the box and can be rendered to HTML code.

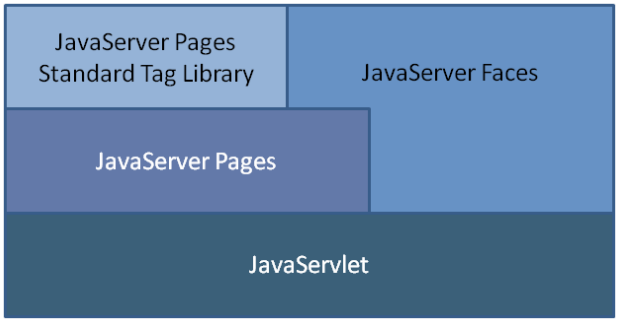

Figure 10: Java EE - View technology overview

As JSF components are embedded into a JSP page, a JSF page is built like a JSP page, enriched with JSF components declared in a taglib directive (see Example 2) at the top of the page. A JSF component always represents a specific view element. Basic components which are included in the JSF reference implementation include input elements, buttons, labels, tables and others. Thus, JSF extends the capabilities of JSP with a view component model. This has among others the following benefits:

- Using a visual builder tool, components can be added and arranged on the page per drag and drop.
- Components and component generated events can be easily linked with server-side application code without having specific programming knowledge in Java and JSP.
- Component properties can be mapped to JavaBeans properties and thus be automatically updated when the model is changed.
- Components can be reused within several views.

[1] Apache MyFaces project page: http://myfaces.apache.org/

- The state of components is maintained during several requests. Thus, the developer must not maintain component state manually and compare the properties on the next POST request to check whether the user has done some changes.
- Rendering of the components is not part of the component itself but is done by a separate renderer, allowing generating different output than HTML easily.
- Strict separation of view components and the model which commonly consists of backing beans. This accelerates development as view and model can be constructed by different parties whereas each needs not to know the technical details of the other.

4.4.6.1. View development

As view development is a bit different compared to JSP, it is discussed here in short. The JavaServer Faces comes with two JSP tag libraries for expressing server-side UI components and custom logic within the page. The HTML component library[1] which contains basic JSF view components representing common HTML elements like tables, option panes, dropdown boxes, buttons, links, image and form elements, etc. and the JSF core library[2] which provides JSF-specific action tags including action listeners, validators, converters, and some more. The following example will demonstrate the usage of the tag libraries and show how the view interacts with the controllers. Figure 11 shows a simple login screen where user name and password have to be entered and a submit button can be activated to send the login information to the server-side application where the information is evaluated. Example 2 shows the corresponding JSF page which is rendered to the login screen.

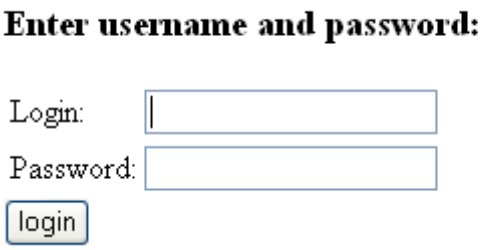

Figure 11: JSF - Sample login screen

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<html>
  <head>
    <title>Login</title>
  </head>
  <body>
    <f:view>
      <h3>
        <h:outputText value="Enter username and password:"/>
      </h3>
      <h:form>
        <h:panelGrid columns="2">
          <h:outputText value="Login:"/>
          <h:inputText value="#{loginBean.userName}" />
          <h:outputText value="Password:"/>
          <h:inputText value="#{loginBean.password}" />
        </h:panelGrid>
        <h:commandButton action="#{loginBean.submit}" value="login" />
      </h:form>
    </f:view>
  </body>
</html>
```

Example 2: JSF - View with PanelGrid component

The JSF page displayed in Example 2 contains two taglib directives (see JSP taglib directives in Section 4.4.4) at the beginning of the page, which enable the usage of the JSF action and component tags.
Since JSF does not provide any components representing the <head> section of a HTML page, this part must be created, when needed, in HTML manner, using the appropriate tags.

[1] JSF component tag-library: http://java.sun.com/jsf/html
[2] JSF core tag-library: http://java.sun.com/jsf/core

Example 2 shows the JSF page with a PanelGrid component. The panelGrid component is rendered by default to a HTML table with the given number of columns. In the example, the panelGrid renderer will automatically create a new row after every nested even-numbered component. The resulting page is displayed in Figure 11: JSF - Sample login screen.
The JSF components are added to the body of the HTML page. It is important to put all the JSF components inside the <f:view> core tag which must be the root of all JSF components. The <h:form> element represents a HMTL form which must be used to send the dynamic form data to the server. Thus, all editable components must be nested inside this tag. To submit the page data, an action component (CommandButton or Link) is used. When activating the action component by clicking on it, the form data are sent to the server application using a HTTP PUT request.
To interact with a Managed Bean (see section 4.4.6.2) which represents the controller, the Unified Expression Language (see section 4.4.6.4) is used. It allows for reading and writing bean properties and calling action methods on a Managed Bean. The expressions are nested inside the "#{" and "}" delimiters. In Example 2, the expressions are used to access the properties *username* and *password* of the managed Bean with the name *loginBean* and to call the submit method of this bean when the submit CommandButton is activated.

4.4.6.2. JSF Managed Beans

The counter-part of the view components is the model which commonly consists of managed beans (Jendrock, 2007) which are typically associated with the UI components used in a particular page. Managed beans are regular JavaBeans with properties and event listeners with the following differences: They are managed by the container and therefore can accept resource injections with container resources including other Managed Beans, DataSources, EJB and Webservice references, EntityManagers and other container managed objects. To use the dependency injection mechanism, the Managed Bean has to declare a dependency to one or more resources using annotations. When the Bean is initialized by the container, its dependencies are resolved and the container will inject the appropriate resource(s) into the Bean. Example 3 shows an abstract Database Access Object which requests an EntityManager object which is used to access the database, to be injected on its initialization.

```
public abstract class AbstractDAO {

   @PersistenceContext
   private EntityManager em;

   public EntityManager getEm() {
      return emf;
   }
   public void setEmf(EntityManager em) {
      this.emf = emf;
   }
}
```

Example 3: Managed Beans - Resource injection

This injection mechanism is an advantage to common JavaBeans which are not container managed and thus cannot accept these injections. As the resource injection information which is represented by annotations must be available at deployment time, JSP pages cannot accept resource injections, too, because they are compiled after deploying the application.
Managed beans can also be initialized with preconfigured parameters or references to other managed beans (see Example 4).
The properties of the bean can be bound to a component's value, a component, converter, listener or validator instance. Its methods are commonly used for validating component data, handling a component event and executing logic to perform the page to which the application must navigate next.
To be clearly identified from inside the view, each bean has a unique name under which it can be accessed from the view.
The beans are created on demand by the container and can be activated and deactivated depending on the scope. Possible scopes are: *none*, *request*, *session*, *application*. The none scope indicates that these beans cannot be accessed from anywhere of the application but inside the configuration file itself. Managed beans with application scope endure the whole time, the application and server is running. Session and request scoped beans are destroyed when the processing of the session or request has terminated.
Managed Beans are configured inside the global configuration file of JSF (commonly *faces-config.xml*). The following items of information are mandatory: A String identifier representing the name of the bean under which it can be accessed from inside the view, a Java class name, and a scope for the bean. The description and configuration

of bean properties which can be used to initialize bean properties with custom values, is optional. Example 4 shows the configuration of the loginBean which was introduced in section 4.4.6.1 to authenticate users.

```
<managed-bean>
   <description>Login Bean: Used for user authentification</description>
   <managed-bean-name>loginBean</managed-bean-name>
   <managed-bean-class>controller.UserController</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>userDAO</property-name>
      <value>#{userDAO}</value>
   </managed-property>
</managed-bean>
```

Example 4: JSF - Managed Bean configuration

The loginBean's userDAO property is configured to be initialized with the userDAO bean which is another managed bean that represents a Data Access Object for user and login data. The userDAO bean is retrieved by evaluating the #{userDAO} expression (see 4.4.6.4 Unified Expression language). For more on Managed Beans see (Jendrock, 2007).

4.4.6.3. Filter

Filters are commonly used in JSF to manipulate the request and response objects during the request processing and dispatching process. They usually do not themselves create the response like components do, but provide additional functionality (e.g. data compression, encryption, authentication, etc.) which can be attached to any kind of web resource which is accessed through a servlet. Besides the request and response object, a filter has access to external resources including the servlet and faces context (if JSF is used). To activate a filter, it must be set up in the deployment descriptor of the web application. There, it is possible to define initial parameters which are passed to the filter when it is instantiated. A web resource can be filtered by zero or more filters which are organized in a so-called filter-chain. Every filter in the chain can decide whether to go on processing of the following chain or to abort it. For more information on filters, please refer to (Jendrock, 2007).

4.4.6.4. Unified Expression language

As JSF should not lack the possibilities of an Expression Language known from JSP 2.0 (see section 4.4.4), its creators decided to develop a custom EL which should support the extended features of JSF. With the introduction of Java EE 5 which should include both, JSP and JSF, the Java EE authors had to face with the problem to have two incompatible Expression Languages together on one page[1] and thus be processed by the same JSP engine. For this reason, they developed the new Unified EL (Srinivasan, 2006) which is used by JSP (since version 2.1) and JSF.
Like the JSP EL, the main goal of the Unified EL is to ease view development by providing an easy way to access the managed beans. It is a simple language used for accessing Managed Beans, implicit objects including the request, response, session, page and application context, and for manipulating collections in an elegant manner.
One of the new features of the Unified EL is the introduction of deferred expression evaluation. In contrast to the immediately evaluated expressions which take the form ${immediateExpr} and are well known from the JSP EL, deferred evaluated expressions are not directly evaluated by the JSP engine but must be handled by the superordinate technology which in our case is JSF. JSF will then decide when to evaluate the expression in the page lifecycle. Deferred expressions take the form #{deferredExpr}.
The former JSP EL only provided expressions which could be used to set and read JavaBeans properties. In addition to these value expressions, the Unified EL introduced method expressions which are used to invoke methods. They are commonly used to handle component events and for validating component data. The current type of the expression is dependent of the context. Here is an example for a deferred value expression which sets the value of an input field to the name property of the person bean:

```
<h:inputText id="name" value="#{person.name}"/>
```

The following JSF representation of an HTML input form tag uses a method expression to validate the telephone number of a person:

[1] JSF and JSP tags can be mixed up.

```
<h:inputText id="tel" value="#{person.tel}" validator="#{person.validateTel}"/>
```

The Unified EL also supports using JavaServer Pages Standard Tag Library (see section 4.4.4 JavaServer Pages) iteration tags with deferred expressions. This allows the use of deferred expressions inside of JSTL iteration tags like:

```
<c:foreach items="#{cart.items}" var="item">
  // print item data
</c:foreach>
```

4.4.6.5. Tag handlers and component trees

When JSF receives the first request for the page, the component tree for the view represented by the page doesn't exist. JSF creates a tree with just a UIViewRoot at the top and forwards the request to the JSP page to add the real components. The JSP container processes the page and invokes the JSF action tag handlers as they are encountered. A JSF tag handler looks for the JSF component it represents in the component tree. If it can't find the component, it creates it and adds it to the component tree. It then asks the component to render itself. This means that on the first request, the components are created and rendered subsequently. On subsequent requests, the component tree exists, so no new components are created; the tag handlers just ask the existing components to render themselves. This is necessary because components are responsible for preserving their state during successive requests. If the components would be simply created on each request, the previous state would be lost and the component would no more be able to reproduce any changes made by the user.

4.4.6.6. Converters

When rendering the result page and rebuilding the component tree on an incoming POST request, model objects must be converted from and to the output format. Therefore converters are required to convert objects to their String representation and backwards.
In most of the cases, converting is done automatically by the JSF framework: When the user sends form data to the application, which are bound to a bean property of one of the supported datatypes[1], JSF will convert these values to the target datatypes of the bean properties they are bound to, using a standard converter. This invisible conversion will not work anymore when component's values are bound to custom objects because JSF cannot know how to convert them to a good and human readable String representation and back again. In this case, a custom converter has to be registered or declared inside the component. This converter will then be called for every conversion of the custom object to its unique string representation, used as the component's value and back again. The following scenario will demonstrate this:
A person who wants to place an order at an online shop has to enter his credit card number. When the person has already entered his credit card information with an earlier order and thus, the data are available in the database, the credit card object can be created from the given credit card number using a custom CreditCardConverter:

```
<h:inputText id="creditCard" value="{user.creditCard}" converter="CreditCardConverter"/>
```

```
public class CreditCardConverter implements Converter {

  public Object getAsObject(FacesContext context,
                            UIComponent component, String creditCardNr) throws ConverterException {
    String convertedValue = null;
    if (creditCardNr == null ) {
      return null;
    }
    Long cardNumber = Long.parseLong(creditCardNr);
    // fetch CreditCard object from data store:
    return entityManager.find(CreditCard.class, cardNumber);
  }
}
```

When the request containing the inputText component with the credit card number is processed from the JSF implementation, it looks up the component's local value and calls the getAsObject method with the FacesContext instance, the component whose value has to be converted and the value to be converted as parameters. The method then looks for a card with the same number in the data store and returns the found object.

[1] Standard converters exist for: (Big)Decimal, (Big)Integer, Boolean, Byte, Date(Time), Double, Float, Long, Number, Short

To define how the data is converted from the model view to the presentation view, the Converter implementation must implement the getAsString method from the Converter interface. Here is the implementation of this method:

```
public String getAsString(FacesContext context, UIComponent component,
                          Object inputObject) throws ConverterException {
  CreditCard creditCard = null;
  if (inputObject == null ) {
    return null;
  }
  // creditCard must be of the type that can be cast to CreditCard.
  try {
    creditCard = (CreditCard) value;
  } catch (ClassCastException ce) {
    throw new ConverterException("Cannot convert CreditCard: Wrong type: " + inputObject);
  }
  String convertedValue = creditCard.getNumber();
  return convertedValue;
}
```

To be found and used by the JSF framework, custom converters must be registered in the global faces configuration file. The configuration entry for the previous credit card object converter would look like this:

```
<converter>
  <description>
    Converter for credit card numbers that normalizes the input to a standard format
  </description>
  <converter-id>CreditCardConverter</converter-id>
  <converter-class>
    de.lmu.ifi.ticket-billing.converter.CreditCardConverter
  </converter-class>
</converter>
```

The converter-id element specifies the name which has to be used in the converter attribute of a JSF input component which wants to use the converter.

4.4.6.7. Event and listener model

JSF differs between component and lifecycle events. Component events are fired when the state of a component was changed during subsequent requests. The moment of the firing is defined in the request processing lifecycle (see Figure 12). The events can be caught and processed nearly anywhere in the model. The specification defines two types of component events: Value-change events and action events. Value-change events are the result of changing a value of a UI component[1] and action events that are fired when the user has activated an UI component representing a link or button. To handle these events, a value-change listener or action listener has to be registered on the component. This event listening model is similar to the listening model of desktop applications, although the events are not fired immediately when the user invokes an action (selecting an element of a dropdown box or clicking a button, etc.) but after the whole page is pushed to the server, the component tree is rebuilded and the components have restored their values.
The JSF implementation fires lifecycle events on every change of the Servlet state (see Figure 6). Furthermore, listeners can catch session and servlet context attribute events. Lifecycle event listeners which are registered on a Life Cycle phase can handle the further processing (e.g. update model or render response). Thus, they can short-circuit the request processing lifecycle when the following phases do not have to be processed anymore (e.g. when an early exception occurs, the processing can jump directly to the render response phase where the result page is created).

4.4.6.8. Request Life Cycle

Unlike JSP, where the request processing is done in one phase, JSF handles HTTP requests with six distinct phases (see Figure 12).

[1] Input components: UIInput, UISelectOne, UISelectMany, and UISelectBoolean

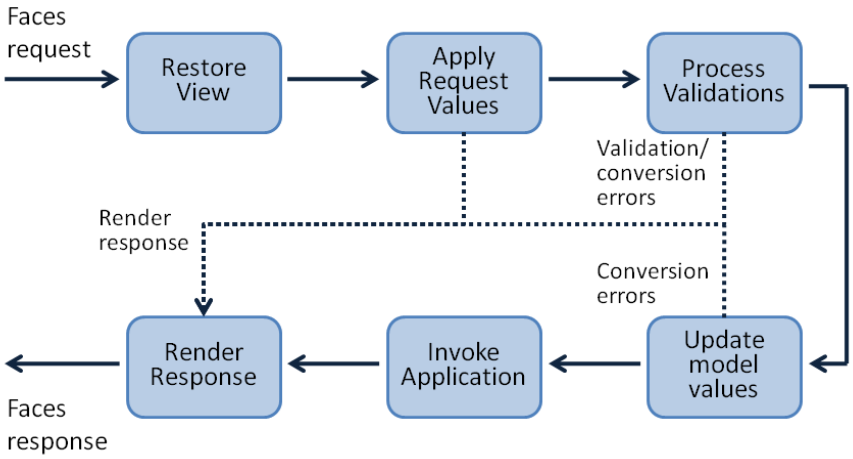

Figure 12: JSF - Request processing life-cycle

This is necessary because the JSF UI component model is considerably more complex than the JSP model. Conversion and validation of component data, event handling and propagating component data into backing beans must be handled in an orderly fashion which is done by the following phases:

1. **Restore View**. Upon receiving a faces request, the JSF engine retrieves the faces context instance associated with this user session. The faces context instance contains a view, that is to say a collection of UI components that represents the client-user interface.
2. **Apply Request Values**. In this phase, the states of the UI components are synchronized with the client-user interface. This is done by reading the appropriate data from the HTTP request, such as headers and parameters. As part of the synchronization, events may be triggered, and the registered event handlers will be invoked. This enables server-side, deferred event handling for client-generated events.
3. **Process Validations**. After setting the states of the UI components, validator instances registered with each UI component are triggered. Errors are immediately reported back to the client.
4. **Update Model Values**. The purpose of this phase is to update the application data. If the value of an input field is bound to a bean property, in this phase, the value of the bean's field is set according to the user's input.
5. **Invoke Application**. The invoke application phase processes the action submitted by the request, by invoking the JavaBean method associated with the submitted action in terms of a method expression. By this, the application logic is executed.
6. **Render Response**. This is the final phase in the faces request processing life-cycle. To identify the result page which will be returned, the navigation rules are evaluated. They define the view to which the request will be forwarded, depending on the navigation outcome defined directly on the page or returned by an evaluated method expression. For the resulting page, the UI components are then rendered into a format which is expected by the client, such as an HTML document.

The JavaServer Faces Request Lifecycle is a very flexible approach with hooks in the right places for inserting custom request handling logic in an elegant way: At each point after the second phase, it is possible to short-circuit the process and move directly to the Render Response phase or even to output a response directly and notify the JSF runtime that response handling is complete.

4.4.6.9. Navigation

In JSF, it is not possible to directly forward from one view to another like the invocation of an ordinary HTML link (next) would do. Instead, JSF propagates a declarative navigation model with navigation outcomes and navigation rules. The navigation rules have to be configured in the global configuration file (commonly faces-config.xml). Each rule specifies a navigation outcome and a view id (optional) and contains the destination view id. The navigation outcome is a String, representing the logical result of evaluating a specific view. For a login view, where the user has to enter his user name and password, one could imagine two navigation outcomes: SUCCESS and FAILURE. The view evaluates to SUCCESS when user name and password were entered correctly and to FAILURE if not. In the first case, the user would be forwarded to the main view (main.jsp) and otherwise he would have to try again (login.jsp). The following navigation rule represents this use case:

```
<navigation-rule>
  <from-view-id>/login.jsp</from-view-id>
  <navigation-case>
    <from-outcome>SUCCESS</from-outcome>
    <to-view-id>/main.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>FAILURE</from-outcome>
    <to-view-id>/login.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

The navigation outcome of the view is commonly generated by an action component (UIButton or UILink) which contains an action attribute. The value of this attribute can either be the navigation outcome String directly (static navigation) or a method expression which evaluates to a navigation outcome (dynamic navigation).
In both cases, JSF will pass this outcome to the default navigation handler instance which looks for the appropriate navigation rule and returns the resulting view id.
Considering the upper navigation rule, activating the link of the following example in the /login.jsp view will directly forward the page flow to the /main.jsp page.

```
<h:commandLink value="Submit" action="SUCCESS"/>
```

If some logic has to be executed to determine which page has to be returned, the dynamic navigation is the best choice. The submit action method of the login bean is evaluated to return the navigation outcome. If the user is authorized, it returns “SUCCESS”, otherwise “FAILURE”. If so, the navigation handler will redisplay the /login.jsp page as configured in the navigation rule.

```
<h:commandLink value="Submit" action="#{loginBean.submit}"/>
```

The corresponding login bean:

```
public String submit() {
  if(authorize(user)) {
    return "SUCCESS";
  } else {
    return "FAILURE";
  }
}
```

The declarative navigation model of JSF makes it easier for tools to define navigation rules in a global business view.

4.4.6.10. Security

One of the advantages of writing a Java EE compliant web application is the ease of setting up security. Java EE comes with a useful built-in security feature which should be applicable for the majority of web applications. The Java EE architects have attached much importance to the ability of setting up security independent from the application code, framework or technology itself. This means that most of the security constraints are not coupled with the source code but can be configured in the applications deployment descriptor. This facilitates the administration of the web application and allows for changing security constraints without recompiling the application.

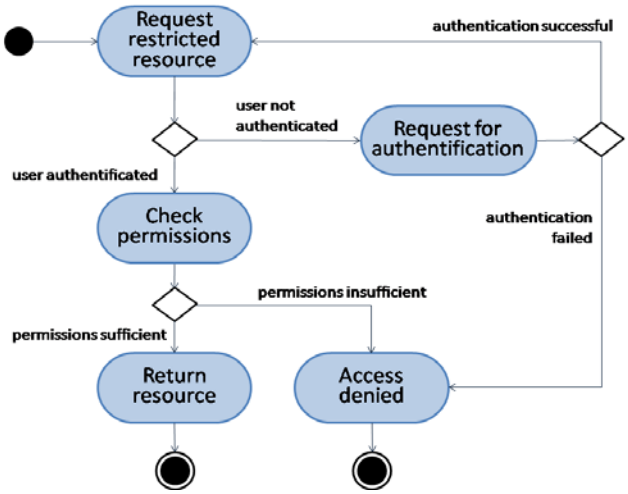

Figure 13: Java EE - Authentication and authorization

1. Setting up security for a web application consists of the following three parts:
2. Setting up security roles
3. Setting up security constraints

Defining the authentication method

Commonly there are different privileges for different user groups accessing an application. Most often, there is one administrator group with unlimited privileges and several user groups with restricted access, but other combinations are imaginable just as well and depend on the use-case itself. Those roles are represented in Java EE by security roles, whereas each of them has to be defined in the deployment descriptor (1). The next step to a secure web application is the definition of security constraints (2) defining one or more resources which have to be protected and the declaration of the security roles which are allowed to access the resource(s). The resources are now protected but the login procedure still has to be configured. Java EE handles requests to protected resources as follows: If a restricted resource is requested and the user is not yet authenticated, he will be asked for authentication. If the authentication succeeded, the user gets redirected to the originally requested resource where his authorization is verified. If the authorization fails, the access to the resource is denied, otherwise it is granted. Figure 13 shows an activity diagram which illustrates the complete authentication and authorization process. In any case, the administrator has to define the authentication method (3). This can be either a custom login form or a simple JavaScript Pop-up prompting for username and password.

Here is the security setup of the example application:

```
<security-constraint>
   <display-name>bookkeeper area</display-name>
   <web-resource-collection>
      <web-resource-name/>
      <url-pattern>/faces/bookkeeper/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>bookkeeper</role-name>
      <role-name>admin</role-name>
   </auth-constraint>
</security-constraint>
```

The upper security-constraint protects all resources located under /faces/bookkeeper/ to be accessed only by users owning security role bookkeeper or admin.

The example application uses the BASIC and built-in JavaScript solution as authentication method:

```
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

4.4.7. Facelets

There is one major disadvantage of JSF: it does not provide any templating facilities. This is very unsatisfactory for larger projects as view elements occurring in more than one page have to be duplicated and changes of the element will always result in changing several pages. This is a very bad and error prone approach and has to be avoided.

Using the JSP include directive to embed external JSP fragments (see section 4.4.4 JavaServer Pages) into the page at first glance solves the problem but will lead to other considerable problems: Although JSF requires JSP for its component model, both technologies do not work together very well. This is based on the very different intentions which underlie to the design of the two technologies. Whereas JSP is singularly focused on creating dynamic output, JSF was introduced to rapidly develop web applications with its component model requesting for a request response life-cycle with six distinct phases. The problem follows from the fact that JSF component values and actions are not evaluated immediately (like JSP does) but at a later moment during one of the phases of the life-cycle (see Figure 9: JSF - Request processing life-cycle) and that this deferred evaluation is contradictory to the JSP philosophy. For more on this topic see (Bergsten, 2004).
The facelets technology[1] promises to be the solution of the problem. Like the tiles library[2] for Struts, it supports templates and is geared toward the JSF component model. Using facelets within a JSF web application requires registering the facelet view handler in the JSF configuration file (faces-config.xml), setting the filename extension for files to be handled by the view handler and to declare the facelet tag library in the pages using facelet tags.
A facelet template commonly consists of static HTML code defining the layout of the template and should contain at least one insertion point. Insertion points must have unique names within one template and can be regarded as slots where the dynamic content can be inserted when the template is used. A slot can optionally contain a default value which can but does not have to be overridden. Example 5 shows an example template with the two slots title and body. It simply defines the default title and includes a page header. The body slot does not contain a default value.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<head>
   <title>
      <ui:insert name="title">Ticket Billing System</ui:insert>
   </title>
   </head>
   <body>
      <div id="header">
         <ui:include src="/WEB-INF/jspf/template/header.jspf"/>
      </div>
      <div id="body">
         <ui:insert name="body"/>
      </div>
   </body>
</html>
```

Example 5: Facelets - View Template

To use a template inside a page, the page has to define a composition referencing the appropriate template. The composition in turn defines the dynamic content which is plugged into the slots of the template. Example 6 represents a composition which uses the template of the previous example (located in /layout/template.jsp). It sets the page title to “Ticket Billing – Bookkeeper” and includes the /company/list.jsp page fragment into the body slot.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets">
   <body>
      <ui:composition template="/layout/template.jsp">
         <ui:define name="title">
            Ticket Billing - Bookkeeper
         </ui:define>
         <ui:define name="body">
            <ui:include src=”/company/list.jsp"/>
         </ui:define>
      </ui:composition>
   </body>
</html>
```

Example 6: Facelets - View Composition

When rendering a page containing a composition tag, the facelet view handler will ignore the elements around this tag.

[1] Facelets on java.net: https://facelets.dev.java.net/
[2] Apache Tiles project page: http://tiles.apache.org/

The facelets library supports much more interesting features, including custom logic tags, expression functions, the creation of component libraries, and much more which are not explained here because it would go beyond the scope of this work.

4.5. Testing

As quality management has established in modern software development, writing tests for an application has become a fixed part in ensuring software quality. Unfortunately, Sun's J2EE specifications do not pay much tribute to this fact: Quality management in J2EE applications is mentioned rarely and possibilities and hints for testing J2EE web application are not available at all. In contrast, the ability of testing has been a main goal of developing Microsoft's .NET framework. This resulted in many available testing tools for automatic test generation etc. being integrated in Visual Studio, Microsoft's software development IDE. Anyway, the possibilities for writing tests have been evaluated for the ticket billing system. As expected, the evaluation was not successful and did not result in a good way of testing the ticket billing web application. Two approaches were regarded to test the application:

1. Structural and functional tests of parts of the system (business logic, etc.)
2. Testing the system as a whole.

The first approach turned out not to be applicable mainly due to the following two reasons: It does not imply to run the tests on parts of the deployed system as this is not possible but rather to create instances of the controllers holding the functionality to test, to run the appropriate methods with different arguments and to check the system's state for correctness afterwards. The problem is that the controllers use services depending on the application server or JSF framework, including the lookup of managed beans or dependency injection, for every more complex business case. As these services are not available and it would have resulted into much effort to introduce a mechanism relying on mock objects instead of the services, this approach was not considered. The second approach should avoid the described problem with the difficult implementation of stubs: It requires the application to be deployed and emulates the user's interactions with the web interface. More precisely, the test driver must generate a HTTP request and has to compare the HTTP response with the requested response. Besides, the state changes of the managed beans have to be checked. Whereas the latter would be possible by requesting the current FacesContext instance which provides access to the managed beans, the implementation of the drivers and the parsing of the resulting documents would have been very complex. Therefore, this approach for testing was discarded, too.

5. Conclusion

Intention of this work was not to evaluate all technologies which could be used for web application development and to decide which one is the best but to provide a deeper understanding of Sun’s Java Enterprise Edition with focus on view technologies, mainly the component based JavaServer Faces (JSF) technology. JSF is not a standalone product which is solely used to develop comprehensive views for web applications but rather has to be regarded along with the Servlet and JavaServer Pages technology which form the basis of JSF. In the Java World, Servlets are the interface between the web server and Java applications. Servlets are very performant due to the good integration into the server and the fact that Servlets are compiled once and can then be executed directly in the JVM and optimized by just-in-time compilation instead of being interpreted on every request. Servlets are portable and thus can be deployed in any web server, containing a Servlet container. As there are many web servers available for most of the operating systems which support the Servlet technology, Servlets can be applied nearly everywhere. Another advantage of Servlet development is the object oriented approach of the Java programming language which also allows the usage of many available and helpful libraries which can ease database access, XML parsing, and much more. On the other hand, it is very difficult to separate logic and view as the HTML is commonly embedded into the Java class(es) of the Servlet. This often results in very unclear code and makes the maintenance of the Servlet code very time consuming. Thus, it is not recommended to develop large web applications with complex web pages solely with the Servlet technology. It makes more sense to use Servlet technology when the web pages are not complex and no or minimal user-interaction together with high performance is required, or when limited server resources are available. Developing a web application which solely uses the Servlet technology will also guarantee the most possible compatibility with different web or application servers in the Java world. Sensible Servlet applications include simple pull services which mainly display the content of a resource (e.g. a database) or when the web interface is the preferred way to access Java code which performs custom actions and only short status messages have to be created in HTML. An example for this purpose is the HTTP interface of the open source WAP and SMS gateway Kannel[1]: Kannel comes with a library which can be accessed through HTTP requests. The result of the executed operations is returned as HTML message.

[1] Project page of WAP and SMS gateway Kannel: http://www.kannel.org/

As mentioned, Servlet's greatest disadvantage is the tight binding of logic and presentation. Therefore, Sun has introduced the JSP technology which should fix this problem by encapsulating business logic into JavaBeans (POJOs) which can easily be accessed from the view. Additionally, JSP, which comes with XML syntax since version 2.0, allows for rolling out of reoccurring view components or elements into custom tags which can be included on several pages. These changes make view development and maintenance much more comfortable. As JSPs are compiled into Servlet code, they inherit most of the advantages of Servlets (high performance, platform independence, etc.). As JSP is still a lightweight technology, it lacks some sensible features useful for larger web applications: A templating mechanism, an additional layer configuring the application's navigation, view components, etc. Nonetheless, for a long time, JSP has been the technology of choice for Java developers to develop common web applications.

JSF constitutes a good alternative to raw JSP. JSF is not a replacement of JSP or Servlets but rather an extension to JSP. With a fast growing set of available view components, its navigation rules and event model which are similar to Java Desktop applications with event and action listeners, JSF eases the development and maintenance of true web applications. The downside of these easements is the complexity of the framework. Hence, the response time of a JSF page is generally noticeably longer than that of the same page written with JSP. Thus, JSF will not be the technology of choice for web applications with a limited fraction of dynamic content or where the dynamic content is confined to displaying and decorating database values. In this case, JSF may be overkill and JSP, which is designated to dynamic output, would probably be the better technology. The best fit for JSF is a true web application: a web site with a lot of user interaction—rather than a web site with some dynamic content. Only then, JSF will bring structure and maintainability to the application user interface.

Literature

Bayern, S. (2002). *JSTL in Action - a gentle introduction to the JSP standard tag library.* London.

Bergsten, H. (09. 06 2004). *O'Reilly ONJava.com*. Abgerufen am 04. 09 2008 von Improving JSF by Dumping JSP: http://www.onjava.com/pub/a/onjava/2004/06/09/jsf.html

Burns, E., & Kitain, R. (05 2006). *JavaServer™ Faces Specification.* Abgerufen am 22. 08 2008 von Java Community Process: http://jcp.org/aboutJava/communityprocess/final/jsr252/index.html

Coffee, P. (1997). *How to program JavaBeans.* Emeryville.

Falkner, J., & Jones, K. (2003). *Servlets and JavaServer Pages - The J2EE Technology Web Tier.* Boston.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Boston.

Hass, B. H. (2008). *Web 2.0 - neue Perspektiven für Marketing und Medien.* Berlin.

Holmes, J. (2008). *Struts - the complete reference.* New York.

Jendrock, Ball, Carson, Evans, Fordin, & Haase. (2007, 09). *The Java EE 5 Tutorial - For Sun Java System Application Server 9.1.* Retrieved 06 04, 2008, from http://java.sun.com/javaee/5/docs/tutorial/doc/

Maki, C. (2007). *JPA 101 Java Persistence Explained.* London.

Mann, K. D. (2005). *JavaServer Faces in Action.* Greenwich.

Marinschek, M., Radinger, W., & Spiegl, T. (2007). *Google Web Toolkit - Pfeilschnelle Ajax-Anwendungen in Java.* Heidelberg.

Müller, B. (2006). *JavaServer Faces - Ein Arbeitsbuch für die Praxis.* München.

O'Conner, J. (2001). *Internationalization using the Java platform.* Harlow.

O'Reilly, T. (2006). *Web 2.0 Compact Definition - Trying Again.* Indinanapolis.

Srinivasan, K. (07. 03 2006). *java.net The Source for Java Technology Collaboration*. Abgerufen am 05. 06 2008 von java.net: Unified Expression Language for JSP and JSF: http://today.java.net/pub/a/today/2006/03/07/unified-jsp-jsf-expression-language.html

Stark, T. (2006). *Jetzt lerne ich J2EE - der einfache Einstieg in die Programmierung mit der Java 2 Enterprise Edition.* München.

Stark, T., & Samaschke, K. (2005). *Das J2EE-Codebook.* München.

Tate, B., & Hibbs, C. (2006). *Ruby on Rails - up and running.* Beijing.

White, E., & Eisenhamer, J. (2007). *PHP 4 in practice.* Boston.

Wolff, E. (2007). *Spring 2 - Framework für die Java-Entwicklung.* Heidelberg.