A General eCommerce Platform with Strong International and Local Aspects

By Martin Ramsin

A Master's Thesis

August 2000

Examiner: Professor Seif Haridi
Supervisors:Andy Neil and Mark Bünger, Icon MediaLab
Vladimir Vlasov, IT/KTH

Department of Teleinformatics

Abstract

The objective of the Master's project was to build a platform for e-commerce solutions. The goal with this platform was to facilitate process of building e-commerce solutions. An e-commerce solution is often a Web store that sells products or services.

Today, when building a Web-store, it is a custom to use a database such as Oracle to store persistent data and a Web server to handle client requests. Most of the components of the Web store are often written in a server-side computing language such as ASP or Java. The requests to the database is translated into SQL-code (SQL is a standard language for databases). When constructing a Web store one has to know how to set up a database-structure and program the SQL calls. One also has to know how a web server works and how to handle transactions, security, stability and much more for both the Web components and the database components.
Recently how ever, the concept of application servers emerged. An application server is a server with built in services for handling these kinds of difficulties when constructing a Web store. Application servers are very powerful and contain the latest technology in server-side application development.

I chouse to build the e-commerce platform upon the J2EE application server developed by SUN Microsystems.

The platform was to be general in that aspect that any kind of e-commerce solution could be built using the e-commerce platform as a base. Another feature of the platform was a built-in support for localization and internationalization.

Preface

This Master's Thesis is a result of my Degree Project performed at Icon Medialab in San Francisco, USA between October 1999 and March 2000. It will conclude my Master of Science in Electronic Engineering at the Royal Institute of Stockholm (KTH).

Icon Medialab is a Swedish Internet solution company with offices in most European countries, Kuala Lumpur, New York and San Francisco.
The San Francisco office is located in south of market (Soma). Soma is close to the center of San Francisco and has been getting popular for new media and Internet companies the last couple of years. There is today about 50 employees in the San Francisco office.

Thanks to Andy Neil and Mark Bünger at Icon Medialab, San Francisco and to Vladimir Vlassov, my advisor at KTH.

Table of contents

1. INTRODUCTION 5

1.1 Background to the project 5

2. CHOICE OF DESIGN STRATEGY 7

2.1 Choosing a architecture for the e-commerce platform 7
2.2 Component models 8
2.3 Choosing a design strategy 10

3. HOW TO BUILD A J2EE APPLICATION 11

3.1 Application server 11
3.2 The Java 2 Enterprise Edition application server 12
3.3 Enterprise Java Beans (EJB) 13
3.4 Java Servlets 15

4. SOME IMPORTANT CONCEPTS IN THE J2EE PLATFORM 16

4.1 JNDI lookup service 16
4.2 RMI 16
4.3 Deployment of the J2EE application 17
4.4 Packaging Components into EJB Modules 19
4.5 Security 20
4.6 Internationalization and Localization 20

5. BUILDING THE E-COMMERCE PLATFORM 22

5.1 In general 22
5.2 Evaluation 22
5.3 The three-tier model 22
5.4 Interaction between components of the platform 23
5.6 The Database structure 31

6. DEPLOYMENT 33

6.1 Deployment of the Web- and EJB components 33
6.2 Interaction between the WEB and EJB modules 34
6.3 Internationalization and Localization 35
6.4 Security 36

7. CONCLUSIONS 37

8. RECOMMENDATIONS 38

9. REFERENCES 39

APPENDIX A - SERVICES PROVIDED BY THE EJB CONTAINER 40

APPENDIX B - SERVICES PROVIDED BY THE J2EE PLATFORM 41

APPENDIX C - TECHNICAL REQUIRES FOR THE J2EE PLATFORM 42

APPENDIX D - DEPLOYMENT IN DETAIL 43

APPENDIX E - DEPLOYMENT DESCRIPTORS IN DETAIL 44

APPENDIX F – GLOSSARY 45

APPENDIX G – CLASS DOCUMENTATION 47

1. Introduction

1.1 Background to the project

The market of e-commerce is growing fast and its potential is gigantic. A store on the Internet has an enormous advantage over traditional stores; the number of potential clients is the number of people that has access to the Internet. Another big advantage of selling products on the Internet is that the producer of a product can sell directly to the client without any people making profit on the product on the way from the producer to the client. Because of this, the price of the product can be lower. Because of these advantages and other it is very likely that e-commerce will revolutionize traditional commerce and the way that people shop.
When building a store on the Internet, the basic structure of the store is often similar. I wanted to construct a platform that any kind of e-commerce store could use with this basic structure already implemented.

The project was to build a general e-commerce platform with local aspects. There are a lot of difficult words in this definition and I will in this section explain what they mean.
E-commerce means doing commerce using electronic means. More specific, e-commerce is selling products or services in virtual stores on the World Wide Web.
A platform for e-commerce is a software that can be used to build the basic structure of an e-commerce store upon. Typically, the platform would handle services such as client register, product- database handling, money transactions, security etc.

The platform that I was building was to be *general* in that aspect that it could be used for this basic services by any kind of e-commerce solution no mater what the store would sell.
Local aspects would be to implement to this platform a service for international and local views.

Server-side requirements are the equivalent to the requirements of the J2EE application server. The J2EE application server requires a platform that is Solaris Operating Environment, version 2.6 or Windows NT, version 4.0. Java JDK version 1.2.2 or later must be available on the platform. The J2EE application also requires at least 128 MB of memory.
Supported databases and JDBC drivers are Oracle8 Server version 8.05, Microsoft SQL Server versions 6.5, 7.0 and Cloudscape version 3.0.

Client-side requirements are a GUI (Graphical User Interface) that is informative, convenient and responsive and that the time to download the client should be short.

In this project, I expected to build a platform that would be a useful tool when developing e-commerce solutions. The platform should be general so that any type of e-commerce solution could be built upon the e-commerce platform.
A prototype has been implemented for this project.

The report is structured as follows:

Chapter 1 - Introduction - specifies what the project is about. This chapter also contain information about what platform I chouse to work on, hardware requirements for running the e-commerce platform and the results that I expected of the final product.

Chapter 2 - Choice of design strategy - presents several different types of design strategies that I chouse from evaluate them and explain the choice of strategy.

Chapter 3 - How to build a J2EE application - explains what is an application server and more closely how the J2EE application server from SUN work. It also presents two Java APIs that are important in the construction of the e-commerce platform.

Chapter 4 - Some important concepts in the J2EE platform - continues the presentation of the J2EE application server by presenting some of its most important concepts. These are:

- JNDI lookup service
- RMI
- Deployment of an J2EE application
- Packaging an J2EE application
- Security in the J2EE platform
- Internationalization and localization an J2EE application

Chapter 5 - Building the e-commerce platform - is the chapter where I explain how I built the e-commerce platform.

Chapter 6 - Deployment - in this section, I explain how the deployment of the e-commerce platform was done.

Chapter 7 - Conclusions

Chapter 8 - Recommendations

Chapter 9 - References

Appendix A is about the services that the EJB container in the J2EE server provides.
Appendix B is about the services that the J2EE server provides.
Appendix C is about the technical requires in the J2EE server.
Appendix D is about the deployment in the J2EE server provides.
Appendix E is a description of deployment descriptors in detail
Appendix F is a glossary for this report
Appendix G is a summary over the most important classes in the prototype of the e-commerce platform

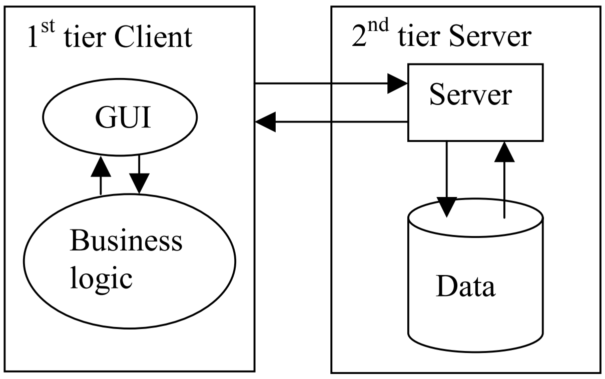
2. Choice of design strategy

In this chapter, I will explain what kind of design strategy I have chosen for the e-commerce platform and why.

2.1 Choosing a architecture for the e-commerce platform

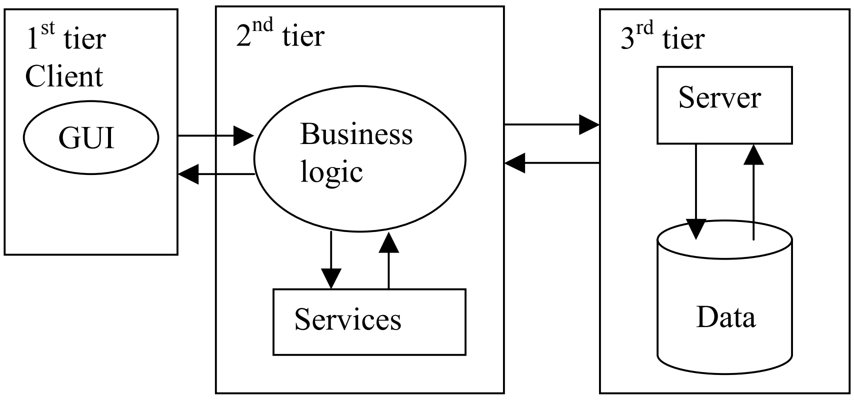
A model that is commonly used is the *two-tier* model (see figure 2.1). This is the famous client-server model with the client as the first tier and the server and database as the second.

Figure 2.1.The two-tier model



Another model is the *three-tier model* (see figure 2.2). In this model, the client represents the first tier. The middle tier is the computing tier and is often called application server or middleware. The third tier represents databases and other back-end products.

Figure 2.2.The three-tier model



In this approach, the client can be thin and independent because the middle tier does almost all the computing. In the two-tier model, the client is not as thin and does a lot of computing itself.
Because the client is thin in the three-tier model, it makes the client independent of the server-side computing and therefore one can write applications that are scalable.
Another benefit with the three-tier model is that the second tier can handle security and personalization of a client. Security and personalization are extremely important in any type of e-commerce. Security rules in the second tier can protect the database in the third tier by

identifying the client and decide through authentication if the client has access to the data. Personalization is an important concept in any type of e-commerce. The three-tier model makes it possible to implement personalization computing in the second tier for localization, language translation, personalization of the GUI etc.
Because most of the computing in this model is in the second tier and not in the client tier as in the two-tier model, the performance is increased. The application server that does the computing in the second tier is typically very fast in comparison to the machine on the client.

I decided to work with the three-tier model for the e-commerce platform for these advantages over the two-tier model. This decision meant working on an application server.

An application server is a server with built-in services for handling database transactions, security, scalability etc. Application servers are being used more and more when developing e-commerce solutions.

One of the most popular type of application server today, and the one that I decided to use, is built upon the concept of **containers**. A container is the part of the application server that hosts **components**. The components are small entities that together form the application(s) running on the server. The container provides a *runtime environment* for the components. This runtime environment performs services for the component such as transactions, security, database connection etc.
Because of this approach of having containers that perform these basic services, the components can be very thin and easy to program. Of course, the components have to be programmed after a certain model so that the container knows how to handle them.

The container-component model suited me well because one of the benefits with this approach is that the applications can be scalable; it is possible to add or remove components to the application without changing the behavior of the other components. This is important when building a platform that will be used to build applications on. I can now build components that perform a certain tasks and together form a platform for e-commerce applications to interact with and the platform-components will not be changed.

2.2 Component models

On the market today, there are mainly three types of component models that spring out of the concept on application servers using containers:

- DCOM
- Enterprise Java Beans
- Corba

COM (Component Object Model) was developed by Microsoft and was the first component standard to be used on a big scale. COM was the core component in many Microsoft products including Office 97, Windows 95 and Windows NT.

DCOM (Distributed Component Object Model) is an extension of COM that allows components to communicate with each other over a network.

A DCOM component can be programmed in almost any programming language and so the DCOM standard is language-independent.
ActiveX is a popular application service that is built on top of DCOM. ActiveX allow components to be embedded in Web sites and can be compared to Java applets.

DCOM is best supported on Windows 95 and NT platforms. However, Microsoft has released versions of DCOM for other platforms for example MacOS and UNIX.
Components written to the DCOM specifications are not platform independent and must be recompiled for a specific platform.

CORBA (Common Object Request Broker Architecture) is a component standard from the Object Management Group (OMG). The latest version (3.0) is a server side CORBA component model. Like DCOM, CORBA is a language-independent distributed computing architecture.

Enterprise JavaBeans (EJB) is an extension of the JavaBeans specification to make Java components suitable for server-based applications.
The architecture of a component model is important as it specifies how the components communicate and collaborate.

The Microsoft DCOM model differs a little bit in behavior from the other two. This is a direct consequence of the difference for what the three models are used; DCOM was developed to build distributed application on a Microsoft platform, CORBA was written to set a universal standard for components and EJBs is SUN Microsystems try to commercialize the CORBA standard.

TABLE 2.1	**DCOM**	**EJB**	**CORBA**
Component types	Session	Session, entity	Session, entity, process, service
Container services	Transaction, security	Transaction, security, persistence	Transaction, security, persistence
Other services	Events notification, load balancing, data caching	Directory, thread pulling, message queuing	Event notification
Cross-platform?	No	Yes	Yes

In table 2.1, some of the main differences between the three component models are. At this time it is not so important to understand all the different types of components and services listed in the table but to see that the CORBA model offers a most complete set of services followed by the EJB model. It is also important that the CORBA and the EJB standards are cross-platform but DCOM is not.

EJB model has the advantage of having an application server implemented, and available for free*– the Java 2 Enterprise Edition (J2EE) platform.

2.3 Choosing a design strategy

As the application I was constructing had the aspect of being general, the model that I picked had to be cross-platform. Therefore the DCOM model could not be used. That left me to choose between the EJB model and the CORBA model.

I decided to work with EJB because I had heard a lot about Java Beans and thought it would be a good thing to learn about the concept of Java Beans. Another advantage is that SUN INC. offers an application server called the J2EE server for free*. This application server is built on the concept of components (EJBs) and containers.
Of course, I could have picked the CORBA model and written CORBA components in Java for example and then tried to find a vendor that was equipped with an application server that run CORBA objects.

The Java programming language was a necessary choice as the EJBs are Java language-specific. But it turned out that the Java language suited me very well because of the many useful APIs for example Servlets, Internationalization and localization etc.

* The Java 2 Enterprise Edition platform is available for free from Java Soft but can only be used for non-commercial development.

3. How to build a J2EE application

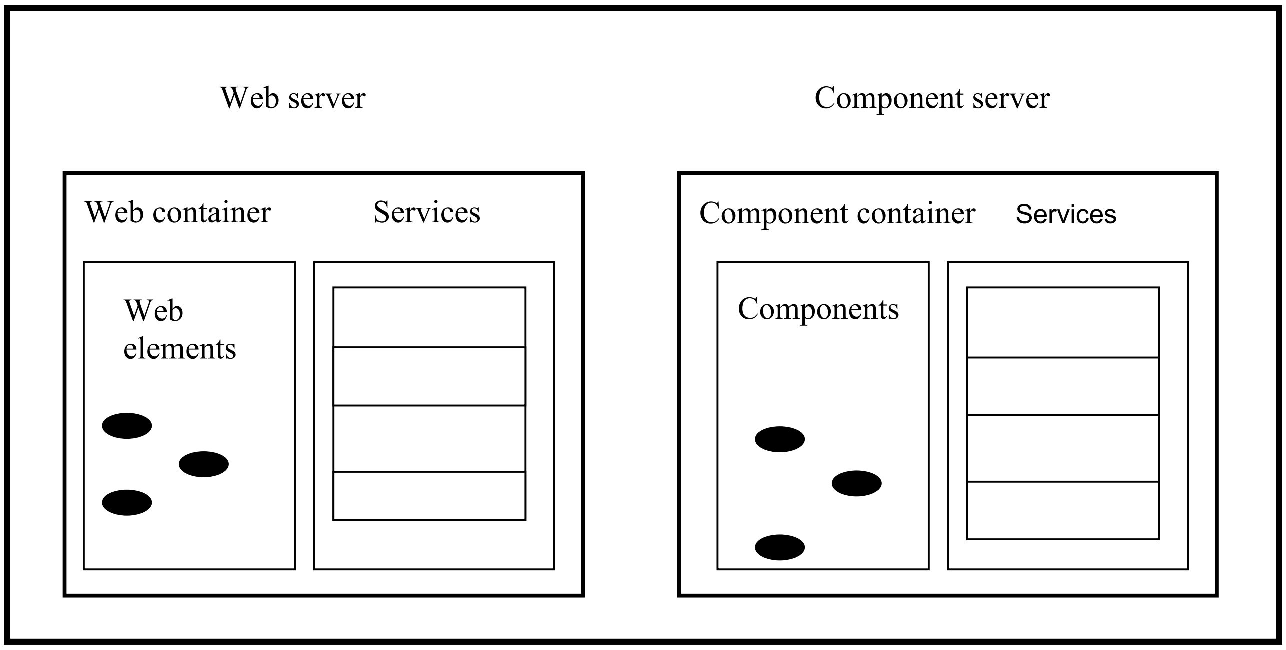
So how do I build a general e-commerce platform with local aspects based on the concept of Enterprise JavaBeans living in an application server? In order to explain that, I first have to explain in more detail what an application server is, how the J2EE application server for Enterprise JavaBeans works and what exactly is an Enterprise JavaBean.

3.1 Application server

A product that can host the types of components described in chapter 2.2 (DCOM, EJB and CORBA) and that provides a container for runtime services are generally called *application server*.

In the tree-tier model, the middle tier can be thought of as the computing tier. Because the middle-tier application server provides a lot of services to the components running on it, the components can be thin, simple, and rapidly developed. It is also easy to integrate new components with existing applications and databases.
An application server is generally equipped with a web server and a component server. The web server hosts web side applications such as HTML pages, Java Servlets, Java Applets, etc. The component server host's components such as the ones described in chapter 2.2. Both the web server and the component server are hosting its components in containers. The container handles the runtime services for the component. The servers are also offering some higher-level services.
Figure 3.1Illustrates the architecture of an application server.

Figure 3.1 Application server



3.2 The Java 2 Enterprise Edition application server

The Java 2 platform, Enterprise Edition (J2EE) developed by SUN INC is an application server that hosts enterprise beans. There are other application servers on the market but this one is free under the agreement that one is not developing commercial products on it. I am therefore using a J2EE to build the e-commerce platform on.
The J2EE application server includes a Web server and an EJB server. These two servers are distinct software entities that may or may not be located on the same machine. If they are located on different platforms, connectivity between the platforms is established using RMI-IIOP.
The J2EE platform can host the following types of components:

- Applets
- Application clients
- Enterprise JavaBeans
- Web components (JSP, Servlets, HTML etc).

EJB Container services

The J2EE platform offers the Enterprise Java Beans a lot of support like Web server, transaction management etc. Applications built on the J2EE platform are scalable and easy maintainable. This is important for the general aspect of the e-commerce platform. If a future e-commerce product would be build based on the e-commerce platform, it would be possible as J2EE application always are scalable; they let you add components or functionality's without altering the existing components.

Enterprise beans instances run within an EJB container. They provided the following runtime services to enterprise beans (see appendix A for detailed information):

- Transaction Management
- Remote Client Connectivity
- Security
- Life Cycle Management
- Database Connection Pooling

Other services provided by the J2EE platform

The J2EE platform offers these additional services (for a closer look, see chapter 3.3 for Enterprise Java Beans, chapter 3.4 on Java Servlets, chapter 4.1 for JNDI, chapter 4.2 for RMI and appendix B for detailed information about the rest of the services):

- Java DataBase Connection, JDBC 2.0 Extension.
- Java Transaction API, JTA 1.0
- Java Naming and Directory Interface, JNDI 1.2
- Servlet 2.2
- Java Server Pages, JSP 1.1
- Enterprice Java Beans, EJB 1.1
- Java Remote Method Invocation over IIOP, RMI-IIOP 1.0
- Java Message Service, JMS 1.0
- Java Mail 1.1
- Java bean Activation Framework, JAF 1.0

3.3 Enterprise Java Beans (EJB)
The Enterprise JavaBean is an extension of the JavaBean model. A JavaBean is a portable, platform-independent software component that enables developers to write components once and run them anywhere - benefiting from the platform-independent power of Java. JavaBeans can be manipulated in a visual builder tool and composed together into applications.

Enterprise JavaBeans are server components for building distributed applications. The best way to picture an enterprise JavaBean is to see it as a small unit that performs one specific task and that, once created, can be used without being changed in many different contexts.

There are two types of enterprise beans: session beans and entity beans

Session Beans
A session bean represents a client in the application server and is non-persistent. Non-persistent means that the state of the bean is not saved when the bean terminates. A session bean can be stateful or stateless.
In a *stateful* session bean, the bean's instance variables may contain a state. A statful session bean can have only one client and can, for example, be used to implement a shopping cart. When the client terminates, its corresponding session bean also terminates.
A *stateless* session bean does not contain any instance variables and can have multiple clients.

Entity Beans
An entity bean represents a business object in a persistent storage mechanism such as a database. For example, an entity bean could represent a customer or a product, which might be stored as a row in the customer or product table of a database.

An entity bean can have bean- or container-managed persistence. In a bean-managed persistence bean, the programmer writes the SQL code that is used to manipulate data in the database in the class file of the bean. The bean also has to manage the connection to the database via JDBC.
In a container-managed persistence bean, the EJB container (in the J2EE platform) handles the calls to the database.
For a container-managed persistence bean the developer do not need to write the SQL-code. Instead, the deployment tool (see chapter 6.1) generates SQL code.

The container-managed persistence beans are mush easier to write because the developer does not need to worry about SQL code and JDBC connections. In some cases however, when the calls to the database are complex, the container may not be able to create the SQL code and bean-managed persistence must be used.

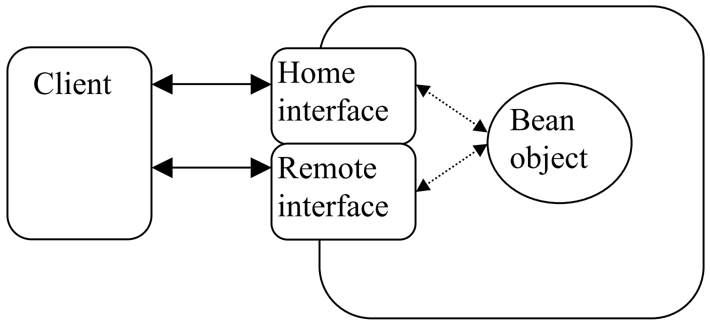
Multiple clients may share entity beans. Because the clients might want to change the same data, it is important that entity beans work within transactions. An entity bean can be of bean- or container-managed transaction nature.
In a bean-managed transactions bean, the developer implements the transactions and in a container-managed transactions bean, the EJB container manages the transactions.

Each entity bean has a unique object identifier that is called the primary key.

Distributed application
The enterprise bean is built to be remotable using the Java RMI API. This means that both local and remote programs can access an enterprise bean. A caller of an enterprise bean method can be another enterprise bean deployed in the same or different container. It can also be an application, applet, or Servlet on the same or a different platform. As RMI objects can interact with CORBA components, the caller of an enterprise bean method can be a component written in a programming other than Java.
The fact that enterprise beans are remotable makes the J2EE application a *distributed application*. A distributed application is made up of distinct components running in separate runtime environments, usually on different platforms connected via a network.

Interfaces
The enterprise bean's *home interface* defines the methods for the client to create, remove, and find EJB objects of the same type. A client can locate an enterprise Bean home interface through the standard Java Naming and Directory Interface (JNDI) API.
The instance of an enterprise bean is accessible via the enterprise bean's *remote interface*. The remote interface defines the methods callable by the client to manipulate the EJB object.
EJB object interface defines the operations that allow the client to access the EJB object's identity and create a persistent handle for the EJB object. Each EJB object lives in a home, and has a unique identity within its home.
For session beans, the container is responsible for generating a new unique identifier for each session object. The identifier is not exposed to the client. However, a client may test if two object references refer to the same session object. Figure 3.2 illustrates the interaction between a client and an Enterprise Java Bean inside an EJB container.

Figure 3.2 Client interaction with the Enterprise JavaBeans Container



Database Access
The Enterprise JavaBean specification does not require a particular type of database. An entity bean's information does not have to be stored in a relational database. It could be stored in an object database, a file, or some other storage mechanism.
Both session and entity beans can access a database.

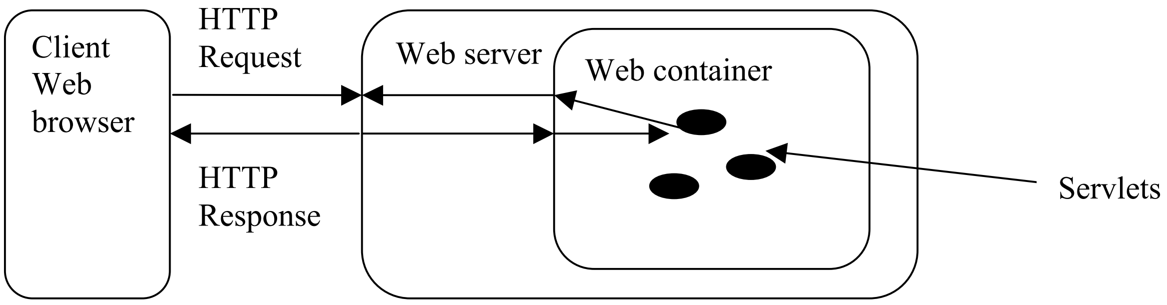
3.4 Java Servlets

A Servlet is a web component, managed by a container that generates dynamic content. Servlets are small, platform-independent Java classes. Servlets interact with web clients via a request-response model implemented by the Web container. This request-response model is based on the Hypertext Transfer Protocol (HTTP).

The Web container manages the response and the request objects and takes care of the Servlets lifecycles. Here follows an example of how a client could use a Servlet:

A client web browser accesses a web server and makes an HTTP request. The request is processed by the web server and is handed off to the web container. The web container, which can run on the same host or on a different host from the web server, determines which Servlet to invoke.
The Servlet uses the request object to find out whom the user is, what HTML form parameters may have been sent as part of this request, and other relevant data. The Servlet then performs the logic it was programmed with and generates data to send back to the client via the response object. This scenario is visually explained in figure 3.3.

Figure 3.3



4. Some important concepts in the J2EE platform
In this section, I will give a summary over some of the most important concepts in the J2EE platform.

4.1 JNDI lookup service
The JNDI (Java Naming and Directory Interface) API is a service that connects a simple name to a Java object.
The functionality of the JNDI API can be thought of, as, for example, a telephone directory service where one can look up the address of a person if one knows the name of the person. Another example of how a Naming and directory service is the DNS (Internet Domain Name System) that maps a name such as www.kth.se to an IP address such as 198.32.432.12.

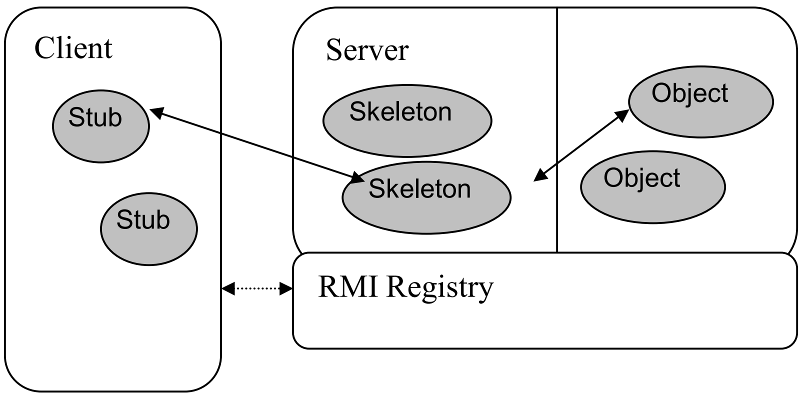
4.2 RMI
The RMI (Remote Method Invocation) is a Java API for writing distributed Java objects. RMI is based on one important principle: *The definition of a Java class* and the *implementation of the class are separated and can be run on different Java Virtual Machines*.

In RMI, the definition of a class is implemented as an interface to the class. The interface is a listing of what methods, in the RMI object, that are accessible to the client.
There are two kinds of classes that implement this interface: the stubs and the skeletons. A *stub* is the client side interface and the *skeleton*, the server side interface.
The actual implementation of the Java class using RMI is placed on the server.

When a developer has created the implementation and the interface of the class and compiled the code, the developer has to execute the *rmic* - a Java command that generate the stubs and skeletons. Once the stubs and skeletons are created, the developer needs to start the RMI registry with the *rmiregistry* command. Both rmic and rmiregistry are comes with the Java Development Kit 1.2.
The RMI Registry is a registry service that uses JNDI. When the server is started it will bind the objects using RMI in the RMI registry.

When the client calls a method in the object using RMI on the server, it first looks up the object using JNDI lookup service, in the RMI Registry. It then sends the request to the skeleton, which forwards it to the object. The skeletons receive a value back from the objects method and send this value back to the stub.
Figure 4.1 shows the general idea of the RMI architecture.

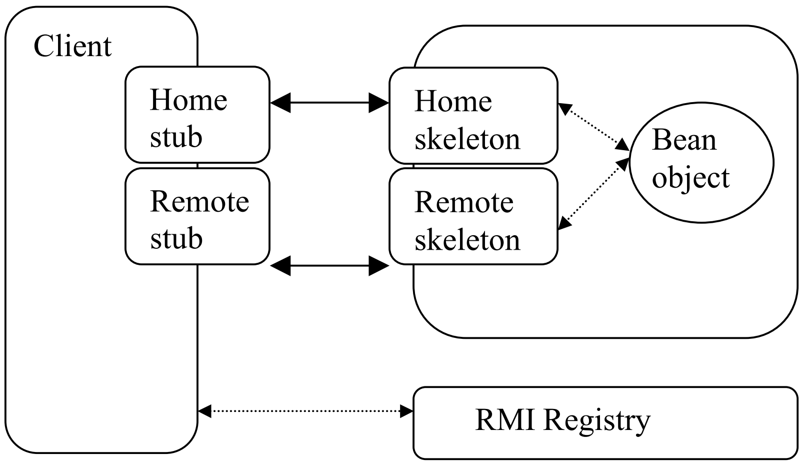
Figure 4.1



An example of a distributed program that could be using RMI is a chat server. A client in this case would be a user. The user needs to download a stub before she can enter the chat area. The interface that is the content of the stubs (one for every user) and the skeleton would be a list of methods to read and write to the chat area. The object would implement these methods and some additional methods for the graphical interface.

The RMI architecture as used in enterprise beans is shown in figure 4.2 Note that the client in this case is a Java application, Servlet, JSP file, applet or another enterprise Java bean. When working with enterprise beans on the J2EE server, the stubs and skeletons will be created automatically in the deployment phase.

Figure 4.2 Communication of EJBs using RMI



On the J2EE platform, almost all communication between components is using RMI. The home- and remote-interfaces of the enterprise beans are being used as stubs and skeletons.

4.3 Deployment of the J2EE application

When a component for the J2EE application is implemented and compiled it has to be *deployed* in the deployment tool that comes with the J2EE platform. The deployment process transforms the component to a format that fits the J2EE server. It also creates the stubs and skeletons and binds the RMI objects to the JNDI registry service (see Appendix D for details about what happens when an application is deployed).

The main idea of the deployment phase is to make it possible to use the same enterprise beans and web components in more than one context without changing the code of the components. In the deployment tool, one can give the components environment entries and link hard coded names to JNDI lookup names. Here are two examples to demonstrate this.

Example 1: Environment entries.
In the first example, the developer wants to write a Web store that contains a Servlet to handle a catalogue. If it is spring, the catalogue Servlet should show the spring catalogue and if it is winter it should show the winter catalogue. With environment entries, the developer doesn't need too re-write the catalogue Servlet code when winter changes into spring. The developer

just has to re-deploy the application and set a new environment entry such as: Season = “spring” instead of: Season = “winter”. The environment entries are bound in the JNDI registry.

Example 2: Linking hard coded names to JNDI lookup names
In the second example, the developer has an enterprise bean that looks up a database to get data. On the J2EE platform, the developer uses JNDI when looking up databases. In the code of this enterprise bean, there will be a JNDI name that is used to locate the database. Let's say now that all data is moved to a new database with a different name. Now, the developer don’t need to rewrite the enterprise bean’s code and change the old JNDI name to the new name of the database. The developer just has to re-deploy and link the old JNDI name to the JNDI name of the new database.

Deployment Descriptors

The role of the deployment descriptor is to capture information that is set while deploying an application for example environment entries and links from hard coded names to JNDI lookup names that I demonstrated in the previous section. Deployment descriptors specify two kinds of information:

- Structural information: The structural information describes the different components of the JAR or WAR file and their relationship with each other.
- Assembly information: The assembly information describes how the contents of the JAR or WAR file can be composed into a larger application deployment unit.

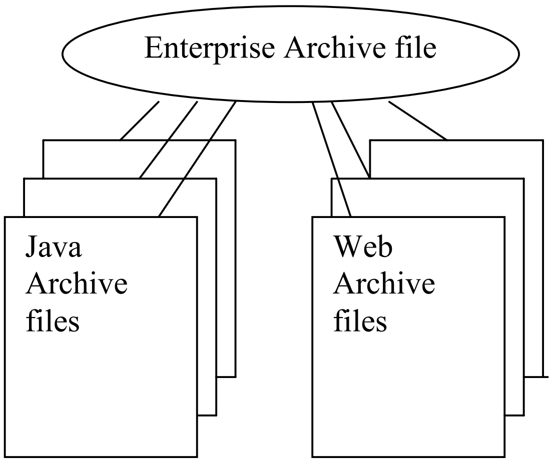
The deployment descriptor is an XML-file and can look like this for a JAR file containing a session bean called TheShoppingClientControler:

```
<session>
<description>The MVC controller</description>
<display-name>TheShoppingClientController</display-name>
<ejb-name>TheShoppingClientController</ejb-name>
<home>com.sun.estore.control.ejb.ShoppingClientControllerHome</home>
<remote>com.sun.estore.control.ejb.ShoppingClientController</remote>
</session>
```

Structure of the J2EE application

When deploying the application, the developer will compose Java Archive files (JAR files) and Web Archive files (WAR files). When the application is ready, the deployment tool will create an Enterprise Archive file (EAR file). The structure of a J2EE application is shown in figure 4.3

Figure 4.3



Enterprise Archive file
The goal of this file is to set the structure of the application and to eliminate portability problems.

Java Archive files (JAR files)
A software unit, that consists of one or more enterprise beans and an EJB deployment descriptor, is called an EJB module. An EJB module is packaged and deployed as an EJB Java Archive (.jar) file. It contains:

- Java class files for the enterprise beans, and their remote and home interfaces.
- Java class files for super classes and super interfaces
- An XML deployment descriptor that provides both the structural and application assembly informations about the enterprise beans in the EJB module.

An EJB JAR file differs from a standard JAR file in one key aspect: it is augmented with a deployment descriptor that contains meta-information about one or more enterprise beans.

Web Archive files (WAR files)
A software unit that consists of one or more web components such as Servlets, JSP or html files and a WEB deployment descriptor is called a web module. A web module is packaged and deployed as a Web Archive (.war) file. It contains:

- Java class files for the Servlets and the classes that they depend on.
- JSP pages and their helper Java classes
- Static documents (for example, HTML, images, sound files, and so on)
- Applets and their class files
- An XML deployment descriptor.

These modules are reusable. It is possible to build new applications from existing enterprise beans and components. And because the modules are portable, the application they comprise will run on any J2EE server that conforms to the specifications.

4.4 Packaging Components into EJB Modules

There are several ways of composing the EJB Modules. Here follows some packaging options:

1. Package each enterprise bean in its own EJB module. Each enterprise bean has its own deployment descriptor and is packaged in one EJB module.
=> Maximum reusability of each enterprise bean.
2. Package all enterprise beans in one EJB module. This is the simplest way to implement the modules.
3. Package related enterprise beans in one EJB module. The enterprise beans are grouped based on their functional nature and put in one EJB module manner.

4.5 Security

An e-commerce application is often designed to minimize the barriers that a user must overcome to become a customer. In contrast to typical computer user authentication environments, where a user must wait for an administrator to set up the user's account, an e-commerce application often let users create their own accounts.
The J2EE platform security services are designed to ensure that resources are accessed only by entities authorized to use them. Access control involves two steps:

1. Authentication
A client must establish its identity through *authentication*. It typically does so by providing her name and password. A client that can be authenticated is called a *principal*. A principal can be a user or another program. Typically, logging in authenticates users.

2. Authorization
When an authenticated principal tries to access a resource, the J2EE container determines whether the principal is authorized to do so based on the security policies in the application's security policy domain.
The J2EE platform authorization model is based on the concept of *security roles*. The security roles are set in the deployment phase and are executed at runtime by the container.
In the EJB container, the security roles can control whether a principal is authorized to use a specific method of an enterprise bean.
In the web container, security roles are used to protect URL patterns and an associated HTTP method for example "GET".
Before all the mapping of security roles to users can work, one has to create the different groups of users. Unfortunately, the J2EE platform only provides a command-line tool to manage the users.

4.6 Internationalization and Localization

Internationalization may often be overlooked when developing a web application. However, if one is developing a web application that will be used in more than one market, it is important to plan for internationalization and localization in an early phase of the development. It is easier to design an application that is capable of being internationalized than to rebuild an existing application, which can be both costly and time consuming. Planning for internationalization and localization at the beginning of a project can save a great deal of time and money.
An internationalized application can present a local view of the application for users from different countries and with different languages. With a local view I mean that not only the application is language specific, but also presents prices in the currency that the user is used to and

Internationalization
Internationalization is a process that aims to make an application possible for many different languages. If this process is well built up from the start, it should be small task to add a new language to the application even after the application is complete. An internationalized application has a well-formed structure for the different *locales*. A locale is a language- and country-specific data type. For example, a locale could be "fr_CA" which would refer to a

French speaking Canadian or it could be “fr_FR”, which would be a French speaking Frenchman.

Localization
Once and application is internationalized it can be localized.
Determining what content is given to the users; localization can be done in a few different ways. Web applications can be designed to deliver localized content based on a user preference or they can be designed to automatically deliver localized content based on information in the HTTP request or by user selection.
When an application allows users to select a language, the preferred language can be stored in the session. The selection can occur through a URL selection or a form post, which can be used to set an application level, preferred language. This data can be maintained as part of a user profile, which will be stored on the client’s system using a cookie or in a persistent data store.
Applications can also automatically deliver content by using Accept-Language attribute in header information of the HTTP request and mapping it to a supported locale. To use automatic application-level locale selection, it is prudent to also provide a mechanism to let the user override the automatic selection and select a preferred language.

5. Building the e-commerce platform

In this section, I will explain how I used the J2EE server to build the e-commerce platform.

5.1 In general

Building a platform for e-commerce solution is not an easy task. Because I chouse to work with the J2EE server however, the job became less difficult - I did not need to program the server myself. The J2EE server also handles security, transactions and database connections and many other useful services.
The e-commerce platform would set up a basic structure for an e-commerce solution on the J2EE platform using EJBs and Servlets. I implemented entity EJBs for the persistent data such as product and profile databases, session EJBs to keep track of the session of a user and Servlets to realize the basic features of a e-commerce solution such as login/logout or a catalog.

When the EJBs and Servlets were implemented, they needed to be deployed in order for the J2EE application server recognizes them as a part of a J2EE application.

5.2 Evaluation

The goal with the project was to construct a platform that could be used to build any type of e-commerce store upon. Basically, when constructing a store upon the e-commerce platform, one would only need to implement the graphical user interface and maybe add some enterprise beans or make some changes to the beans that I had built. The enterprise beans would be so general that they could work in any kind of e-commerce solution.
The difficulty when implementing general enterprise beans was that I had to find a complete set of variables for each enterprise bean.
For example, the Product EJB defined a product to be sold in a Web store and its parameters (Id, name, description etc.) had to meet the demands for parameters from any Web store selling any kind of product. Of course, a complete set of parameters can not be found and therefore the e-commerce platform can never be general for all types of e-commerce. Anyhow, I tried to make the e-commerce platform as general as possible and made a survey about e-commerce on Icon Medialab about what kind of parameters that are normally used in a Web store.

What I needed to implement was a set of enterprise beans and Servlets that together form the basic services of an e-commerce store. As it turned out, the most difficult and time-consuming was not to build the beans and Servlets but the process of integrating them into the J2EE server.

5.3 The three-tier model

I created two user interfaces: one Web store and one interface to manage the database. There are three different types of users to these interfaces: *Tour*, *Profile* and *Administrator*.

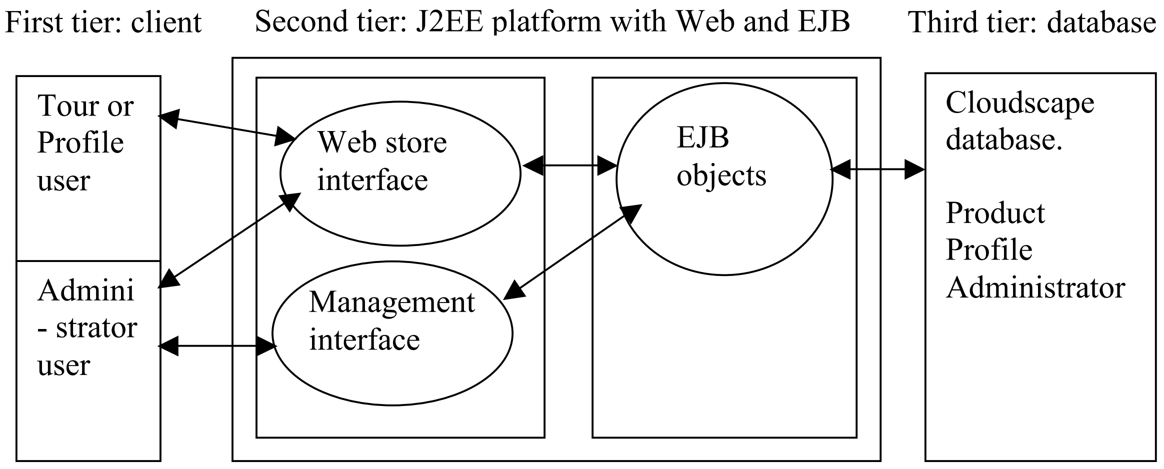
A Tour is a user that can access the Web store and all the functionalities of the Web store but can not access the Management interface. The Tour user can become a member by creating a Profile. There is no record stored in the database of the Tour users.

A Profile is a member in the Web-store and, like the Tour, can not access the Management interface. The Profile reaches the same functionalities as the Tour in the Web store but can profit of member prices and can see what product that he/she has recently purchased. The record of a Profile user is stored in the Profile database and can be accessed by the Profile EJB.

An Administrator has access to both the Web store and the Management tool. In the Web store, the Administrator has the same view as the Tour has but can not become a member. The record of an Administrator user is stored in the Administrator database and can be accessed by the Administrator EJB

The three-tier model of the e-commerce platform is shown in Figure 5.1.

Figure 5.1

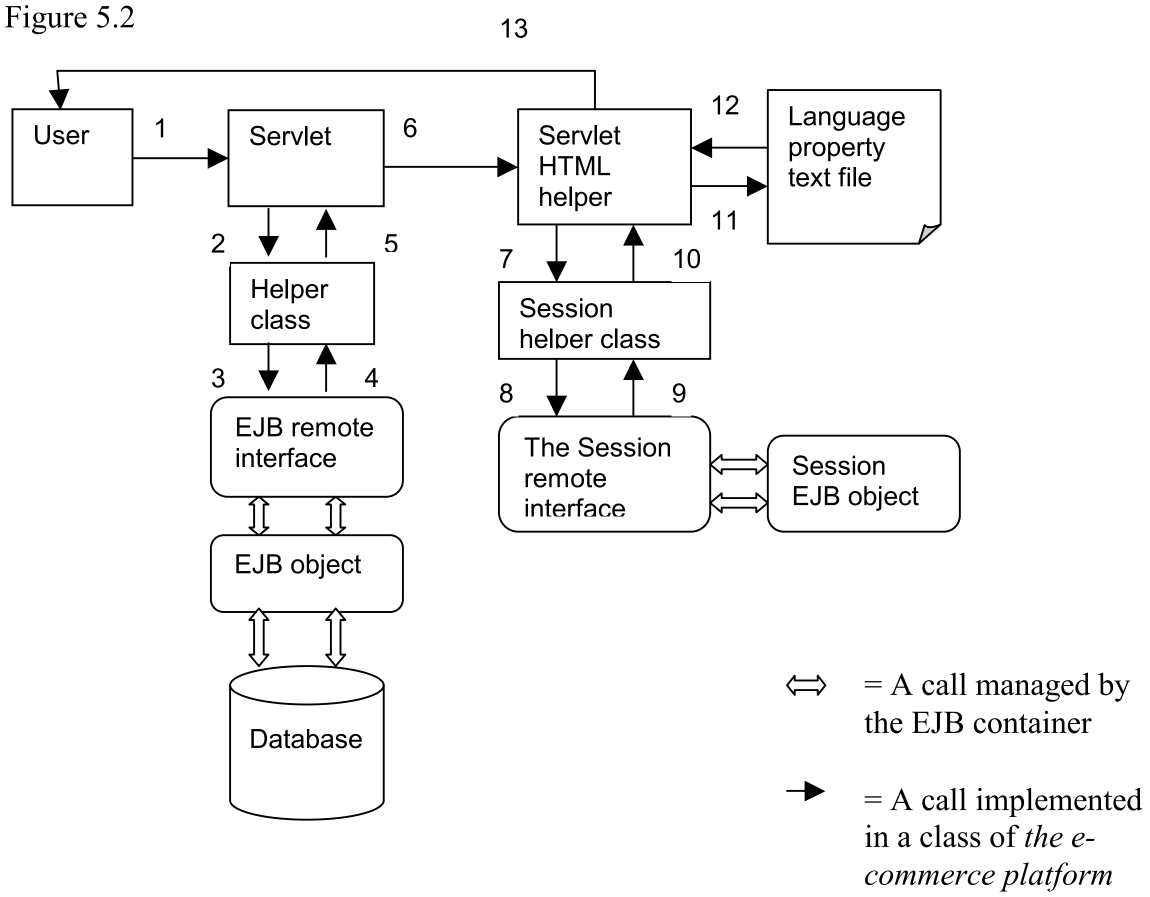


Connection between user and web interface is using the HTTP protocol. Between the Web interfaces and the EJB objects and when EJB objects are interacting with each other, the connection is RMI. The connection between EJB objects and the database is using JDBC.

5.4 Interaction between components of the platform

I chouse to use Servlets to present the graphical user interfaces, three entity enterprise beans to store data in the database and two session beans keep track of the user when the user has logged in.
Figure 5.2 demonstrates a scenario of a use-case where an entity EJB is used to fetch a value in a database.

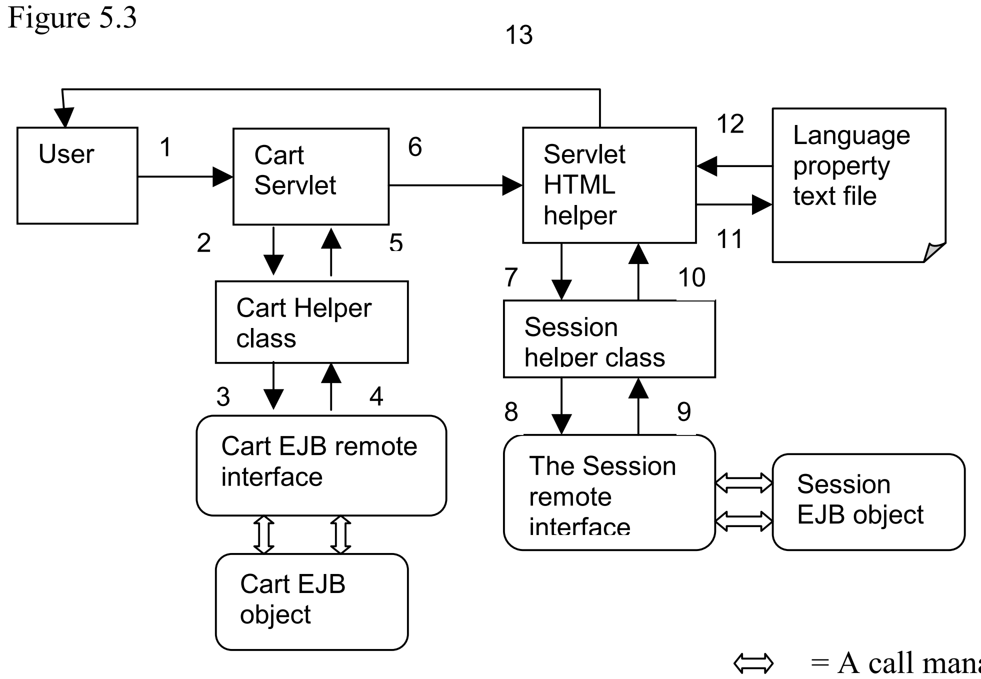
Figure 5.2


13
User
1
Servlet
6
Servlet HTML helper
12
Language property text file
11
2
5
Helper class
7
10
Session helper class
3
4
8
9
EJB remote interface
The Session remote interface
Session EJB object
EJB object
Database
⇔ = A call managed by the EJB container
→ = A call implemented in a class of *the e-commerce platform*

1. The Servlet gets an HTTP Request from the user.
2. The Servlet asks a helper class to compute the tasks.
3. The helper class connects to the EJB objects through the remote homes and accesses the EJB methods. These methods operate on the instance variables of the EJB object. The EJB container translates the calls on the instance variables to SQL-calls to a database.
4. The EJB object methods return the values that were fetched in the database, through the remote interface to the helper class.
5. The helper class returns the values to the Servlet.
6. The Servlet send the computed values to the Servlet HTML helper class.
7. The Servlet HTML helper class asks the Session helper class for the locale of the user. The locale is used for internationalization and localization tasks.
8. The Session helper class asks the Session session beans remote interface for the locale. The remote interface accesses the Session object and gets the value.
9. The Session object methods return the value that was stored in the Session session bean object, through the remote interface to the helper class.
10. The helper class returns the values to the Servlet HTML helper class.
11. The Servlet HTML helper class gets the locale value and calls a resource bundle to open a Language property text file, specific for the users locale, for translation of the final HTML response. The Servlet HTML helper class can also use the Locale for localization (see chapter 6.3).
12. The Language property text file sends the translation back to the Servlet HTML helper class.
13. The Servlet HTML helper class finally sends the HTML response to the client.

Figure 5.3 demonstrates a scenario of a use-case where the Cart session bean, that represents the shopping cart, is used.

Figure 5.3



⇔ = A call managed by the EJB

→ = A call implemented in a class of *the e-commerce platform*

1. The Cart Servlet gets an HTTP Request from the user.
2. The Cart Servlet asks the Cart helper class to compute the tasks.
3. The Cart helper class connects to the Cart EJB objects through its remote home and access the Cart EJB methods.
4. The EJB object methods return the values that were fetched in the Cart object, through the remote interface to the helper class.
5. The Cart helper class returns the values to the Servlet.
6. The Cart Servlet send the computed values to the Cart Servlet HTML helper class.
7. The Cart Servlet HTML helper class asks the Session helper class for the locale of the user. The locale is used for internationalization and localization tasks.
8. The Session helper class asks the Session session beans remote interface for the locale. The remote interface accesses the Session object and gets the value.
9. The Session object methods return the value that was stored in the Session session bean object, through the remote interface to the helper class.
10. The helper class returns the values to the Servlet HTML helper class.
11. The Servlet HTML helper class gets the locale value and calls a resource bundle to open a Language property text file, specific for the users locale, for translation of the final HTML response. The Servlet HTML helper class can also use the Locale for localization (see chapter 6.3).
12. The Language property text file sends the translation back to the Cart Servlet HTML helper class.

13. The Cart Servlet HTML helper class finally sends the HTML response to the client.

5.5 The components of the platform

The components of the platform are:

- HTML files
- Servlets
- Session Java Beans
- Entity Java Beans
- Helper classes

The HTML files

The HTML files are typically being used when no business logic is needed. For example, there is an HTML file for the welcome page of the Web store.

The Servlets

For both the Web store and the management tool, I am using one Servlet class for every function of the interface. One Servlet for every function makes the application code easy to comprehend.
The Servlets in the web store are listed in table 5.1 and those of the management interface are listed in table 5.2.

Table 5.1. The Servlets in the Web store

Servlet	**Description**
BuyServlet	BuyServlet will check if there are any products in the clients shopping cart. If there are any, BuyServlet will present them to the client and the total cost. If the client chooses to buy the products, it will add them to the clients purchase in his profile and delete them from the shopping cart.
CartServlet	Manages the cart in the Web store. CartServlet shows the location in the Catalog, if the "Action" attribute of the HTTP-request to the CartServlet is "Erase" then it erases selected products from the Cart EJB. If the "Action" attribute is "Add" it adds the product to the Cart EJB. When this is done it displays the content of the Cart EJB.
CatalogServlet	Manages the catalog in the Web store. Lets the user jump from between product areas or search for a product by text search. CatalogServlet calls it self every time the client does a new search with the id number of the requested product or product area as an HTTP-request attribute. When a product is presented, CatalogServlet first get the product information from the Product EJB.
IndexServlet	Displays the index of the functionalities of the Web-store. For example, the cart and the catalog are two of these services. The functionalities depend on what kind of user that uses the store. If the user is a "Tour"

	one functionality will be "Sign up for membership!" but this functionality will not be presented for a user that is already a member etc.
LoginServlet	Handles login from the user to the Web store. The LoginServlet first check what kind of user that tries to log in and then checks in the corresponding EJB if the login name and password exists. If a user is found with the correct login name and password, LoginServlet will create a new Session EJB and a new Cart EJB for this user and set the locale of the user in the Session EJB. LoginServlet is also used when a "Tour" user has filled in forms to become a member in NewProfileServlet for creating this new "Profile" user in the Profile EJB. When the information submits this information in the NewProfileServlet, LoginServlet receives it and check the information before creating the new "Profile".
LogoutServlet	Handles logout from the user to the Web store. The LogoutServlet erases the Session EJB and the Cart EJB.
NewProfileServlet	Presents the forms for the user to fill in to make a new "Profile" user or change a "Profile" user. If the user is already a "Profile" user, NewProfileServlet will extract the information about the user from the Profile EJB and displays it.

Table 5.2. Servlets in the Management interface

Servlet	**Description**
CheckServlet	When the "Administrator" user has decided to erase a product, profile or administrator, the CheckServlet does an additional check with the "Administrator" user before erasing the product, profile or administrator from its EJB. Another functionality of the CheckServlet is the updating of the type attribute (the type attribute is defined later in this chapter under Product EJB) of all the products in product database. The type attribute tells if the product is valid or not or if the product is on sale. The product could for example be set to be on sale the 15th October and when that date occurs, the type should be changed from no sale to sale. An administrator should run the updating function in CheckServlet every day.
ManageLoginServlet	Handles login of administrator to the management tool.
ManageProductServlet	The ManageProductServlet presents an interface for managing the product database. The administrator can create, change or erase a product or a product area in the database.
ManageProfileServlet	The ManageProductServlet presents an interface for managing the profile and administrator database. The administrator can create, change or erase a profile or an administrator in the database.

The session beans
I have implemented two session beans: the Cart session bean and the Session session bean.

The *Cart* EJB handles the customers shopping cart. The bean is created when a customer is logging in to the store and it is deleted when the customer is logging out. The ShoppingCart bean is storing the products that the customer puts in the cart. It is also implementing some functions for adding and deleting the products in the cart and to get the contents of the cart.

The *Session* EJB stores the user's locale and looks up the home interfaces to the Cart EJB, Product EJB, Profile EJB and Administrator EJB and stores them as variables. A locale is a Java object that contains two strings: language and country. These variables are used for the internationalization and localization of the interfaces. Like the Cart EJB, the bean is created when a customer is logging in to the store and it is deleted when the customer is logging out.

The entity beans
The store is using the following three entity beans for storing data in the database:

- Product EJB
- Profile EJB
- Administrator EJB

Each of these beans contains a number of parameters (id, name, address etc) that will become rows in the database when the bean is created. The entity bean's parameters are what make the beans *general* so that any kind of Web-store can be built using these beans. The parameters that I have implemented to the entity beans may not be complete but more parameters can be added to the beans. The parameters, that I implemented to the beans, comes from a survey about e-commerce that I did on Icon Medialab.
The entity beans offer a set of functions for retrieving and to set their variables through the remote interface. They also prevent some functions for finding the rows in the database through their home interfaces (search by primary key, search by description etc).
The three entity beans are using container-managed persistence and container-managed transactions.

Product EJB is storing the products in the database.
The parameters of the Product entity bean are listed in table 5.3.

Table 5.3 Parameters of the Product EJB

Parameters	**Java type**	**Description**
Product id	String	Tells were in the pyramid of product and product areas the product is (see chapter 5.6 for more information about the database structure). For example a product id could be " product_cigars_Mexican cigars". The id is unique.
Father	String	Tells what product that is the father to this product. If the product id is " product_cigars_Mexican cigars" then the father would be " product_cigars "
Name	String	The name of the product. If the product id is "

		product_cigars_Mexican cigars " then the name would be " Mexican cigars "
Type	String	See description below.
Sale code	String	Sale code is a string that could be used when a product has different prices to different users.
Valid start	String	Tells the start- and end-date for when the product is valid. The string is on the form YEAR-MONTH-DATE or for example 1999-12-14.
Valid end	String	Tells the end date for when the product is valid. The string is on the same form as the Valid start parameter.
Sale start	String	Tells the start date for when the product has sale. The string is on the same form as the Valid start parameter.
Sale end	String	Tells the end date for when the product has sale. The string is on the same form as the Valid start parameter.
Description	String	The description of the product
Price	int	The price of the product.
Sale	Int	The sale of the product.

The Type parameter is specifying whether it is a product area (see chapter 5.6 for information about products and product areas) or a product and if it is valid (which means that the product is not yet for sale) or if it is on sale. An area is always valid and can not be on sale. The possible types are:

"A" If it is an area.
"PV" If the product is valid and is not on sale (Product Valid).
"PNV" If the product is not valid (Product Not Valid).
"PVSV" If the product is valid and is on sale (Product Valid and Sale Valid).
"PVSNV" If the product is valid and will be on sale in the future (Product Valid and Sale Not Valid).
"PNVSNV" If the product is not valid and will be on sale in the future (Product Not Valid and Sale Not Valid).

Profile EJB is storing the customers to the web store. A customer has access to the store interface but not to the management interface.
The parameters of the Profile entity bean are listed in table 5.4.

Table 5.4 Parameters of the Profile EJB

Parameters	**Java type**	**Description**
Id	Integer	The id number is unique.
Login	String	Login name
Password	String	Login password
First name	String	First name of the customer.
Last name	String	Last name of the customer.
Status	String	Status is a String that could be used in an e-commerce store to

		decide whether to give a person a sale on a product or not.
Profession	String	The profession of the customer.
City	String	The city in the customers address.
Country	String	The country in the customers address.
Telephone	String	The telephone number of the customer.
Email	String	The email of the customer.
Interest 1 – Interest 6	Strings	The interest strings could be used by an e-commerce store to store information about the clients interests
Purchase 1 – Purchase 10	String	The purchase strings could be used in an e-commerce store to store what products that this client has bought.
Sex	String	"M" for Male and "F" for Female.
Marital status	String	Marital status of the customer. The Marital status could be “single”, “married” or “divorced”.
Street	String	The name of the street in the customers address.
City code	String	The city code in the customer's address.
Language	String	The language of the customer.
Age	int	The age of the customer.
Total purchase	int	The total purchase string could be used in an e-commerce store to store the total amount of the products that this client has buy.
Latest login	String	The date when this client was latest logged on to the application.
Favorite color	String	The favorite color of the customer.

Administrator EJB stores the administrators, which are the persons that manages the products, profiles and administrator in the database. An administrator has access to the store interface and to the management interface.
The parameters of the administrator entity bean are listed in table 5.5.

Table 5.5 Parameters of the Administrator EJB

Parameters	**Java type**	**Description**
Id	Integer	The id number is unique.
Login	String	Login name
Password	String	Login password
First name	String	First name of the administrator.
Last name	String	Last name of the administrator.
Country	String	The country of the administrator.
Email	String	The email of the administrator.
Language	String	The language of the administrator.

HTML helper classes
For each Servlet there is an HTML helper class that translate text to the language that the customer desires and sends the HTML response to the client. In this approach of having a helper class that takes care of the localization and output of HTML code, the business logic in the Servlets is clearer. This is also why I did not use JSP technique but Servlets. In JSP classes, the HTML and the Java computing tasks are blended.

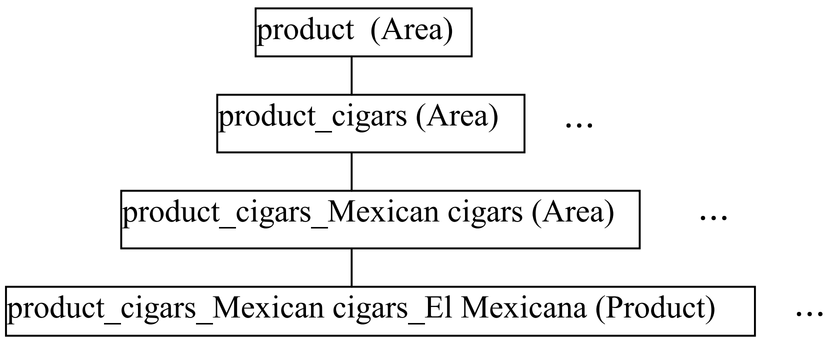
EJB helper classes
For each EJB there is a Helper class that simply gets or sets a value in a bean. It also prevents methods to call the beans create method, remove method, finder method etc. These helper classes have the same purpose for the EJBs as the HTML helper classes for the Sevlets, to make the business logic more clear.

Language property text file
The Language property text files are used for translation from English to the language of the user. (See chapter 6.3)

5.6 The Database structure
I am using Cloudscape database, which is written in Java. It is populated with Profiles, Products and Administrators. There is one table for each entity EJB that is represented in the database and so there are three tables: the Product table, the Profile table and the Administrator table.
Because I am using container-managed persistence, I never see these tables. In container-managed persistence, the developer operates the EJBs which handles the database access and SQL calls to the three tables.
Each entity EJB has a primary key that is unique. The nature of the primary key is set in the deploy tool. In the Profile EJB and the Administrator EJB, I am using a unique Integer as a primary key. In the Product EJB, the primary key, which is a String, is used not only to keep every Product EJB unique but also to describe where in the database structure the product is.

A product EJB object can act either as a *product area* or as a *product*. The Type parameter of the Product EJB tells whether it is a product area or a product (See chapter 5.5 for a definition of the Type parameter).
The database structure of the products in the database is in tree form. The Id attribute of the Product EJB tells where in the tree the product or product area is. On top of the pyramid is the product area "product”. For example one branch of the tree could look like figure 5.2.

Figure 5.2.The database structure.


product (Area)
product_cigars (Area)
…
product_cigars_Mexican cigars (Area)
…
product_cigars_Mexican cigars_El Mexicana (Product)
…

Basic Rules about products and areas:

- Products can not have sons
- An end date can not be before its start date
- The period of a sale must be within the valid-period

The database structure for the Profiles and Administrator is very simple. All the Profiles are stored in the Profile table with a unique key but there is no hierarchy like the Product database structure. The Administrators are stored in the Administrator table in a similar way as the Profiles.

6. Deployment

In this section, I explain how the deployment of the e-commerce platform was done.

6.1 Deployment of the Web- and EJB components

Both the Servlets and the EJBs needed to be deployed in the deploy-tool, that comes with the J2EE application server, to fit in the J2EE application server (see chapter 4.3 for more information on deployment of a J2EE application).

The deploy-tool comes with the J2EE application server. When deploying the EJBs and Servlets for the e-commerce platform in this tool, there are certain steps to be followed. These are:

1. Creating the EAR file.
2. Putting together EJB and WEB modules and specify the structural and assembly information about the modules in the deployment descriptors (see Appendix E that explains what kind of information the deployment descriptors contain).
3. Specify the security roles and the environmental entries.
4. Run the verifier tool in the deployment tool. The verifier tool search for and present errors in the modules.
5. If there are no errors in the verifier tool, execute the deployment of the application.

If the deployment executes without errors, the deployment tool will create the J2EE application (see Appendix D for more details of what happens when a J2EE application is deployed).

When I had created the EAR file, I started with creating two small EJB and WEB modules with just one Servlet in the WEB module and one EJB in the EJB module. I then tried to execute the deployment of these two modules. After correcting errors I finally managed to create a deployed J2EE application. When this was done, I added modules after modules to this J2EE application.

I found that Servlets that call each other must be packaged in the same WEB. This discovery turned out to render the deployment of the Servlets very time-consuming. Every time I needed to make a change in a Servlet, for example change something in the GUI or a computing task, I needed to re-deploy the whole module which meant creating a new deployment descriptor and decide what class files that the WEB module should contain.

I user one WEB module for the Web-store interface and one for the Management interface. To these WEB module, the HTML helper classes, Servlet HTML helper classes, Language property text files and Servlet class files needed to be added.

It was easier to create the EJB modules because they could be small units that only consisted of one EJB and its deployment descriptor. I only needed to re-deploy these modules when there was a change in the EJB and that was not very often.

So, I created one EJB module for each EJB. The final list of modules in the e-commerce platform J2EE application is:

- Management interface WEB module
- Client interface WEB module (the Web-store interface)
- Session EJB module
- Shopping cart EJB module
- Product EJB module
- Profile EJB module
- Administrator EJB module

6.2 Interaction between the WEB and EJB modules

When the EJB modules is being deployed, they all get an identification name. This name will be linked to the actual EJB object in the JNDI lookup service (see chapter 4.1) of the J2EE application server.
As figure 6.1 and figure 6.2 show, in the e-commerce platform an EJB never calls a Servlet. It is always a Servlet that calls an EJB for database or session bean information.
When a Servlet calls an EJB, it first needs to look up the EJB object in the JNDI lookup service. With a reference to the EJB object obtained, one can find the home interface to the EJB object. Here follows an example of how a Servlet looks up the home interface of the Cart session bean:

```
try{
  InitialContext ctx = new InitialContext();(1)
  Object objref = ctx.lookup("Cart");   (2)
  CartHome homeCart = (CartHome)PortableRemoteObject.narrow(objref, CartHome.class);
                                              (3)
}
catch (Exception NamingException) {
  NamingException.printStackTrace();
}
```

1. The InitialContext object is used for the JNDI lookup service to specify where to look for the EJB object. In this case, there are not many references to EJBs in the lookup service and so there is no specification on where to look.
2. The Servlet looks up the Cart EJB object. The J2EE server calls its JNDI lookup interface to find if there is any EJB modules with the identification name "Cart". The lookup service finds the identification name and returns the EJB object of the EJB module that the identification name pointed at.
3. With the reference to the EJB object, the home interface of the Cart session bean is found.

With a reference to a home interface of a session EJB one can create a session bean. This is for example made every time a user logs in to the Web-store when a Session session bean and a Cart session bean is created. The creating of a the Cart session bean looks like this:

```
try{
  Cart ShoppingCart = homeCart.create();
}
catch(javax.ejb.CreateException ce){
  System.out.println("Createexception in CartHelper: createCart ");
}
```

```
catch(java.rmi.RemoteException re) {
  System.err.println("RemoteException in CartHelper: createCart " + re.getMessage());
}
```

With a reference to a home interface of an entity EJB one can search in the database for an entity bean. Here is an example:

```
try{
  Product product1 =homeProduct.findByPrimaryKey("product_cigars_exclusive royal");
}
catch…
```

This call will search in the database for a product with the Id "product_cigars_exclusive royal".
When the product is found one can use its business methods defined in the remote interface. For example, to get the description of a product the following call should be used:

```
String description = Product1.getDescription();
```

6.3 Internationalization and Localization

The internationalization and the localization were an important part of the project. I realized this part by using the localization features of the Java programming language and by setting the locale of the user in the Session EJB that is created when the user is logging in. Both the web store interface and the management interface is internationalized and localized.
If the user is a known profile that is stored in the database, the language and country attributes of the profile are collected from the database. If the user is logging in to the web store as a “tour”, the user has to choose a language before logging in.

Internationalization

The locale of the user is set when the user logs in to the Web-store or management interface. Before the user gets back an HTTP-response when requesting a Web page, the page will pass through an HTML helper class. This class will check the locale of the user in the Session EJB, look up a resource bundle for that specific locale and Web page and translate the text into the language set in the locale.

A resource bundle is a Java type that contains a text file. The creation method of a resource bundle takes two parameters: A string that is the general name of the text file and a Locale. In the following example, the Servlet html helper looks up a title and prints out the html title.

```
Locale locale = SessionHelper.getLocale(ManageLoginServlet.session);
ResourceBundle titleText = ResourceBundle.getBundle("titleText", locale);
String LANG_Title = titleText.getString("Title");
out.println("<TITLE>" + LANG_Title + "</TITLE>");
```

I wrote the e-commerce platform for thee languages: English, French and Swedish. In order for this to work I had to write three “titleText” text files, one for every language. These three files have to have the names: titleText_en, titleText_fr and titleText_sw. Otherwise, the resource bundle would not find them. Each of these files contains a row starting with “Title = “ and the translation. The titleText_fr file would have the row “Title = Titre”. I created these three text files for every Servlet. For example the loginText_en, loginText_fr and loginText_sw contains the text output from the login Servlet.

With this approach, it is very easy to add a language to the application. One only has to write a couple of new text files and nothing more. It is a little more complicated when one wants to change the output text. Then one has to add the new words to all the language files (in my case three). If I was to change the title, I had to open and edit the three titleText files.

The more traditional approach is to have a Servlet or JSP file for every language. If one has a login Servlet for example that prints a welcome message to the screen, one would need to have three login Servlets if there were three languages. I didn’t like this idea mainly because that if one would add a new language, one had to rewrite (or copy and paste) the hole Servlet and then change the language in it. If one was to change the code of the Servlet, one had to do it in as many Servlets as there were languages.

Localization
The prices of the products in the web store are given in the currency that depends on the country of the user. Again the Servlet html helper looks up the Locale is the Session session bean. It gets the Country attribute in the locale and passes it to a function in a helper class that calculates and returns the value of the product I the currency of that country.

6.4 Security
As I wrote in the chapter 4.5 about security in the J2EE platform, before all the mapping of security roles to users can work, one has to create the different groups of users with a command-line tool to manage the users. I did not like that way of handling the grouping of users. It meant that every time a new profile or administrator was created, someone had to manually put her profile in to a security group with this tool. In the e-commerce platform however, a user logged in as a tour can create a profile by herself and it would be a difficult job to keep count of all the new profiles and putting them into security groups.

I decided to manage the security by using the already existing groups in the database: profiles and administrators. The profiles and administrators are authenticated by filling in a login-password form. The only difference in security is that only the administrator can use the management tool for managing the database over products, profiles and administrators. That meant that I somehow had to be sure that a profile could not use the management interface Servlets.
I assured this by setting a parameter with the value “Administrator” in the Session session bean when the administrator is logging in to the management interface. Every time a Servlet is used in the management interface, the Servlet asks for this parameter in the Session session bean and checks that it is equal to “Administrator” If it is not, the Servlet stops all operations and writes a error message to the screen.

7. Conclusions

The project turned out to be quite big and I did not manage to deliver complete software but I think that I learned a lot on the way. I did a couple of presentations about EJBs, Servlets, and the J2EE application server for the technology team in the office at Icon Medialab, San Francisco. It was appreciated.

I have learned how to use the J2EE Platform with deploying etc. I also learned how to work with Servlets and Java Enterprise Beans and how to use the RMI and JNDI technology.

I developed a prototype for the e-commerce platform with two user interfaces using the J2EE application server. The prototype can be deployed on any of the platforms that the J2EE application server supports and use any of the suported databases (see Appendix C - Technical requires for the J2EE platform).

What are the positive aspects about developing applications on the J2EE platform?
One does not need to worry about transaction management or database connections and best of all: it is not necessary to write any SQL code. The J2EE EJB container can produce the SQL code itself.
Other positive aspects are:

- Easy to manage security.
- Produces scalable applications.
- Produces applications that can are remoteble to all sorts of clients thanks to the RMI and RMI-IOOP structure of the J2EE platform.

What are the negative aspects about developing applications on the J2EE platform?
Deployment is VERY time-consuming. When deploying the Web-store, for example, I need to put all the Servlets in one Web module as they all use each other. Whenever I changed a Servlet or a class in the Web store interface, I needed to re-deploy the entire module to be able to see if the changes were correct.

The new release, version 1.2.1, of the Application Deployment Tool has a new re-deployment feature that let the deployer "hot deploy" the J2EE application.
In this version, the developer does not need to re-deploy the whole Web module or EJB module if a class is altered. Instead it uses the same module but changes the old classes for the new that has been changed.

8. Recommendations

Application servers are very powerful tools to work with. The J2EE application server is easy to use once one learned to implement the EJBs and how to deploy the J2EE application.
The new version of the Application Deployment Tool to the J2EE platform makes it much easier to deploy the J2EE application than with the previous version that I used. This new deployment tool makes the J2EE platform even more powerful for developing commercial products.

Sun provides a developers-connection with forums at http://forum.java.sun.com/ where developers can ask questions about Java APIs. The question is normally answered the same day or in some cases the next day.
I can specially recommend the EJB forum and the J2EE forum.

There is a very good Java tutorial online that is possible to download at:
http://java.sun.com/docs/books/tutorial/

9. References

- Database Programming with JDBC and JAVA,1997 O'REILLY by George Reese
- The complete Reference JAVA 1.1 second edition, 1998 Osborne by Patrick Naughton and Herbert Schildt
- JavaSofts Online Tutorial, http://java.sun.com/docs/books/tutorial/
- Java Developer Connection and the Enterprise JavaBeans forum at http://forum.java.sun.com/
- Sun BluePrints Design Guidelines for J2EE
- Simplified Guide to the Java 2 Platform, Enterprise edition. Copyright 1999 by SUN INC.
- Java 2 SDK, Enterprise Edition Release Notes
- Fundamentals of Java RMI at SUN: http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html#IntroRMI
- Enterprise JavaBeans tutorial at SUN: http://developer.java.sun.com/developer/onlineTraining/Beans/EJBTutorial/index.html
- Java Servlet Specification, v2.2 at SUN, www.javasoft.com.
- Containers by Neil Ward-Dutton is an article in Component Strategies & Architectures January 2000. The article is available at http://www.adtmag.com.
- A Detailed Comparison of CORBA, DCOM and Java/RMI by Gopalan Suresh Raj at http://www.execpc.com/~gopalan/misc/compare.html
- Overview of the concept of e-commerce at http://www.premiersite.com/html/new_call_e-commerce.htm?support/commerce/whatis

Appendix A - Services provided by the EJB container

Transaction Management
When a client invokes a method in an enterprise bean, the container intervenes in order to manage the transaction. Because the container manages the transaction, you do not have to code transaction in the enterprise bean. The code that is required to control distributed transactions can be quite complex. Instead of writing complex code, the developer simply declare the enterprise bean's transactional properties in the deployment descriptor file. The container reads the file and handles the enterprise bean's transactions.

Remote Client Connectivity
The container manages the low-level communications between clients and enterprise beans. After an enterprise bean has been created, a client invokes methods on it as if it were in the same virtual machine.

Security
The container permits only authorized clients to invoke an enterprise bean's methods. Each client belongs to a particular role, and each role is permitted to invoke certain methods. The developer declares the roles and the methods they may invoke in the enterprise bean's deployment descriptor. Because of this declarative approach, the developer does not need to code routines that enforce security.

Life Cycle Management
An enterprise bean passes through several states during its lifetime. The container creates the enterprise bean, moves it between a pool of available instances and the active state, and finally, removes it. Although the client calls methods to create and remove an enterprise bean, the container performs these tasks behind the scenes.

Database Connection Pooling
Obtaining a database connection is time-consuming and the number of connections may be limited. To alleviate these problems, the container manages a pool of database connections. An enterprise bean can quickly obtain a connection from the pool. After the bean releases the connection, it may be re-used by another bean.

Appendix B - Services provided by the J2EE platform

JDBC 2.0
JDBC is an API for database connectivity between the J2EE platform and a wide range of data sources. JDBC allows:

- Load and configure a database driver on a client
- Perform connection and authentication to a database server
- Manage transactions
- Move SQL statements to a database engine for preprocessing and execution
- Inspect the results from Select statements

Java Transaction API and Service
The Java Transaction API (JTA) specifies standard Java interfaces between a transaction manager and the transactional application, the J2EE server, and the manager that controls access to the shared resources affected by the transactions.

Java Naming and Directory Interface
The Java Naming and Directory Interface (JNDI) is an API that provides naming and directory functionality. The JNDI provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using the JNDI, an application can store and retrieve any type of named Java object. In particular, a J2EE application uses JNDI to find interfaces used to create enterprise beans, JTA User Transaction objects, JDBC Data Source objects, and message connections.

Java Message Service
The Java Message Service (JMS) is an API for using enterprise-messaging systems.

Java Mail
The Java Mail API provides a set of abstract classes and interfaces that comprise an electronic mail system. The abstract classes and interfaces support many different implementations of message stores, formats, and transports. Many simple applications will only need to interact with the messaging system through these base classes and interfaces. The abstract classes in Java Mail can be subclasses to provide new protocols and add functionality when necessary.

Java bean Activation Framework
The Java bean Activation Framework (JAF) integrates support for MIME data types into the Java platform. The JAF is used by Java Mail to handle the data included in email messages; typical applications will not need to use the JAF directly, although applications making sophisticated use of email may need it.

Appendix C - Technical requires for the J2EE platform

Supported Platforms
Solaris ^{TM} Operating Environment, version 2.6
Windows NT, version 4.0

Supported Databases and JDBC Drivers
Oracle8 Server, version 8.05
Microsoft SQL Server, versions 6.5, 7.0
Cloudscape, version 3.0

Limitations
The J2EE platform requires at least 128 MB of memory. Memory problems have occurred when a large number (> 16) of J2EE applications are deployed, or when a large number (> 1000) of enterprise beans are instantiated.

Appendix D - Deployment in detail

Deployment of enterprise beans consists of following sub-tasks:

1. The J2EE server compiles the stubs and skeletons for each enterprise bean.
2. The J2EE server sets up the security environment to host the enterprise beans according to their deployment descriptor.
3. The J2EE server sets up the transaction environment for the enterprise beans according to their deployment descriptor.
4. The J2EE server registers the enterprise beans, their environment properties, resources references and so on, in the JNDI name space.
5. For enterprise beans that need container-managed persistence, the J2EE server creates database tables.

Deployment of Web components consists of following sub-tasks:

1. The server transfers all the con-tents of the Web components underneath the document root of the Web server.
2. The J2EE server initializes the security environment of the application.
3. The J2EE server registers the environment properties, resource references, and EJB references in the JNDI name space.
4. The J2EE server sets up the environment for the Web application. For example, it performs the alias mappings, and configures the Servlet context parameters.
5. The J2EE server pre-compiles the JSP pages as specified in the deployment descriptor.

Appendix E - Deployment descriptors in detail

An EJB deployment descriptor contains the following structural information:

- Identification. Specifies the name and description of the EJB module.
- Remote and home interfaces. Specifies the address of the two interfaces.
- Type. The type can be either stateful session, stateless session or entity
- Environment entries
- References to other enterprise beans
- References to external resources
- References to security roles
- (Only for session EJBs) Whether transactions are managed by the bean or by the container
- (Only for entity EJBs) Whether persistence is managed by the bean or by the container
- (Only for entity EJBs) Primary key class

An EJB deployment descriptor contains the following assembly information:

- Security roles
- Method permissions. These permissions specify which security roles are permitted to execute a given method on the EJB.
- Transaction attributes. These attributes are used for container-managed EJBs to decide what kind of transaction should be used for each method in the EJB.

A Web component deployment descriptor contains the following elements:

- Identification. Specifies the name and description of the WEB module.
- Servlet initialization parameters
- Servlet context parameters
- Session configuration
- Localization configuration
- Servlet and JSP definitions
- Servlet and JSP mappings
- MIME type mappings
- Welcome and error page list
- Tag library information
- Security roles
- Security constraints, which map security roles and authorization methods to
- collections of Web resources
- Environment entries
- References to enterprise beans
- References to external resources
- References to security roles

Appendix F – Glossary

.ear file A JAR file that contains a J2EE application.
.jar file A JAR file that contains a EJB module
.war file A JAR file that contains a Web module.
API Application Programming Interface
Applet A Java component that typically executes in a Web browser.
Application client A first-tier Java client program.
Bean-managed persistence When the calls to the database are implemented in the entity bean.
Bean-managed transaction When the transaction management is implemented in the entity bean.
Component An application-level software unit supported by a container.
Container An entity part of a application server that provides life cycle management, security, deployment, and runtime services to components.
Container-managed persistence When the SQL-calls to the database is managed by the enterprise bean's container.
Container-managed transaction When the transactions are managed by the container
CORBA Common Object Request Broker Architecture. A language-independent, distributed object model specified by the Object Management Group (OMG).
Deployment The process when it is decided how the components are going to work together and what responsibility the container should take using the deployment tool.
Deployment descriptor An XML file provided with each module and application that describes how they should be deployed.
Distributed application An application made up of components running in separate runtime environments, usually on different platforms connected via a network.
EJB container A container that implements a runtime environment for enterprise beans that includes security, concurrency, life cycle management, transaction, deployment, and other services. An EJB container is provided by an EJB- or a J2EE-server.
EJB .jar file A JAR archive that contains an EJB module.
EJB module A software unit that consists of one or more enterprise beans and an EJB deployment descriptor.
EJB server Software that provides services to an EJB container. For example, an EJB container typically relies on a transaction manager that is part of the EJB server to perform. The J2EE architecture assumes that an EJB container is hosted by an EJB server from the same vendor. An EJB server may host one or more EJB containers.
Enterprise JavaBeans (EJB). A component architecture for the development and deployment of object-oriented, distributed, applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, multi-user and secure.
Entity bean. An enterprise bean that represents persistent data in a database. A primary key identifies an entity bean. If the container in which an entity bean is hosted crashes, the entity bean, its primary key, and any remote references survive the crash.
Finder method A method defined in the home interface and invoked by a client to find an entity bean.
Home interface One of two interfaces for an enterprise bean. The home interface defines methods for creating and removing an enterprise bean. For entity beans, the home interface also defines finder methods.
IIOP Internet Inter-ORB Protocol. A protocol used for communication between CORBA objects.
J2EE application A J2EE application is made up of several modules packaged into an .ear file with a J2EE application deployment descriptor. J2EE applications are often distributed applications.
J2EE server The runtime portion of a J2EE product. A J2EE server provides a Web- and a EJB-container.
Java 2 Platform, Enterprise Edition (J2EE platform) An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, APIs, and protocols that provide the functionality for developing multi-tiered, Web-based applications.
Java 2 SDK, Enterprise Edition Sun's implementation of the J2EE platform. This implementation provides an operational definition of the J2EE platform.
Java Naming and Directory Interface (JNDI) An API that provides naming and directory functionality.
JavaBean A Java class that can be manipulated in a visual builder tool and composed together into applications. A JavaBeans component must be programmed after certain conventions.
Java Server Pages (JSP) An extensible Web technology that uses template data, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements.
JDBC API for database connectivity between the Java platform a data source.

Module A software unit that consists of one or more J2EE components of the same container type and one deployment descriptor of that type. There are three types of modules: EJB, Web, and application client.
Primary key An object that uniquely identifies an entity bean within a home.
Remote interface One of two interfaces for an enterprise bean. The remote interface defines the business methods that a client can access.
RMI Remote Method Invocation. A technology that allows an object running in one Java virtual machine to invoke methods on an object running in a different Java virtual machine.
RMI-IIOP [RMI over IIOP] RMI over IIOP is a version of RMI implemented to use the CORBA IIOP protocol. RMI over IIOP provides a bridge to CORBA implemented objects.
Servlet A Java program that extends the functionality of a Web server, generating dynamic content and interacting with Web clients using a request-response model.
Session bean An enterprise bean that is created by a client and that usually exists only for the duration of a single client/server session. A session bean performs operations, such as calculations or accessing a database, for the client. Session bean objects can be either stateless or they can maintain a state. If a crash in the system, the state of a stateful session bean is not recoverable.
SQL Structured Query Language. The standardized relational database language for defining database objects and manipulating data.
Stateful session bean A session bean that contains a state. Data can be stored in a stateful session bean but only as long as the session bean lives. When the session bean is terminated, the state will be lost.
Stateless session bean A session bean without state. All instances of a stateless session bean are identical.
Transaction Transactions enable multiple users to access the same data.
Web component A component that provides services in response to requests. A Web component in the J2EE architecture can be a Servlet, JSP- or HTML file or an applet.
Web container A container that implements a runtime environment for Web components that includes security, concurrency, life cycle management, transaction, deployment, and other services. A Web container is provided by a Web- or a J2EE-server.
Web server A Web server hosts Web sites, provides support for HTTP and other protocols, and executes server-side programs (such as CGI scripts or Servlets). In the J2EE architecture, a Web server provides services to a Web container. For example, a Web container typically relies on a Web server to provide HTTP message handling. The J2EE architecture assumes that a Web container is hosted by a Web server from the same vendor. A Web server may host one or more Web containers.

Appendix G - Class documentation

Some of the most important classes in the prototype of the e-commerce platform are here listed in alphabetic order.

Beans.administrator

Interface Administrator

public interface Administrator
extends javax.ejb.EJBObject
Defines the business methods of the Administrator EJB

Method Detail

The following business methods are implemented in the AdministratorEJB

public int **getId**() throws java.rmi.RemoteException
public void **setId**(int id) throws java.rmi.RemoteException
public java.lang.String **getLogin**() throws java.rmi.RemoteException
public void **setLogin**(java.lang.String login) throws java.rmi.RemoteException
public java.lang.String **getPassword**() throws java.rmi.RemoteException
public void **setPassword**(java.lang.String password)throws java.rmi.RemoteException
public java.lang.String **getFirstName**() throws java.rmi.RemoteException
public void **setFirstName**(java.lang.String firstName) throws java.rmi.RemoteException
public java.lang.String **getLastName**() throws java.rmi.RemoteException
public void **setLastName**(java.lang.String lastName) throws java.rmi.RemoteException
public java.lang.String **getEmail**()throws java.rmi.RemoteException
public void **setEmail**(java.lang.String email) throws java.rmi.RemoteException
public java.lang.String **getLanguage**() throws java.rmi.RemoteException
public void **setLanguage**(java.lang.String language) throws java.rmi.RemoteException
public java.lang.String **getCountry**() throws java.rmi.RemoteException
public void **setCountry**(java.lang.String country) throws java.rmi.RemoteException

Beans.administrator

Class AdministratorEJB

```
java.lang.Object
  |
  +--Beans.administrator.AdministratorEJB
```

public class AdministratorEJB
extends java.lang.Object
implements javax.ejb.EntityBean
Implements the bussiness methods and creation methods of the Administrator EJB

Field Detail

- public java.lang.Integer **id**
- public java.lang.String **login**
- public java.lang.String **password**
- public java.lang.String **firstName**
- public java.lang.String **lastName**
- public java.lang.String **email**

- public java.lang.String **language**
- public java.lang.String **country**

Constructor Detail

AdministratorEJB
public **AdministratorEJB**()

Method Detail

ejbCreate
public void **ejbCreate**(int id, java.lang.String login, java.lang.String password, java.lang.String firstname, java.lang.String lastname, java.lang.String email, java.lang.String language, java.lang.String country) throws javax.ejb.CreateException
ejbCreate creates an Administrator EJB if parameters are correct When ejbCreate is called, the EJB container creates an Administrator EJB. The creation method of a EJB must be named ejbCreate() and can not return a value.
Parameters:
The parameters are the same as the instance variables of the Administrator

getId
public int **getId**() throws java.rmi.RemoteException
Returns the Id of the Administrator

setId
public void **setId**(int id) throws java.rmi.RemoteException
Sets a new Id to the Administrator
Parameters:
id - the new Id of the Administrator

getLogin
public java.lang.String **getLogin**() throws java.rmi.RemoteException
Returns the login name of the Administrator

setLogin
public void **setLogin**(java.lang.String login) throws java.rmi.RemoteException
Sets a new login name to the Administrator
Parameters:
login - the new login name of the Administrator

getPassword
public java.lang.String **getPassword**() throws java.rmi.RemoteException
Returns the password of the Administrator

setPassword
public void **setPassword**(java.lang.String password) throws java.rmi.RemoteException
Sets a new password to the Administrator
Parameters:
password - the new password of the Administrator

getFirstName
public java.lang.String **getFirstName**() throws java.rmi.RemoteException
Returns the first name of the Administrator

setFirstName
public void **setFirstName**(java.lang.String firstName) throws java.rmi.RemoteException
Sets a new first name to the Administrator
Parameters:
firstName - the new first name of the Administrator

getLastName
public java.lang.String **getLastName**() throws java.rmi.RemoteException
Returns the last name of the Administrator

setLastName
public void **setLastName**(java.lang.String lastName) throws java.rmi.RemoteException
Sets a new last name to the Administrator
Parameters:
lastName - the new last name of the Administrator

getEmail
public java.lang.String **getEmail**() throws java.rmi.RemoteException
Returns the email of the Administrator

setEmail
public void **setEmail**(java.lang.String email) throws java.rmi.RemoteException
Sets a new email to the Administrator
Parameters:
email - the new email of the Administrator

getLanguage
public java.lang.String **getLanguage**() throws java.rmi.RemoteException
Returns the language of the Administrator

setLanguage
public void **setLanguage**(java.lang.String language) throws java.rmi.RemoteException
Sets a new language to the Administrator
Parameters:
language - the new language of the Administrator

getCountry
public java.lang.String **getCountry**()
throws java.rmi.RemoteException
Returns the country of the Administrator

setCountry
public void **setCountry**(java.lang.String country)
throws java.rmi.RemoteException
Sets a new country to the Administrator
Parameters:
country - the new country of the Administrator

The following methods must be overridden because AdministratorEJB implements EntityBean :

public void **setEntityContext**(javax.ejb.EntityContext context)
public void **ejbRemove**()
public void **ejbActivate**()
public void **ejbPassivate**()
public void **unsetEntityContext**()
public void **ejbLoad**()
public void **ejbStore**()
public void **ejbPostCreate**(int id, java.lang.String login, java.lang.String password, java.lang.String firstname, java.lang.String lastname, java.lang.String email, java.lang.String language, java.lang.String country)

Beans.administrator

Interface AdministratorHome

public interface AdministratorHome
extends javax.ejb.EJBHome
Defines the creation and finder methods of the Administrator EJB

Method Detail

create
public Administrator **create**(int id, java.lang.String login, java.lang.String password, java.lang.String firstname, java.lang.String lastname, java.lang.String email, java.lang.String language, throws java.rmi.RemoteException, javax.ejb.CreateException
The create method is implemented in the AdministratorEJB as ejbCreate()

The following finder-methods are implemented by the EJB container in the deployment phase:

FindByPrimaryKey public **Administrator** findByPrimaryKey(java.lang.Integer Id) throws javax.ejb.FinderException, java.rmi.RemoteException

FindByLogin public **java.util.Collection** findByLogin(java.lang.String login, java.lang.String password) throws javax.ejb.FinderException, java.rmi.RemoteException

FindByName public **java.util.Collection** findByName(java.lang.String firstName, java.lang.String lastName) throws javax.ejb.FinderException, java.rmi.RemoteException

FindByEmail public **java.util.Collection** findByEmail(java.lang.String email) throws javax.ejb.FinderException, java.rmi.RemoteException

ClientInterface.Servlets

Class BuyServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.BuyServlet
```

public class BuyServlet
extends javax.servlet.http.HttpServlet
BuyServlet will check if there are any products in the clients shopping cart. If there are any, BuyServlet will present them to the client and the total cost. If the client chooses to buy the products, it will add them to the clients purchase in his profile and delete them from the shoppingcart.

Constructor Detail

BuyServlet
public **BuyServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

Beans.cart

Interface Cart

public interface Cart
extends javax.ejb.EJBObject
Defines the business methods of the Cart EJB

Method Detail

The following business methods are implemented in the CartEJB

public void **addItem**(Product product) throws java.rmi.RemoteException
public void **addItemPrice**(java.lang.Double productPrice) throws java.rmi.RemoteException
public void **removeItem**(Product product) throws java.rmi.RemoteException
public void **removeItemPrice**(java.lang.Double productPrice)throws java.rmi.RemoteException
public java.util.Vector **getContents**() throws java.rmi.RemoteException
public java.util.Vector **getContentPrices**() throws java.rmi.RemoteException
public void **removeItemAt**(int n) throws java.rmi.RemoteException
public void **removeItemPriceAt**(int n) throws java.rmi.RemoteException

Beans.cart

Class CartEJB

```
java.lang.Object
  |
  +--Beans.cart.CartEJB
```

public class CartEJB
extends java.lang.Object
implements javax.ejb.SessionBean
Implements the bussiness methods and creation methods of the Cart EJB.

Constructor Detail

CartEJB
public **CartEJB**()

Method Detail

ejbCreate
public void **ejbCreate**() throws javax.ejb.CreateException, java.rmi.RemoteException
When ejbCreate is called, the EJB container creates a CartEJB. The creation method of a EJB must be named ejbCreate() and can not return a value.

addItem
public void **addItem**(Product product) throws java.rmi.RemoteException
Adds a product to the shopping cart.
Parameters:
product - the product

addItemPrice
public void **addItemPrice**(java.lang.Double productPrice) throws java.rmi.RemoteException
Adds a product's price to the price table in the shopping cart. The price table remembers the prices of all the products in the shopping cart.
Parameters:
productPrice - the price of a product

removeItem
public void **removeItem**(Product product) throws java.rmi.RemoteException
Removes a product from the shopping cart
Parameters:
product - the product

removeItemPrice
public void **removeItemPrice**(java.lang.Double productPrice) throws java.rmi.RemoteException
Removes a price from the price table in the shopping cart The price table remembers the prices of all the products in the shopping cart.
Parameters:
productPrice - the price of a product

removeItemAt
public void **removeItemAt**(int n) throws java.rmi.RemoteException
Removes the n:th product in the shopping cart
Parameters:
n - the n:th product in the shopping cart

removeItemPriceAt
public void **removeItemPriceAt**(int n) throws java.rmi.RemoteException
Removes the n:th price the price table in the shopping cart The price table remembers the prices of all the products in the shopping cart.
Parameters:
n - the n:th product in the shopping cart

getContents
public java.util.Vector **getContents**() throws java.rmi.RemoteException
returns the content of the shopping cart
Returns:
a vector of the content of the shopping cart

getContentPrices
public java.util.Vector **getContentPrices**() throws java.rmi.RemoteException
returns the price table of the shopping cart The price table remembers the prices of all the products in the shopping cart.
Returns:
a vector of the price table of the shopping cart

The following methods must be overridden because CartEJB implements SessionBean :

public void **ejbRemove**() throws java.rmi.RemoteException
public void **ejbActivate**() throws java.rmi.RemoteException
public void **ejbPassivate**() throws java.rmi.RemoteException
public void **setSessionContext**(javax.ejb.SessionContext sc) throws java.rmi.RemoteException
public void **unsetSessionContext**() throws java.rmi.RemoteException
public void **ejbLoad**() throws java.rmi.RemoteException

public void **ejbStore**() throws java.rmi.RemoteException

Beans.cart

Interface CartHome

public interface CartHome
extends javax.ejb.EJBHome
Defines the creation and finder methods of the Cart EJB

Method Detail

create
public Cart **create**() throws java.rmi.RemoteException, javax.ejb.CreateException
The create method is implemented in the CartEJB as ejbCreate()

ClientInterface.Servlets

Class CartServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.CartServlet
```

public class CartServlet
extends javax.servlet.http.HttpServlet
CartServlet shows the location in the Catalog, if the "Action" attribute of the request to the CartServlet is "Erase" then it erases selected products from the Cart EJB. If the "Action" attribute is "Add" it adds the product to the Cart EJB. When this is done it desplays the content of the Cart EJB.

Constructor Detail

CartServlet
public **CartServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

ClientInterface.Servlets

Class CatalogServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.CatalogServlet
```

public class CatalogServlet
extends javax.servlet.http.HttpServlet
Manages the catalog in the Web store. Lets the user jump from between product areas or search for a product by text search. CatalogServlet calls it self everytime the client does a new search with the id number of the requested product or product area as a HTTP-request attribute.When a product is presented, CatalogServlet first get the product information from the Product EJB.

Field Detail

products
public static java.util.Collection **products**

location
public static java.lang.String **location**

Constructor Detail

CatalogServlet
public **CatalogServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides: init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides: doGet in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides: destroy in class javax.servlet.GenericServlet

ManagementInterface.Servlets

Class CheckServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
```

```
          |
          +--ManagementInterface.Servlets.CheckServlet
```

public class CheckServlet
extends javax.servlet.http.HttpServlet
When the "Administrator" user has desided to erase a product, profile or administrator, the CheckServlet does an additional check with the "Administrator" user before erasing the product, profile or administrator from its EJB. Another functionalety of the CheckServlet is the updating of the type attibute of all the products in product database. The type attribute tells if the product is valid or not or if the product is on sale. The product could for example be set to be on sale the 15th october and when that date occures, the type should be changed from no sale to sale. The updating function in CheckServlet should be run every day by a aministrator.

Field Detail

productsToErase
public static java.util.Vector **productsToErase**

Constructor Detail

CheckServlet
public **CheckServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request,javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

doPost
public void **doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doPost in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

UpdateProducts
public void **UpdateProducts**(Product product)

ClientInterface.Servlets

Class IndexServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.IndexServlet
```

public class IndexServlet
extends javax.servlet.http.HttpServlet
Displays the index of the functionaleties of the Web-store. For example, the cart and the catalog are two of these services. The functionaleties depends on what kind of user that uses the store. If the user is a "Tour" one functionalety will be "Sign up for memberchip!" but this functionalety will not be presented for a user that is already a member etc.

Constructor Detail

IndexServlet
public **IndexServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

ClientInterface.Servlets

Class LoginServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.LoginServlet
```

public class LoginServlet
extends javax.servlet.http.HttpServlet
Handles login from the user. The LoginServlet first check what kind of user that tries to log in and then checks in the coresponding EJB if the login name and password exists. If a user is found with the correct login name and password, LoginServlet will create a new Session EJB and a new Cart EJB for this user and set the locale of the user in the Session EJB. LoginServlet is also used when a "Tour" user has filled in forms to become a member in NewProfileServlet for creating this new "Profile" user in the Profile EJB. When the information submits this information in the NewProfileServlet, LoginServlet recieves it and check the information before creating the new "Profile".

Field Detail

session
public static Session **session**

customer
public static Profile **customer**

cart
public static Cart **cart**

Constructor Detail

LoginServlet
public **LoginServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doPost
public void **doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doPost in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

ClientInterface.Servlets

Class LogoutServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.LogoutServlet
```

public class LogoutServlet
extends javax.servlet.http.HttpServlet
Handles logout from the user to the Web store. The LogoutServlet erases the Session EJB and the Cart EJB.

Constructor Detail

LogoutServlet
public **LogoutServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

ManagementInterface.Servlets

Class ManageLoginServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ManagementInterface.Servlets.ManageLoginServlet
```

public class ManageLoginServlet
extends javax.servlet.http.HttpServlet
Handles login of administrator to Management Tool.

Field Detail

session
public static Session **session**

Constructor Detail

ManageLoginServlet
public **ManageLoginServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

doPost
public void **doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException

Overrides:
doPost in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

securetyCheck
public static boolean **securetyCheck**(java.io.PrintWriter out)

ManagementInterface.Servlets

Class ManageProductServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ManagementInterface.Servlets.ManageProductServlet
```

public class ManageProductServlet
extends javax.servlet.http.HttpServlet
The ManageProductServlet presents an interface for manageing the product database. The administrator can create, change or erase a product or a product area in the database.

Field Detail

public static Product **selectedProduct**
public static java.lang.String[] **info**
public static java.lang.String **Key**
public static java.lang.String[] **Keys**
public static java.util.Vector[] **Rows**
public static java.lang.String **Action**
public static int **index**

Constructor Detail

ManageProductServlet
public **ManageProductServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

doPost
public void **doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doPost in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

getInfo
public void **getInfo**(java.lang.String type)

addProduct
public static void **addProduct**(java.lang.String type)

setSelectedProduct
public static void **setSelectedProduct**()

setInfoNull
public static void **setInfoNull**()

setNewRow
public static void **setNewRow**(java.lang.String father, int row)

ManagementInterface.Servlets

Class ManageProfileServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ManagementInterface.Servlets.ManageProfileServlet
```

public class ManageProfileServlet
extends javax.servlet.http.HttpServlet
The ManageProductServlet presents an interface for manageing the profile and administrator database. The administrator can create, change or erase a profile or a administrator in the database.

Field Detail

public static java.lang.String[] **info**
public static Profile **selectedProfile**
public static java.util.Collection **profiles**
public static Administrator **selectedAdmin**
public static java.util.Collection **admins**
public static java.lang.String **typeOfSearch**
public static java.lang.String **searchStr1**
public static java.lang.String **searchStr2**

Constructor Detail

ManageProfileServlet
public **ManageProfileServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

doPost
public void **doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doPost in class javax.servlet.http.HttpServlet

setSearchStrings
public void **setSearchStrings**(java.lang.String TypeOfSearch, java.lang.String SearchStr1, java.lang.String SearchStr2)

setInfoNull
public void **setInfoNull**()

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

getInfo
public void **getInfo**(java.lang.String type)

ClientInterface.Servlets

Class NewProfileServlet

```
java.lang.Object
  |
  +--javax.servlet.GenericServlet
        |
        +--javax.servlet.http.HttpServlet
              |
              +--ClientInterface.Servlets.NewProfileServlet
```

public class NewProfileServlet
extends javax.servlet.http.HttpServlet
Presents the table for the client with input fields to fill in in order to make a new or change a profile. if the client comes from LoginServlet to change the Profile, it extract all the information about the client from his profile and displays it.

Constructor Detail

NewProfileServlet
public **NewProfileServlet**()

Method Detail

init
public void **init**(javax.servlet.ServletConfig config) throws javax.servlet.ServletException
Overrides:
init in class javax.servlet.GenericServlet

doGet
public void **doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response) throws javax.servlet.ServletException, java.io.IOException
Overrides:
doGet in class javax.servlet.http.HttpServlet

destroy
public void **destroy**()
Overrides:
destroy in class javax.servlet.GenericServlet

Beans.product

Interface Product

public interface Product
extends javax.ejb.EJBObject
Defines the bussiness methods of the Product EJB

Method Detail

The following business methods are implemented in the ProductEJB

public java.lang.String **getId**() throws java.rmi.RemoteException
public void **setId**(java.lang.String id) throws java.rmi.RemoteException
public java.lang.String **getFather**() throws java.rmi.RemoteException
public void **setFather**(java.lang.String father) throws java.rmi.RemoteException
public java.lang.String **getName**() throws java.rmi.RemoteException
public void **setName**(java.lang.String name) throws java.rmi.RemoteException
public java.lang.String **getType**() throws java.rmi.RemoteException
public void **setType**(java.lang.String type) throws java.rmi.RemoteException
public java.lang.String **getSalecode**() throws java.rmi.RemoteException
public void **setSalecode**(java.lang.String Salecode) throws java.rmi.RemoteException
public java.lang.String **getValidstart**() throws java.rmi.RemoteException
public void **setValidstart**(java.lang.String validstart) throws java.rmi.RemoteException
public java.lang.String **getValidend**() throws java.rmi.RemoteException
public void **setValidend**(java.lang.String validend) throws java.rmi.RemoteException
public java.lang.String **getSalestart**() throws java.rmi.RemoteException
public void **setSalestart**(java.lang.String salestart) throws java.rmi.RemoteException
public java.lang.String **getSaleend**() throws java.rmi.RemoteException
public void **setSaleend**(java.lang.String saleend) throws java.rmi.RemoteException
public java.lang.String **getDescription**() throws java.rmi.RemoteException
public void **setDescription**(java.lang.String description) throws java.rmi.RemoteException
public double **getPrice**() throws java.rmi.RemoteException
public void **setPrice**(double price) throws java.rmi.RemoteException
public double **getSale**() throws java.rmi.RemoteException
public void **setSale**(double sale) throws java.rmi.RemoteException

Beans.product

Class ProductEJB

```
java.lang.Object
  |
  +--Beans.product.ProductEJB
```

public class ProductEJB
extends java.lang.Object
implements javax.ejb.EntityBean
Implements the business methods and creation methods of the Product EJB

Field Detail

public java.lang.String **productId**
public java.lang.String **father**
public java.lang.String **name**
public java.lang.String **type**
public java.lang.String **salecode**
public java.lang.String **validstart**
public java.lang.String **validend**
public java.lang.String **salestart**
public java.lang.String **saleend**
public java.lang.String **description**
public double **price**
public double **sale**

Constructor Detail

ProductEJB
public **ProductEJB**()

Method Detail

ejbCreate
public java.lang.String **ejbCreate**(java.lang.String productId, java.lang.String father, java.lang.String name, java.lang.String type, java.lang.String salecode, java.lang.String validstart, java.lang.String validend, java.lang.String salestart, java.lang.String saleend, java.lang.String description, double price, double sale)
throws javax.ejb.CreateException
When ejbCreate is called, the EJB container creates a ProductEJB. The creation method of a EJB must be named ejbCreate() and can not return a value.
Parameters:
The - parameters are the instance variables of the Product EJB

getId
public java.lang.String **getId**()
Returns the Id of the Product

getFather
public java.lang.String **getFather**()
Returns the father of the Product

getName
public java.lang.String **getName**()
Returns the name of the Product

getType
public java.lang.String **getType**()

Returns the type of the Product

getSalecode
public java.lang.String **getSalecode**()
Returns the sale code of the Product

getValidstart
public java.lang.String **getValidstart**()
Returns the start of validation period of the Product

getValidend
public java.lang.String **getValidend**()
Returns the end of validation period of the Product

getSalestart
public java.lang.String **getSalestart**()
Returns the start of sale period of the Product

getSaleend
public java.lang.String **getSaleend**()
Returns the end of sale period of the Product

getDescription
public java.lang.String **getDescription**()
Returns the description of the Product

getPrice
public double **getPrice**()
Returns the price of the Product

getSale
public double **getSale**()
Returns the sale of the Product

setId
public void **setId**(java.lang.String id)
Sets a new Id to the Product
Parameters:
id - The new Id of the Product

setFather
public void **setFather**(java.lang.String father)
Sets a new father to the Product
Parameters:
father - The new father of the Product

setName
public void **setName**(java.lang.String name)
Sets a new name to the Product
Parameters:
name - The new name of the Product

setType
public void **setType**(java.lang.String type)
Sets a new type to the Product
Parameters:
type - The new type of the Product

setSalecode
public void **setSalecode**(java.lang.String salecode)

Sets a new salecode to the Product
Parameters:
salecode - The new salecode of the Product

setValidstart
public void **setValidstart**(java.lang.String validstart)
Sets a new start of validation period to the Product
Parameters:
validStart - The new start of validation period of the Product

setValidend
public void **setValidend**(java.lang.String validend)
Sets a new end of validation period to the Product
Parameters:
validend - The new end of validation period of the Product

setSalestart
public void **setSalestart**(java.lang.String salestart)
Sets a new start of sale period to the Product
Parameters:
salestart - The new start of sale period of the Product

setSaleend
public void **setSaleend**(java.lang.String saleend)
Sets a new end of sale period to the Product
Parameters:
saleend - The new end of sale period of the Product

setDescription
public void **setDescription**(java.lang.String description)
Sets a new description to the Product
Parameters:
description - The new description of the Product

setPrice
public void **setPrice**(double price)
Sets a new price to the Product
Parameters:
price - The new price of the Product

setSale
public void **setSale**(double sale)
Sets a new sale to the Product
Parameters:
sale - The new sale of the Product

The following methods msut be overriden beacause ProductEJB implements EntityEJB:

public void **setEntityContext**(javax.ejb.EntityContext context)
public void **ejbActivate**()
public void **ejbPassivate**()
public void **ejbRemove**()
public void **ejbLoad**()
public void **ejbStore**()
public void **unsetEntityContext**()
public void **ejbPostCreate**(java.lang.String productId, java.lang.String father, java.lang.String name, java.lang.String type, java.lang.String salecode, java.lang.String validstart, java.lang.String validend, java.lang.String salestart, java.lang.String saleend, java.lang.String description, double price, double sale)

Beans.product

Interface ProductHome

public interface ProductHome
extends javax.ejb.EJBHome
Defines the creation and finder methods of the Product EJB

Method Detail

create
public Product **create**(java.lang.String productId, java.lang.String father, java.lang.String name, java.lang.String type, java.lang.String salecode, java.lang.String validstart, java.lang.String validend, java.lang.String salestart, java.lang.String saleend, java.lang.String description, double price, double sale) throws java.rmi.RemoteException, javax.ejb.CreateException
The create method is implemented in the ProductEJB as ejbCreate()

findByPrimaryKey
public Product **findByPrimaryKey**(java.lang.String productId) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByFather
public java.util.Collection **findByFather**(java.lang.String father) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByName
public java.util.Collection **findByName**(java.lang.String name) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByType
public java.util.Collection **findByType**(java.lang.String type) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findBySaleCode
public java.util.Collection **findBySaleCode**(java.lang.String saleCode) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByDescription
public java.util.Collection **findByDescription**(java.lang.String description) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findInPriceRange
public java.util.Collection **findInPriceRange**(double lowprice, double highprice) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

Beans.profile

Interface Profile

public interface Profile
extends javax.ejb.EJBObject
Defines the business methods of the Profile EJB

Method Detail

The following business methods are implemented in the ProfileEJB

public int **getId**() throws java.rmi.RemoteException
public void **setId**(int id) throws java.rmi.RemoteException
public java.lang.String **getLogin**() throws java.rmi.RemoteException
public void **setLogin**(java.lang.String login) throws java.rmi.RemoteException
public java.lang.String **getPassword**() throws java.rmi.RemoteException
public void **setPassword**(java.lang.String password) throws java.rmi.RemoteException
public java.lang.String **getFirstName**() throws java.rmi.RemoteException
public void **setFirstName**(java.lang.String firstName) throws java.rmi.RemoteException
public java.lang.String **getLastName**() throws java.rmi.RemoteException
public void **setLastName**(java.lang.String lastName) throws java.rmi.RemoteException
public java.lang.String **getStatus**() throws java.rmi.RemoteException
public void **setStatus**(java.lang.String status) throws java.rmi.RemoteException
public java.lang.String **getProfession**() throws java.rmi.RemoteException
public void **setProfession**(java.lang.String profession) throws java.rmi.RemoteException
public java.lang.String **getSex**() throws java.rmi.RemoteException
public void **setSex**(java.lang.String sex) throws java.rmi.RemoteException
public java.lang.String **getMaritalStatus**() throws java.rmi.RemoteException
public void **setMaritalStatus**(java.lang.String maritalStatus) throws java.rmi.RemoteException
public java.lang.String **getAdressStreet**() throws java.rmi.RemoteException
public void **setAdressStreet**(java.lang.String adressStreet) throws java.rmi.RemoteException
public java.lang.String **getAdressCity**() throws java.rmi.RemoteException
public void **setAdressCity**(java.lang.String adressCity) throws java.rmi.RemoteException
public java.lang.String **getAdressCityCode**() throws java.rmi.RemoteException
public void **setAdressCityCode**(java.lang.String adressCityCode) throws java.rmi.RemoteException
public java.lang.String **getAdressCountry**() throws java.rmi.RemoteException
public void **setAdressCountry**(java.lang.String adressCountry) throws java.rmi.RemoteException
public java.lang.String **getTelephone**() throws java.rmi.RemoteException
public void **setTelephone**(java.lang.String telephone) throws java.rmi.RemoteException
public java.lang.String **getEmail**() throws java.rmi.RemoteException
public void **setEmail**(java.lang.String email) throws java.rmi.RemoteException
public java.lang.String **getLanguage**() throws java.rmi.RemoteException
public void **setLanguage**(java.lang.String language) throws java.rmi.RemoteException
public int **getAge**() throws java.rmi.RemoteException
public void **setAge**(int age) throws java.rmi.RemoteException
public java.lang.String[] **getIntrests**() throws java.rmi.RemoteException
public void **setIntrests**(java.lang.String[] intrests) throws java.rmi.RemoteException
public int **getTotalPurchase**() throws java.rmi.RemoteException
public void **setTotalPurchase**(int totalPurchase) throws java.rmi.RemoteException
public java.lang.String **getLatestLogin**() throws java.rmi.RemoteException
public void **setLatestLogin**(java.lang.String latestLogin) throws java.rmi.RemoteException
public java.lang.String[] **getPurchase**() throws java.rmi.RemoteException
public void **setPurchase**(java.lang.String[] purchase) throws java.rmi.RemoteException
public java.lang.String **getFavoritColor**() throws java.rmi.RemoteException
public void **setFavoritColor**(java.lang.String favoritColor) throws java.rmi.RemoteException

Beans.profile

Class ProfileEJB

```
java.lang.Object
  |
  +--Beans.profile.ProfileEJB
```

public class ProfileEJB

extends java.lang.Object
implements javax.ejb.EntityBean
Implements the business methods and creation methods of the Profile EJB

Field Detail

public java.lang.Integer **Id**
public java.lang.String **Login**
public java.lang.String **Password**
public java.lang.String **FirstName**
public java.lang.String **LastName**
public java.lang.String **Status**
public java.lang.String **Profession**
public java.lang.String **Sex**
public java.lang.String **MaritalStatus**
public java.lang.String **AdressStreet**
public java.lang.String **AdressCity**
public java.lang.String **AdressCityCode**
public java.lang.String **AdressCountry**
public java.lang.String **Telephone**
public java.lang.String **Email**
public java.lang.String **Language**
public int **Age**
public java.lang.String **Intrest01**
public java.lang.String **Intrest02**
public java.lang.String **Intrest03**
public java.lang.String **Intrest04**
public java.lang.String **Intrest05**
public java.lang.String **Intrest06**
public java.lang.String **Purchase01**
public java.lang.String **Purchase02**
public java.lang.String **Purchase03**
public java.lang.String **Purchase04**
public java.lang.String **Purchase05**
public java.lang.String **Purchase06**
public java.lang.String **Purchase07**
public java.lang.String **Purchase08**
public java.lang.String **Purchase09**
public java.lang.String **Purchase10**
public int **TotalPurchase**
public java.lang.String **LatestLogin**
public java.lang.String **FavoritColor**

Constructor Detail

ProfileEJB
public **ProfileEJB**()

Method Detail

ejbCreate
public void **ejbCreate**(int id, java.lang.String login, java.lang.String password, java.lang.String firstname, java.lang.String lastname, java.lang.String status, java.lang.String profession, java.lang.String sex, java.lang.String maritalStatus, java.lang.String adressStreet, java.lang.String adressCity, java.lang.String adressCityCode, java.lang.String adressCountry, java.lang.String telephone, java.lang.String email, java.lang.String language, int age, java.lang.String intrest01, java.lang.String intrest02, java.lang.String intrest03, java.lang.String intrest04, java.lang.String intrest05, java.lang.String intrest06, int totalPurchase, java.lang.String latestLogin, java.lang.String favoritColor) throws javax.ejb.CreateException

When ejbCreate is called, the EJB container creates a ProfileEJB. The creation method of a EJB must be named ejbCreate() and can not return a value.
Parameters:
The parameters are the instance variables of the Profile EJB

getId
public int **getId**() throws java.rmi.RemoteException
Returns the Id of the Profile

setId
public void **setId**(int id) throws java.rmi.RemoteException
Sets a new id to the Product
Parameters:
id - The new id of the Product

getLogin
public java.lang.String **getLogin**() throws java.rmi.RemoteException
Returns the login name of the Profile

setLogin
public void **setLogin**(java.lang.String login) throws java.rmi.RemoteException
Sets a new login name to the Product
Parameters:
login - The new login name of the Product

getPassword
public java.lang.String **getPassword**() throws java.rmi.RemoteException
Returns the password of the Profile

setPassword
public void **setPassword**(java.lang.String password) throws java.rmi.RemoteException
Sets a new password to the Product
Parameters:
password - The new password of the Product

getFirstName
public java.lang.String **getFirstName**() throws java.rmi.RemoteException
Returns the first name of the Profile

setFirstName
public void **setFirstName**(java.lang.String firstName) throws java.rmi.RemoteException
Sets a new first name to the Product
Parameters:
firstName - The new first name of the Product

getLastName
public java.lang.String **getLastName**() throws java.rmi.RemoteException
Returns the last name of the Profile

setLastName
public void **setLastName**(java.lang.String v) throws java.rmi.RemoteException
Sets a new last name to the Product
Parameters:
lastName - The new last name of the Product

getStatus
public java.lang.String **getStatus**() throws java.rmi.RemoteException
Returns the status of the Profile

setStatus
public void **setStatus**(java.lang.String status) throws java.rmi.RemoteException
Sets a new status to the Product
Parameters:
status - The new status of the Product

getProfession
public java.lang.String **getProfession**() throws java.rmi.RemoteException
Returns the profession of the Profile

setProfession
public void **setProfession**(java.lang.String profession) throws java.rmi.RemoteException
Sets a new profession to the Product
Parameters:
profession - The new profession of the Product

getSex
public java.lang.String **getSex**() throws java.rmi.RemoteException
Returns the sex of the Profile

setSex
public void **setSex**(java.lang.String sex) throws java.rmi.RemoteException
Sets a new sex to the Product
Parameters:
sex - The new sex of the Product

getMaritalStatus
public java.lang.String **getMaritalStatus**() throws java.rmi.RemoteException
Returns the marital status of the Profile

setMaritalStatus
public void **setMaritalStatus**(java.lang.String maritalStatus) throws java.rmi.RemoteException
Sets a new marital status to the Product
Parameters:
maritalStatus - The new marital status of the Product

getAdressStreet
public java.lang.String **getAdressStreet**() throws java.rmi.RemoteException
Returns the street adress of the Profile

setAdressStreet
public void **setAdressStreet**(java.lang.String adressStreet) throws java.rmi.RemoteException
Sets a new street adress to the Product
Parameters:
adressStreet - The new street adress of the Product

getAdressCity
public java.lang.String **getAdressCity**() throws java.rmi.RemoteException
Returns the city adress of the Profile

setAdressCity
public void **setAdressCity**(java.lang.String adressCity) throws java.rmi.RemoteException
Sets a new city adress to the Product
Parameters:
adressCity - The new city adress of the Product

getAdressCityCode
public java.lang.String **getAdressCityCode**() throws java.rmi.RemoteException
Returns the city code of the Profile

setAdressCityCode
public void **setAdressCityCode**(java.lang.String adressCityCode) throws java.rmi.RemoteException
Sets a new city code to the Product
Parameters:
adressCityCode - The new city code of the Product

getAdressCountry
public java.lang.String **getAdressCountry**() throws java.rmi.RemoteException
Returns the country of the Profile

setAdressCountry
public void **setAdressCountry**(java.lang.String adressCountry) throws java.rmi.RemoteException
Sets a new country to the Product
Parameters:
adressCountry - The new country of the Product

getTelephone
public java.lang.String **getTelephone**() throws java.rmi.RemoteException
Returns the telethone nr of the Profile

setTelephone
public void **setTelephone**(java.lang.String telephone) throws java.rmi.RemoteException
Sets a new telephone to the Product
Parameters:
telephone - The new telephone of the Product

getEmail
public java.lang.String **getEmail**() throws java.rmi.RemoteException
Returns the email of the Profile

setEmail
public void **setEmail**(java.lang.String email) throws java.rmi.RemoteException
Sets a new email to the Product
Parameters:
email - The new email of the Product

getLanguage
public java.lang.String **getLanguage**() throws java.rmi.RemoteException
Returns the language of the Profile

setLanguage
public void **setLanguage**(java.lang.String language) throws java.rmi.RemoteException
Sets a new language to the Product
Parameters:
language - The new language of the Product

getAge
public int **getAge**() throws java.rmi.RemoteException
Returns the age of the Profile

setAge
public void **setAge**(int age) throws java.rmi.RemoteException
Sets a new age to the Product
Parameters:
age - The new age of the Product

getTotalPurchase
public int **getTotalPurchase**() throws java.rmi.RemoteException
Returns the purchase of the Profile

setTotalPurchase
public void **setTotalPurchase**(int totalPurchase) throws java.rmi.RemoteException
Sets a new total purchase to the Product
Parameters:
totalPurchase - The new total purchase of the Product

getLatestLogin
public java.lang.String **getLatestLogin**() throws java.rmi.RemoteException
Returns the last login of the Profile

setLatestLogin
public void **setLatestLogin**(java.lang.String latestLogin) throws java.rmi.RemoteException
Sets a new last login to the Product
Parameters:
latestLogin - The new last login of the Product

getIntrests
public java.lang.String[] **getIntrests**() throws java.rmi.RemoteException
Returns the intrests of the Profile

setIntrests
public void **setIntrests**(java.lang.String[] intrests) throws java.rmi.RemoteException
Sets a new intrests to the Product
Parameters:
intrests - The new intrests of the Product

getPurchase
public java.lang.String[] **getPurchase**() throws java.rmi.RemoteException
Returns the purchase of the Profile

setPurchase
public void **setPurchase**(java.lang.String[] purchase) throws java.rmi.RemoteException
Sets a new purchase to the Product
Parameters:
purchase - The new purchase of the Product

getFavoritColor
public java.lang.String **getFavoritColor**() throws java.rmi.RemoteException
Returns the favorite color of the Profile

setFavoritColor
public void **setFavoritColor**(java.lang.String favoritColor) throws java.rmi.RemoteException
Sets a new favorit color to the Product
Parameters:
favoritColor - The new favorit color of the Product

The following methods must be overridden because Profile EJB implements EntityEJB :

public void **setEntityContext**(javax.ejb.EntityContext context)
public void **ejbRemove**()
public void **ejbActivate**()
public void **ejbPassivate**()
public void **unsetEntityContext**()
public void **ejbLoad**()
public void **ejbStore**()
public void **ejbPostCreate**(int id, java.lang.String login, java.lang.String password, java.lang.String firstname, java.lang.String lastname, java.lang.String status, java.lang.String profession, java.lang.String sex, java.lang.String maritalStatus, java.lang.String adressStreet, java.lang.String adressCity, java.lang.String adressCityCode, java.lang.String adressCountry, java.lang.String telephone, java.lang.String email, java.lang.String language, int age, java.lang.String intrest01, java.lang.String intrest02, java.lang.String

intrest03, java.lang.String intrest04, java.lang.String intrest05, java.lang.String intrest06, int totalPurchase, java.lang.String latestLogin, java.lang.String favoritColor)
.

Beans.profile

Interface ProfileHome

public interface ProfileHome
extends javax.ejb.EJBHome
Defines the creation and finder methods of the Profile EJB

Method Detail

create
public Profile **create**(int id, java.lang.String login, java.lang.String password, java.lang.String firstname, java.lang.String lastname, java.lang.String status, java.lang.String profession, java.lang.String sex, java.lang.String maritalStatus, java.lang.String adressStreet, java.lang.String adressCity, java.lang.String adressCityCode, java.lang.String adressCountry, java.lang.String telephone, java.lang.String email, java.lang.String language, int age, java.lang.String intrest01, java.lang.String intrest02, java.lang.String intrest03, java.lang.String intrest04, java.lang.String intrest05, java.lang.String intrest06, int totalPurchase, java.lang.String latestLogin, java.lang.String favoritColor) throws java.rmi.RemoteException, javax.ejb.CreateException
The create method is implemented in the ProfileEJB as ejbCreate()

findByPrimaryKey
public Profile **findByPrimaryKey**(java.lang.Integer profileId) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByLogin
public java.util.Collection **findByLogin**(java.lang.String login, java.lang.String password) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByName
public java.util.Collection **findByName**(java.lang.String firstName, java.lang.String lastName) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

findByEmail
public java.util.Collection **findByEmail**(java.lang.String email) throws javax.ejb.FinderException, java.rmi.RemoteException
This finder-method is implemented by the EJB container in the deployment phase

Beans.session

Interface Session

public interface Session
extends javax.ejb.EJBObject
Defines the business methods of the Session EJB

Method Detail

The following business methods are implemented in the SessionEJB :

public ProfileHome **getProfileHome**() throws java.rmi.RemoteException
public ProductHome **getProductHome**() throws java.rmi.RemoteException
public CartHome **getCartHome**() throws java.rmi.RemoteException
public java.util.Locale **getLocale**() throws java.rmi.RemoteException
public void **setLocale**(java.lang.String language, java.lang.String country) throws java.rmi.RemoteException
public java.lang.String **getType**() throws java.rmi.RemoteException
public void **setType**(java.lang.String type) throws java.rmi.RemoteException

Beans.session

Class SessionEJB

```
java.lang.Object
  |
  +--Beans.session.SessionEJB
```

public class SessionEJB
extends java.lang.Object
implements javax.ejb.SessionBean
Implements the business methods and creation methods of the Session EJB

Constructor Detail

SessionEJB
public **SessionEJB**()

Method Detail

ejbCreate
public void **ejbCreate**() throws javax.ejb.CreateException, java.rmi.RemoteException
When ejbCreate is called, the EJB container creates a SessionEJB. The creation method of a EJB must be named ejbCreate() and can not return a value.

getProfileHome
public ProfileHome **getProfileHome**() throws java.rmi.RemoteException
This method returns the Home interface of the Profile EJB that is stored in the Session EJB

getProductHome
public ProductHome **getProductHome**() throws java.rmi.RemoteException
This method returns the Home interface of the Product EJB that is stored in the Session EJB

getCartHome
public CartHome **getCartHome**() throws java.rmi.RemoteException
This method returns the Home interface of the Cart EJB that is stored in the Session EJB

getLocale
public java.util.Locale **getLocale**() throws java.rmi.RemoteException
This method returns the locale of the user that is stored in the Session EJB

setLocale
public void **setLocale**(java.lang.String language, java.lang.String country) throws java.rmi.RemoteException
This method sets the locale of the user
Parameters:
language - the language of the user
country - the country of the user

getType
public java.lang.String **getType**() throws java.rmi.RemoteException

This method returns the type of the user that is stored in the Session EJB

setType
public void **setType**(java.lang.String type) throws java.rmi.RemoteException
This method sets the type of the user
Parameters:
type - the type of the user

The following methods must be overridden because SessionEJB implements SessionBean:

public void **ejbRemove**() throws java.rmi.RemoteException
public void **ejbActivate**() throws java.rmi.RemoteException
public void **ejbPassivate**() throws java.rmi.RemoteException
public void **setSessionContext**(javax.ejb.SessionContext sc) throws java.rmi.RemoteException
public void **unsetSessionContext**() throws java.rmi.RemoteException
public void **ejbLoad**() throws java.rmi.RemoteException
public void **ejbStore**() throws java.rmi.RemoteException

Beans.session

Interface SessionHome

public interface SessionHome
extends javax.ejb.EJBHome
Defines the creation method of the Session EJB

Method Detail

create
public Session **create**() throws java.rmi.RemoteException, javax.ejb.CreateException
The create method is implemented in the SessionEJB as ejbCreate()