17

Case Study: J2EE, EJBs, and Tag Libraries

The JavaServer Pages technology is a part of a bigger platform: the **Java2 Enterprise Edition** (J2EE). In this chapter I will try to give a short introduction to what the Java2 Enterprise Edition is about, and show a simple case study where I design a J2EE application (a simple online bookstore) from scratch. I am a part of the team behind the Orion Application Server, a Java application server supporting the full J2EE standard (http://www.orionserver.com).

Introduction to J2EE

Until recently, companies wishing to develop a serious enterprise system have had to build their own framework to handle complex issues like persistence (often including a lot of SQL code), transactions, security, and making sure the system is scalable.

This is hard work, and it is a natural step to develop servers that handle all this "plumbing" for the application to speed up development and ensure scalability and maximum performance. These servers are called application servers.

The Java 2 Enterprise Edition is a specification that aims to provide a standard for Java application servers. This is a good thing for several reasons:

- No investment in a single proprietary platform.
- Best-of-breed implementation. You can select the implementation that best suits your needs and provides your application with the best scalability and performance.
- Price competition between different vendors ensures good pricing.
- The broad knowledge base that comes with an established standard makes it easier to find good developers, good education, and everything that comes with it.

How about the disadvantages, like bad compatibility? Some people might argue that an open specification and different implementations inevitably leads to bad compatibility problems, and this has been the case with several earlier specifications. To avoid this, many measures have been taken: whether this will be enough remains to be seen. The most important things that could make the difference are:

- A Compatibility Test Suite (CTS). All J2EE compliant application servers have to pass a number of tests to guarantee that they indeed work as the specification mandates.
- A Reference Implementation, the J2EE RI, is available, with sources, to vendors that are in the process of building J2EE compliant application servers.
- Binary compatibility for applications. Binary formats like WAR and EAR files can be directly moved between application servers, ensuring that not even recompilation is required when moving an application from one server to another.

Overview

The Java 2 Enterprise Edition is a specification consisting of many parts. J2EE is made up of a lot of different technologies, and a specification for how these work together. The main parts of the J2EE runtime environment are:

- **Application components**. There are four mandatory types of application components that a J2EE compliant product must support:
  - Application clients. These are fat clients written as normal Java applications. They normally access the application server using Remote Method Invocation (RMI).
  - Applets. These are graphical clients that normally execute within a web browser.
  - Servlets and JavaServer Pages.
  - Enterprise JavaBeans. These are components that execute in a container in the application server that manages them and handles much of the "plumbing" for them. These are more thoroughly discussed in a later section.
- **Containers**. Every kind of component lives inside a container that provides the component with runtime services. There is one type of container for every type of component. The most interesting containers for us are the web container that the servlets and JavaServer Pages live in, and the EJB container that manages the Enterprise Java Beans.

- **Resource manager drivers**. A resource manager driver is a driver that provides some kind of connectivity to an external component. The only type that is supported today is a driver that implements any of the J2EE standard services, like Java Messaging Service (JMS), Java DataBase Connectivity (JDBC), or JavaMail. In future versions of the specification something called connectors will be supported, and they will provide support for easily making drivers for connecting to external applications, for example Enterprise Resource Planning (ERP) systems.
- **Database**. Databases are accessible via JDBC, and need to be accessible from all component types except for applets.

The J2EE platform also includes some standard services that must be provided by a J2EE compliant product. They include the following mandatory services:

- HTTP. The J2EE server must make its services available via the HTTP protocol through support for servlets and JSP pages.
- HTTPS. A J2EE server must also be able to layer HTTP over SSL to make connections secure.
- Java Transaction API (JTA). A J2EE server must implement transaction support.
- JavaIDL. J2EE clients are required to be able to connect to CORBA objects using the JavaIDL ORB.
- JDBC
- Java Naming and Directory Interface (JNDI). Naming and directory services are used server wide and have to be present.
- JavaMail and JavaBeans Activation Framework. The J2EE platform includes services for sending email over the Internet. To support this the JavaBeans Activation Framework is also needed.

The following services, however, are optional:

- **RMI-IIOP**. RMI-IIOP support is not yet required by a J2EE implementation, but it will become so in future versions. However, a J2EE application is already required to use the RMI-IIOP APIs when accessing Enterprise Java Beans via RMI.
- **Java Message Service** (JMS) allows enterprise messaging. A J2EE server does not yet have to implement this service, but it will be required in later versions.

Web applications

The servlet 2.2 specification adds the concept of web applications, which are collections of web resources, such as servlets, JSPs, HTML pages, etc. A web application is rooted at a specific path in a web server, and can be conveniently distributed as a Java Archive (JAR) file with a `.war` filename extension. Later in this chapter we will build a web application for the webshop example.

Enterprise applications

The J2EE specification introduces the concept of Enterprise applications. An Enterprise application contains J2EE modules, which could be web applications, EJBs, application clients (normal Java applications talking to a J2EE server), and applets. A J2EE Enterprise archive is packaged as a Java Archive file with a `.ear` filename extension. For a more complete explanation of enterprise applications, read chapter 8 of the J2EE specification, available at http://java.sun.com/j2ee/docs.html.

Sun J2EE Blueprints

The Sun J2EE Blueprints, available online at http://java.sun.com/j2ee/blueprints/, describe how to build J2EE applications. They describe best practices, and contain a real example of a full J2EE application, the Java Pet Store.

Platform roles

One of the important concepts in J2EE is platform roles. The J2EE standard provides separation between different roles, so that different players in the market can specialize on their role. Increased specialization will hopefully increase productivity as individuals and organizations focus on building competence in one area.

The roles that are discussed in the J2EE specification are:

- J2EE Product Provider. This is the application server vendor.
- Application Component Provider. Companies like The Theory Center specialize in building components that take care of certain common tasks: these could be components for e-commerce, or for some other common application.
- Application Assembler. The application assembler uses a set of components and builds an application from them, which is then delivered as an Enterprise Archive (`.ear` file).
- Deployer. The deployer is the one who deploys the applications into production environments. This usually means that the deployer installs, configures, and executes the application.
- System Administrator. The system administrator monitors the application's well-being and administers the application after it has been deployed.
- Tool Provider. Finally, a tool provider makes tools for development and packaging of the application components.

Obviously, one player can have many of these roles, and that is probably the situation we will see until the market has fully matured. We see today that certain companies span across many roles, like BEA Systems (http://www.bea.com), which is a J2EE product provider, an application component provider, and a tool provider.

Platform Contracts

The J2EE platform communicates with the outside world in a few different ways. These ways, known as contracts, are well defined, and separated into:

- APIs. The J2EE Application Programming Interfaces defines the interface between the application components and the platform.
- Network protocols. For access by outside applications, specific network protocols have to be implemented by the J2EE product. These are normally HTTP (and HTTPS) and RMI.
- Deployment descriptors. The deployer specifies the behavior and configuration of a J2EE application using deployment descriptors. These are a standard way to specify the configurations of all the application components using XML.

Scalability and Fault tolerance

In a modern environment, a large enterprise application has to be able to survive large numbers of users, high loads, and even server failures without collapsing. The J2EE platform makes sure this is possible, but does not mandate how it is to be done (for example, how to build clusters of J2EE servers): this is left to the J2EE product provider. Normally these implement transparent failure handling and load balancing across a cluster of servers.

Future directions

Of course, development doesn't stop with the current specification of the J2EE platform. All the covered technologies will develop over time and more problems will be solved. Examples of things that will make an impact shortly are:

- With CORBA 3, the CORBA Component Model provides a superset of the EJB specification for building server side components. There is also a process of standardizing how fault tolerance is handled in CORBA. This might eventually give us the possibility of deploying applications in a cluster across not only multiple machines, but also multiple application server implementations, giving another level of fault tolerance as software failures in one application server can be masked.
- J2EE 1.3 is currently under development, and is likely to include better JMS support and integration of EJB with JMS. It also introduces the Connectors API for access to legacy systems. It will also contain the new versions of the contained standards, like EJB 2.0, JSP 1.2, and Servlet 2.3. These specifications are currently being developed.
- The platform roles will become clearer. As companies specialize in one platform role we might see a new industry. New business opportunities will appear for people who are ready to take on a niche.

Enterprise Java Beans

Enterprise Java Beans (EJB) is a technology specification from Sun, which standardizes how to build server side components for distributed applications.

Overview of EJB

To gain a better understanding of what EJB is all about, let us start with breaking the architecture apart and look at the different parts.

The Components

The components, the actual beans, are large-grained objects representing either some functionality (like a shopping cart) or some data (like a customer bean that holds information about a customer). There is a big difference between these kinds of components, and because of this EJBs exist in two basic types: Session beans, and Entity beans. The main differences are outlined in this table:

Type of bean	Purpose	Client access	Persistence	Transactions
Session bean	Perform work	One client	Normally transient	Usually initiates transactions
Entity bean	Represent persistent object	May have many clients	Persistent	Can take part in transactions

Session Beans

A Session bean is a collection of functionality. It can be seen as an extension to the client, since a session bean is connected to a specific client and when the client disconnects the session bean is destroyed. In other words, a session bean is transient. In the normal case, a session bean does all the work and talks to entity beans for access to persistent data.

Entity Beans

An entity bean represents a persistent object. Often, the content of the bean is stored in a relational database, but the contents could also be stored directly to disk, in an object database, or some other storage. The persistence of the entity beans can either be declarative (the administrator simply defines in what data source and table the bean data is stored) or manual (the component developer explicitly writes code that handles the persistence). These two modes are called **Container Managed Persistence** (CMP) and **Bean Managed Persistence** (BMP).

The advantage of Container Managed Persistence is simplicity, separation, and room for optimization by the application server. Since you don't write any code for the persistence, a layer of complexity is removed. This also means that you don't tie yourself to a certain data store, and it means that the application server can optimize access to the data source since it controls all access to it. Bean Managed Persistence has the advantage of giving the bean developer full control to store data in more customizable or advanced ways. Contrary to what some people believe, CMP usually is much faster than BMP, due to the optimizations the application server can do to the more abstract description. With a BMP bean the bean developer has written actual code for persistence, so there is no room for the application server to use any smart tricks. So, for many reasons, CMP is what should be used in most cases. Only use BMP when you need absolutely full control of the persistence.

To summarize,

Use CMP if you need:	Use BMP if you need:
Performance	Full control
Simplicity	
Database portability	

EJB container

Enterprise Java Beans run within a container, which controls the beans and provides them with system services. The services a container offers are:

- Transactions. The EJB specification provides transactional semantics for components, allowing the component developer to specify transactional properties of components.
- Security
- Remote access (RMI)
- Life cycle management
- Database connection pooling. To provide efficient access to databases it is better to make clients share database connections – this means a new database connection doesn't have to be opened for every client request. Instead there is a pool of a number of connections, and when a client wants a database connection it gets an already opened connection to use so there is no start-up time. When the client doesn't need the connection any more, it's returned to the pool so that it can be reused. The EJB server handles this pooling.

Limitations when writing EJBs

When writing portable components in which resources are handled by an application server, certain things have to be done to make sure that the application server can really take control of resource management. If the EJBs try to access certain resources themselves, it will be impossible for the server to optimize their use. Because of, this some restrictions apply to writing EJBs, and you cannot do any of the following things:

- Manage threads
- Access files directly
- Use graphics
- Listen to a socket or use multicast sockets
- Load a native library
- Change the `SocketFactory` for `ServerSocket` and `Socket`
- Change the `StreamHandlerFactory` for the `URL` class

Stateful and Stateless Operation

There are two kinds of Session EJBs: stateful and stateless. A stateful Session bean is a bean that contains state for the client of the Session bean. Keeping state in a Session bean can be very usable and simple, for example when implementing the shopping cart for the webshop we are building. What does this mean for scalability, though? There needs to be one instance of the stateful session bean for each client, and every instance will contain some state. This could mean that memory usage gets high when there are many concurrent users, and it could also make clustering more difficult since state would have to be replicated to the cluster if a client is to access the bean through another server.

JMS

The Java Message Service (JMS) is a standardized interface that enables Java programs to create, send, receive, and read messages using any enterprise messaging system that supports JMS. Just as JDBC provides an abstraction for database access so that your code will not be database dependent, JMS gives you the possibility to write code that will not be dependent on the underlying messaging system, be it IBM MQSeries, Java Message Queue, BEA Weblogic, Orion, or any JMS compliant messaging system.

JMS Capabilites

JMS supports both **publish/subscribe messaging** and **point-to-point messaging**.

- In publish-subscribe messaging, clients publish messages to, and subscribe to messages from, so-called topics. A topic can be seen as a chat room for messaging clients: everyone can join, and will then receive all messages sent to the topic. It can also send messages with that topic and everyone that is subscribed will receive the message. The topic is fully dynamic and automatically adjusts as publishers and subscribers enter and leave.
- In point-to-point messaging, a client sends a message directly to another client.

The JMS API definitions must be included in a J2EE compliant application server, but the server is not required to actually implement the JMS services. A future version of the J2EE specification will, however, require J2EE-compliant application servers to provide support for JMS messaging, and make those services available to the application developer.

Servlets/JSP

Servlets and JavaServer pages are an important part of J2EE, and are described extensively in this book.

The Java Transaction API and the Java Transaction Service (JTA/JTS)

Transactions enable developers to guarantee that a batch of operations execute atomically. The classic example is that when doing a bank transfer between two accounts the withdrawal from one account and the deposit to another account must *both* succeed, or the changes should be undone to both accounts. Transactions can also be used to guarantee consistency, isolation (if two customers are doing transfers between the same accounts at the same time they must not interfere with one another), and durability (the data is not lost if a failure occurs). A transaction that fulfils these requirements is called an ACID transaction, for Atomicity, Consistency, Isolation, and Durability.

A **distributed transaction** is a transaction that spans multiple data sources, and must be handled with protocols that make sure that the ACID properties are kept between the different databases.

The J2EE platform gives the developer two main options for handling transaction demarcation. The server can automatically handle the boundaries for a transaction, by committing when a method completes without throwing an exception, or you can specify that transaction demarcation is handled manually. In the latter case, a session bean or a servlet uses the `javax.jta.UserTransaction` interface to initiate and commit (or rollback) a transaction. There is also support for application code to register as an event listener for a transaction, and thus for finding out about the progress of a transaction.

Transactions and Enterprise Java Beans

In EJBs, you can let the transactions be controlled automatically by the server (declaratively), you can handle transactions programmatically in an EJB, or they can be handled in client code. Declarative transactions can be specified at the method-level, which means that each method in an enterprise bean class can perform transactions differently. For even finer-grained transaction control, the developer can control transactions programmatically via the Java Transaction API (JTA), which can also be used for client-controlled transactions. Application code can register as a transaction participant by implementing an optional `Synchronization` interface.

The underlying transaction service in a J2EE product is vendor-specific, and does not affect application code. Not all J2EE product vendors support distributed transactions.

Transaction support for an EJB is set to one of the following:

- `NotSupported`: Never participate in a transaction
- `Supports`: If invoked from inside a transaction, take part in it; If not, do not start a new transaction.
- `RequiresNew`: Every method invocation on the component starts a new transaction
- `Required`: If invoked inside a transaction, take part in it; otherwise start a new transaction.
- `Mandatory`: If invoked with an active transaction, participate in it, if not, throw `javax.transaction.TransactionRequiredException`.
- `Never`: If invoked with an active transaction, throw `java.rmi.RemoteException`.

These conditions are summarized in the following table:

EJB's Transaction attribute	Caller's transaction context	EJB's transaction context
`NotSupported`	None	None
`NotSupported`	Transaction	None
`Supports`	None	None
`Supports`	Transaction	Transaction
`RequiresNew`	None	Transaction
`RequiresNew`	Transaction	New transaction
`Required`	None	Transaction
`Required`	Transaction	Transaction
`Mandatory`	None	Exception
`Mandatory`	Transaction	Transaction
`Never`	None	None
`Never`	Transaction	Exception

JDBC

The Java DataBase Connectivity API gives J2EE applications access through a common interface to datasources for which there exist JDBC drivers. Normally this will mean a relational database, and JDBC drivers are available for all popular relational Database Management Systems.

JNDI

The Java Naming and Directory Interface provides a common interface for accessing naming and directory services. Just like with JDBC, JNDI drivers can be made for different naming and directory services, such as Domain Name Service (DNS), COSNaming, LDAP directory servers, and proprietary naming systems. A J2EE server provides a JNDI tree where developers can bind their EJBs, JDBC datasources, JavaMail sessions, and other values.

JavaMail

JavaMail gives a J2EE application the possibility to send and receive emails.

Like many other J2EE technologies, JavaMail is based on pluggable service providers that do the actual work for a specific mail protocol. This means that many different mail protocols can be supported, and implementations for the most common protocols are available.

XML and XSL

In the current J2EE version (1.2) XML support is not mandated, but it will become mandated in the future. We will not mention XML further at present, but restrict ourselves to a brief note about XSL (Extensible Stylesheet Language). XSL is two things:

- A language for transforming XML documents
- An XML vocabulary for specifying formatting semantics.

The first part is referred to as XSLT (XSL Transformations, http://www.w3.org/TR/xslt). It can be used to transform XML documents into HTML, and will be used in this way later in this chapter. The second part is similar to Cascading Style Sheets (CSS), and is used for providing formatting.

For more info about XSL, see http://www.w3.org/Style/XSL/

Introduction to the Case Study

The Java 2 Enterprise Edition is a very broad platform, and simplifies development of many types of applications. To choose one application as an example for this book can therefore be a matter of finding a familiar application that is well understood and well known. For this reason I chose to develop a very simple web store.

Our example is not a supposed to be a complete webstore, but an example of J2EE technologies working together to build an application. If you want to look at a more complete webstore implementation using J2EE that exemplifies many features in a great way, Sun has made a sample webstore (the Java Pet Store) available at http://java.sun.com/blueprints/.

In the example, I show how different approaches can be used for the system: how to achieve a strict separation between different layers in the webstore, and also a more pragmatic approach. This chapter does not contain all the sources, since they are fairly large. Instead, I will just provide parts of the source code in this chapter but of course you can find complete sources online at http://www.wrox.com, along with the `.ear` file for this project. You can also find the webshop live on http://www.orionserver.com/wroxshop/.

Requirements for the Webstore Application

Our requirements for the webstore application can be summarized as follows:

Interface Requirements

The webstore is to be based on the very common shopping cart model. The requirements are that a customer can add items to his shopping cart, change the number of items of a specific product, remove items, and all the time have a clear view of what is in the cart. He should not have to create an account or log in before selecting products to buy.

Feature Requirements

- Customer management: the webstore must include customer management, but does not have to handle personalization in any way. The customers have to reside in a database.
- Product management: the products are found in a database.

Design Requirements

Separation of presentation, logic and data is a well-known mantra of modern software engineering. It is certainly important to encapsulate certain parts of a system to make it easier to understand and to develop and maintain. The common rules of separation claim that you should separate between presentation, logic, and data. In the design section the solution adopted for this is discussed more thoroughly. The application should:

- Exemplify these models and explain the good and bad things with the models (to be explained further):
- Show both XML and HTML JSPs.
- Show both direct access to entity beans, and going via a session bean for all EJB access.
- Show how to use both custom tag extension libraries and third party tag extension libraries.

Limitations

This example is not supposed to demonstrate how to build and deploy a real life production webstore. The purpose is to show how to use different J2EE technologies together. For that reason, many issues are not handled in this example, including:

- Security. No regard is taken to security in this simple example. Deploying the webstore using the HTTPS protocol is very simple and basically an administration issue that is server-dependent.
- Order processing.
- Payment. Obviously, in this example no real purchases are made and none of the issues with payment are handled.
- Product categories. Normally we would like to present a hierarchical view of products with different product categories. In this example we skip that part but we keep this in mind when we design things and discuss how it can be easily added.

Webstore Design

So, with the requirements in place, we can start to design the application.

Basic Architectural Design

One of the requirements was that we separate different parts of the implementation from each other. The MVC model was discussed, and this is the fundamental division for a normal computer program, but it is a pretty non-granular division and for a specific system there are usually aspects that should be separated from each other. For the webstore, I have chosen to divide the application into four different main layers, some of which contain sub-layers. Normally only three layers are described, but I chose to introduce an information structure layer which, in my opinion, does not quite fall into the presentation layer. This information structure layer contains information about the structure and contents of the presented information. Later, I will start by designing this layer.

Layers

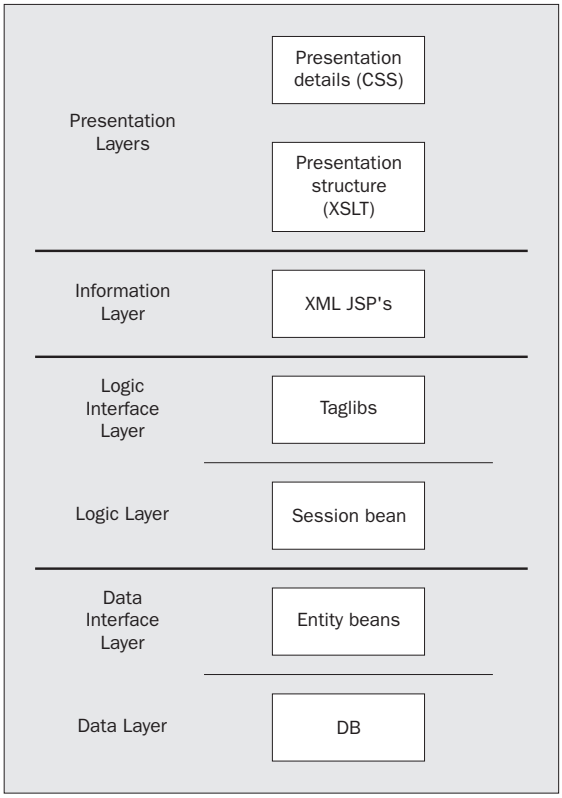
The layers of our application are then:

- Presentation layer, consisting of the presentation details (colors, fonts, etc.) using CSS, and the presentation structure, using XSLT.
- Information structure layer, using JSPs to generate XML.
- Logic layer, using tag extension libraries to interface the logic to the information structure layer, and session beans.
- Data layer, comprising entity beans (data interface) and the database.

It might seem like overkill to use XML/XSL instead of plain HTML with this structure, since the use of tag extension libraries will almost fully separate the logic from the HTML pages. For this reason, I chose to demonstrate both techniques: some JSPs will use XML and some pages will use HTML. This example is not a recommendation on how to make the best web store and, in many cases, if you use JSP 1.1 and tag extension libraries, you can get nice separation without XML. The reason for using XML here is to demonstrate how it can be done and, even though I've mostly spoken about separation, there are some other advantages of using XML that might be applicable in some cases. Some of these are:

- XML will be (and already is in some cases) handled directly by clients in the future. This means that the XSL can live client-side and the presentation can be changed without a round trip to the server.
- XSLT provides advanced transformation rules that can be performed without writing any code. Standard tag extension libraries will later make this possible in pure HTML JSPs.

The structure now looks like this:



Designing the JSPs

To start with, we design the information structure of the webstore. I chose to start at this end because this is something that is easy to imagine without knowing the other parts of the system. If we start with the actual structure of our site, we will get to know what will be required for the other parts. In this section, all the JSPs will be designed in XML. However, because we are aiming to show different ways of doing things, not all JSPs will be implemented in XML. Hopefully you will see the advantages and disadvantages of different approaches.

The Welcome Page

The first page a user will see is what we call the welcome page. In this case, we will let the welcome page simply contain a product browsing facility, and a control bar where you can see what is in your shopping cart, and also log in, proceed to checkout, etc. We choose to separate this into two HTML frames, one for the product browsing, and one for the control bar.

The Product Browsing Page

The top frame will contain the page from which you browse and buy the actual products. The structure of this is simple; the page will basically consist of:

- A title and welcome text
- A structure containing the items in the store

If we model this in XML, we could get something like:

```
<title>
  Welcome to the Professional JavaServer Pages sample store
</title>
<items>
  <item name="foo" price="$200" image="foo.gif"/>
</items>
```

The welcome text is part of the information layer, and can therefore be coded into the actual JSP. The items, however, should exist in a database and be retrieved from it in real time, so that up to date product information is always presented to the user. When we need access to data in a database, we use an Entity bean as the interface to this data. Exactly how this is done is decided later in the implementation section, but we make a note now that we will need an entity EJB for the products. We will need to know this when we design our beans in the next section.

The Toolbar/Cart Page

The bottom frame of the welcome page will contain the toolbar/cart page. From now on I will just call it the cart page. The structure of this page should be very simple. We basically want to present the contents of the cart (and be able to modify it), and be able to proceed to checkout and log in or out. For this, we need the cart contents and the user needs to be logged in. We model this as XML and it looks something like:

```
<cart>
  <item name="foo" price="$200" count="2">
  <item name="bar" price="$300" count="1">
</cart>
<user name="baz"/>
```

Of course, the items and user will not be hard-coded into the page but extracted from EJBs through the use of tag extensions for cart and user management.

The Checkout Page

The checkout page is the page the customer goes to when he is done browsing the product. This page will look a lot like the cart page: it will present the items in the cart, allow a customer to log in and log out, but it also needs to be able to present the customer information. This means we will have a structure that looks something like this (in XML):

```
<cart>
  <item name="foo" price="$200" count="2">
  <item name="bar" price="$300" count="1">
</cart>
<user name="foo" email="foo@bar.com" address="baz" postalcode="12345"/>
```

As in the cart page, the user information and the cart information will be got using tag extensions.

Pages for functions

Beside our normal viewable pages, we will need a few JSPs that simply do something and don't present anything.

- Login page
- Logout page

Since these do not present anything they will basically just be calls to tag extensions using EJBs, and will therefore not need any information structure design.

Designing the EJBs

We have already concluded that we need a few Enterprise Java Beans for our design. We start by designing our cart bean, which is connected to a certain client for which it holds conversational state. Thus, it is a part of a stateful business process and can be nicely modeled by a stateful session bean. We will keep this Session bean in the JSP Session for easy access.

CartBean

The cart bean should, as we concluded, be a stateful session bean, storing the contents of a customer's shopping cart. A decision we must make is whether to expose our internal storage structure to the outside world. Normally this should be encapsulated and `getProducts()` should just return a collection of items. In this case, however, I chose to return a `HashMap` hashed on the product id and containing the number of that item in the cart, because that is a very simple way to keep track of the items. That means the cart must support these operations:

- `addProduct(String isbn)`
- `HashMap getProducts()`
- `updateProduct(String isbn, int count)`
- `getTotalCost()`
- `login(String name, String password)`
- `logout()`
- `getCustomer()`

ProductBean

The product bean represents a product in a database, and is a typical entity bean. To simplify development we will use Container Managed Persistence. The bean will contain a few attributes:

- ISBN. This will be the primary key (it is a good primary key since it is unique for every product).
- Title (of the product)
- Price
- Author

The product-browsing page needs to be able to get all the products. This can be done using a custom finder, which we call `findAll()`. How this is implemented is discussed in the implementation section.

CustomerBean

The customer bean also represents a persistent entity and is therefore modeled as another CMP entity bean. The attributes for this bean are:

- Name
- Password
- Address
- Postal code
- E-mail

Designing the Tag Extension Libraries

Our design requires two tag extension libraries, for the User library and the Cart.

User Tag Extension Library

The User tag extension library has to be able to let a user log in and out, and needs to allow JSPs to find out who the currently logged in user is. We also want a tag that can be used to find out if there is no user logged in, to be able to present a login form in that case. The only function that needs to take any arguments is `login`, which needs a username and a password. These arguments should not, however, be hard coded in the page, since this will be input from the actual user, so the arguments to the `login` tag should be the parameter names that will contain the username and password. With this design, the tags look something like this:

The login tag:

```
<user:login name="username" password="password"/>
```

(attributes are params, not hardcoded names)

The `logout` tag:

```
<user:logout/>
```

The `getUser` tag:

```
<user:getUser id="user">
  Contents to only exist if someone is logged in.
  The logged in user is made available as the user variable.
</user:getUser>
```

The `noUser` tag:

```
<user:noUser>
  Contents to exist only if no-one is logged in.
</user:noUser>
```

Cart Tag Extension Library

The Cart tag extension library is used to add items to the Cart EJB, and to update the products that have already been put there. For this, one tag is enough – the case of removing a product is covered by updating a product's count to `0`. The attribute `count` is optional. If it is not set, the product count will be set to `1`: this is used for adding a product.

```
<cart:updateProduct product="productname" count="1"/>
```

For the cart page, and for the checkout and order, we also need to be able to iterate through all products in the `Cart` bean, so we need a tag that iterates through the cart:

```
<cart:iterate id="product" countid="count">
  Contents to iterate
</cart:iterate>
```

Finally, we also need to get the total cost of all products in the cart:

```
<cart:totalCost/>
```

That concludes our basic design of our components. This design does not include design of our XSLTs or CSSs, because these parts are not central to the concepts I am trying to demonstrate. The CSS part will be skipped totally for the rest of this chapter.

Implementing Webstore

We can now go through the implementations of our different parts step by step. We have designed all the important parts, so where we start implementing can simply be a choice of what can be tested without implementing the rest. We start implementing the EJBs, since they can function without any of the other components. But before doing this, we need to decide on the package naming. The following packages are used:

- `com.wrox.webshop.ejb.cart`
- `com.wrox.webshop.ejb.customer`
- `com.wrox.webshop.ejb.product`
- `com.wrox.webshop.tags`

Implementing the EJBs

We want to keep all our EJBs in one package, that we will turn into a `.jar` file for easy distribution and for inclusion into a `.ear` Enterprise Archive.

For this we use the following directory structure:

```
META-INF/
  ejb-jar.xml
com/wrox/webshop/ejb/
  cart/
    Classes for the Cart EJB
  customer/
    Classes for the Customer EJB
  product/
    Classes for the Product EJB
```

The file `ejb-jar.xml` is the deployment descriptor for the EJBs, and is discussed later in this section.

Let us implement the EJBs in the same order as we designed them, starting with the cart bean.

CartBean

You need to create 3 files for your bean: an interface for the bean, a home for the bean and the bean itself: `Cart.java`, `CartHome.java` and `CartBean.java`.

`Cart.java` defines the interface for your bean. It is an `interface` and must extend `EJBObject`, and is placed in the `com.wrox.webshop.ejb.cart` package .To be able to use the RMI and `java.util` classes, as well as the EJB and our own packages, we add a few imports. To the top of the file, add the following:

```
package com.wrox.webshop.ejb.cart;

import java.util.*;
import java.rmi.*;
import javax.ejb.*;

import com.wrox.webshop.ejb.customer.*;
```

After these imports we put the definition of the interface. It should look like:

```
public interface Cart extends EJBObject
```

Now we will define the methods of the interface. According to the design, this bean will have seven methods. They must all throw `RemoteException` and be `public`. Some of the methods must be able to throw other exception. A failed login must result in an exception, and the calculation of the total cost could possibly go wrong. I added a generic `CartException` for this case. These definitions should look like:

```
  public void addProduct(String isbn) throws RemoteException;

  public HashMap getProducts() throws RemoteException;

  public void updateProduct(String isbn, int count) throws RemoteException;

  public int getTotalCost() throws CartException, RemoteException;

  public void login(String name, String password)
    throws RemoteException, LoginException;

  public void logout() throws RemoteException;

  public Customer getCustomer() throws RemoteException;
```

Now the definition of the `Cart` interface is done. Now let us create the `CartHome` interface. Again, we start by specifying the package and importing a few classes:

```
package com.wrox.webshop.ejb.cart;

import java.rmi.*;
import javax.ejb.*;
```

The `CartHome` interface must extend the `EJBHome` interface:

```
public interface CartHome extends EJBHome
```

and will only contain one method, `create()`, which must throw `CreateException` and `RemoteException`:

```
    public Cart create() throws CreateException, RemoteException;
```

Now the `CartHome` interface is done, and we can begin to write the code for the actual bean implementation. This is done in the class `CartBean`. As usual, we start with package definition and imports. The class itself must implement `SessionBean`.

```
package com.wrox.webshop.ejb.cart;

import java.rmi.*;
import java.util.*;

import javax.ejb.*;
import javax.naming.*;

import com.wrox.webshop.ejb.product.*;
import com.wrox.webshop.ejb.customer.*;

public class CartBean implements SessionBean
```

The contents of the cart is kept as a `HashMap`. We also need to keep the logged in `Customer` and we reserve a field for the `SessionContext`:

```
    private SessionContext context;
    private HashMap items;
    private Customer customer;
```

The `SessionBean` interface contains a few methods we must implement, which gives the bean access to its context/environment. These are:

- `public void ejbCreate()`
- `public void ejbActivate()`
- `public void ejbPassivate()`
- `public void ejbRemove()`
- `public void setSessionContext(SessionContext context)`

In this bean we don't need to implement these, but just give them empty bodies, except for the `setSessionContext()` method which we simply implement by storing the context in a private variable in case we need it later.

Now we implement the actual functionality of the bean, by implementing the methods we earlier defined to be in the interface `Cart`. These were `addProduct(String isbn)`, `getProducts()`, `updateProduct()`, `getTotalCost()`, `login(String username, String password)`, `logout()`, and `getCustomer()`.

The method `addProduct()` should just add a product to the cart. If the product already exists in the cart, we increase the count by 1:

```
    public void addProduct(String isbn) {
      if (isbn != null && !(isbn.equals("null"))) {
        Integer count = (Integer) items.get(isbn);
        if (count == null) count = new Integer(0);
        items.put(isbn, new Integer(count.intValue()+1));
      }
    }
```

The method `getProducts()` should just return the `Hashtable`:

```
  public Hashtable getProducts() {
    return items;
  }
```

The method `updateProduct(String isbn, int count)` should update the count of a specific product to the count specified; if the count is `0`, it will be removed.

```
    public void updateProduct(String isbn, int count) {
      if (isbn != null) {
        if (count==0) {
          items.remove(isbn);
        } else {
          items.put(isbn, new Integer(count));
        }
      }
    }
```

The method for calculating the total cost looks like this:

```
    public int getTotalCost() throws RemoteException, CartException {
      int totalCost = 0;
      Iterator iterator = items.keySet().iterator();

      try {
        InitialContext context = new InitialContext();
        ProductHome productHome =
          (ProductHome) javax.rmi.PortableRemoteObject
            .narrow(context.lookup("Product"), ProductHome.class);

        while (iterator.hasNext()) {
          try {
            String productID = (String) iterator.next();
            Product product = productHome.findByPrimaryKey(productID);
            totalCost += product.getPrice()
                         * ((Integer) items.get(productID)).intValue();
          } catch (FinderException e) {
            throw new CartException(e.getMessage());
          }
        }
      } catch (NamingException e) {
        throw new CartException(e.getMessage());
      }
      return totalCost;
    }
```

The `login()`, `logout()`, and `getCustomer()` methods are implemented as:

```
  public void login(String username, String password)
          throws LoginException, RemoteException {
    try {
      InitialContext context = new InitialContext();
      CustomerHome customerHome =
        (CustomerHome) javax.rmi.PortableRemoteObject
          .narrow(context.lookup("Customer"), CustomerHome.class);
      Customer customer = customerHome.findByPrimaryKey(username);
      if (customer.authenticate(password)) {
        this.customer = customer;
      } else {
        throw new LoginException();
      }
    } catch (FinderException e) {
      throw new LoginException();
    } catch (NamingException e) {
      throw new LoginException();
    }
  }

  public void logout() {
    customer = null;
  }

  public Customer getCustomer() {
    return customer;
  }
```

Now the `CartBean` class is all done, and we finish off by defining the two exceptions we use:

```
package com.wrox.webshop.ejb.cart;

public class CartException extends Exception {
  public CartException() {}

  public CartException(String message) {
    super(message);
  }
}
```

```
package com.wrox.webshop.ejb.cart;

public class LoginException extends Exception {
  public LoginException() {}

  public LoginException(String message) {
    super(message);
  }
}
```

Now all classes for the bean are done, and we proceed to the next bean before writing the bean deployment descriptor.

ProductBean

The product bean was designed to be a CMP (Container Managed Persistence) Entity bean with a few attributes:

- isbn, (the primary key)
- Title of the product
- author
- price

It also was designed to implement one custom finder method, `findAll()`, that would return all existing products.

This means the bean should basically just contain these fields and their get-methods, and set-methods for those fields that are not read-only. We make all but the isbn attribute (which we call `ID`) both writeable and readable, giving us a remote interface that looks like this:

```
package com.wrox.webshop.ejb.product;

import java.rmi.*;
import javax.ejb.*;

public interface Product extends EJBObject {
  public String getID() throws RemoteException;
  public String getName() throws RemoteException;
  public String getAuthor() throws RemoteException;
  public double getPrice() throws RemoteException;

  public void setName(String name) throws RemoteException;
  public void setAuthor(String author) throws RemoteException;
  public void setPrice(double price) throws RemoteException;
}
```

The home interface will also be pretty simple. An entity bean normally contains at least a `create()` method and a `findByPrimaryKey()` method: the first gets called when an Entity bean is created using the home, and the `findByPrimaryKey()` method finds an object based on its primary key. We also want to be able to find books based on the title (name). We call this one `findByName(String name)`. Besides these methods, we only need the `findAll()` method:

```
package com.wrox.webshop.ejb.product;

import javax.ejb.*;
import java.rmi.*;
import java.util.*;

public interface ProductHome extends EJBHome {
  public Product create(String id)
          throws CreateException, RemoteException;

  public Product findByPrimaryKey(String key)
          throws RemoteException, FinderException;

  public Collection findAll() throws RemoteException, FinderException;

  public Collection findByName(String name)
          throws RemoteException, FinderException;
}
```

Now it is time to make the actual bean class, which will simply be an implementation of our remote interface with the additional methods that needs to be defined when implementing the interface `javax.ejb.EntityBean`. Note that we do not actually define the `ProductBean` class to implement the `Product` interface in the class declaration; this is possible since the `ProductBean` will never be used directly. Rather, the application server will handle these invocations and generate classes that do implement the interface. For space reasons, the full source is not listed here, but you can find it online.

CustomerBean

I choose to implement the `CustomerBean` as an extension of the `EJBUser` implementation that comes with the Orion Application Server. It can easily be built from scratch without using this `EJBUser` bean if you are using another application server. In the design we concluded that the `CustomerBean` would be another CMP entity bean and that we would need these attributes:

- Name
- Password
- Address
- Postal code
- E-mail

The `EJBUser` bean already contains a name and a password.

As usual we start with the remote interface – we do not show the code again. The home interface will not contain anything special this time, just the `create()` and `findByPrimaryKey()` methods. The primary key is the username, which must be unique. The `create()` method will this time require a username and a password. Again, the source can be found online.

Having created the remote and home interfaces, we make the actual bean class. This one is basically the same as for the `Product` bean, but without the methods that are inherited from the `EJBUserBean`.

Deployment Descriptors

J2EE uses deployment descriptors, which contain meta-information about components. The descriptors are written in XML format, but different component types uses different tags. In the deployment descriptor for EJBs you define such things as

- Bean properties, like name, type (Session or Entity), classes, fields (for entity beans), etc.
- Transaction settings
- Security settings

The deployment descriptor for our EJB jar is called `ejb-jar.xml`, and looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar SYSTEM "ejb-jar.dtd">
<ejb-jar>
  <description>Wrox webshop beans</description>
  <enterprise-beans>
    <entity>
      <description>Product in the webshop</description>
      <ejb-name>Product</ejb-name>
      <home>com.wrox.webshop.ejb.product.ProductHome</home>
      <remote>com.wrox.webshop.ejb.product.Product</remote>
```

```
        <ejb-class>com.wrox.webshop.ejb.product.ProductBean</ejb-class>
        <prim-key-class>java.lang.String</prim-key-class>
        <reentrant>True</reentrant>
        <persistence-type>Container</persistence-type>
        <cmp-field><field-name>id</field-name></cmp-field>
        <cmp-field><field-name>name</field-name></cmp-field>
        <cmp-field><field-name>author</field-name></cmp-field>
        <cmp-field><field-name>price</field-name></cmp-field>
        <primkey-field>id</primkey-field>
      </entity>
      <entity>
        <display-name>com.wrox.webshop.ejb.customer.Customer</display-name>
        <description>Customer in the webshop</description>
        <ejb-name>Customer</ejb-name>
        <home>com.wrox.webshop.ejb.customer.CustomerHome</home>
        <remote>com.wrox.webshop.ejb.customer.Customer</remote>
        <ejb-class>com.wrox.webshop.ejb.customer.CustomerBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <primkey-class>java.lang.String</primkey-class>
        <reentrant>False</reentrant>
        <cmp-field>
          <field-name>username</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>password</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>locale</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>postalCode</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>address</field-name>
        </cmp-field>
        <cmp-field>
          <field-name>email</field-name>
        </cmp-field>
        <primkey-field>username</primkey-field>
      </entity>

      <session>
        <description>Webshop cart</description>
        <ejb-name>Cart</ejb-name>
        <home>com.wrox.webshop.ejb.cart.CartHome</home>
        <remote>com.wrox.webshop.ejb.cart.Cart</remote>
        <ejb-class>com.wrox.webshop.ejb.cart.CartBean</ejb-class>
        <session-type>Stateful</session-type>
        <transaction-type>Container</transaction-type>
      </session>
    </enterprise-beans>
    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>Customer</ejb-name>
          <method-name>*</method-name>
        </method>
        <method>
          <ejb-name>Product</ejb-name>
          <method-name>*</method-name>
        </method>
```

```
        <method>
          <ejb-name>Cart</ejb-name>
          <method-name>*</method-name>
        </method>
        <trans-attribute>NotSupported</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
  </ejb-jar>
```

After having finished the deployment descriptor, our EJBs are all done. We compile our java files and package it all up into a file called `webshop-ejb.jar`, which contains the bean classes and the deployment descriptor `ejb-jar.xml` in the `META-INF` directory, as described earlier in this section.

Implementing the Tag Extension Libraries

For the sake of not going through the same process twice, I chose to combine the two tag extension libraries I designed into one. This was purely done to make the example more accessible, in book form and not something I am would recommend in a normal project.

In the design section, we decided that the user tag extension library needs four tags and that the cart tag extension library also needs four tags. These eight tags are:

the `login` tag:

```
<user:login name="username" password="password"/>
```

the `logout` tag:

```
<user:logout/>
```

the `getUser` tag:

```
<user:getUser id="user">
  Contents to exist only if someone is logged in.
  The logged in user is made available as the user variable.
</user:getUser>
```

the `noUser` tag:

```
<user:noUser>
  Contents to exist only if no-one is logged in.
</user:noUser>
```

the `updateProduct` tag:

```
<cart:updateProduct product="productname" count="1"/>
```

the `iterate` tag:

```
<cart:iterate id="product" countid="count">
  Contents to iterate
</cart:iterate>
```

and finally, the `totalcost` tag:

```
<cart:totalCost/>
```

The `login` and `logout` tag need only to call the `login()` and `logout()` methods on the `Cart` session bean. The `getUser()` tag should use the `Cart` bean to get the logged in user/customer and add an implicit variable to the JSP, and the `noUser` tag should just use the `Cart` to see if anyone is logged in and execute the contained code if nobody is. The `updateProduct()` tag will be an invocation to the `Cart` bean, the `iterate` tag will iterate over the results of a call to the `Cart` bean, and finally the `totalCost` tag will simply call the corresponding method on the `Cart`. As you might have noticed, all these tags contain very little actual logic, but they rather make up an interface between the JSPs and the EJBs. I still see them as a part of the logic layer, but they make up the logic layer's interface towards the information structure layer.

Tag extension libraries are covered in Chapter 8, and we will not cover them in detail here. We will look in only at the `getUser` tag here.

The `getUser` tag needs to get a reference to the `Cart`. This reference will be kept as a JSP session variable, and can easily be retrieved by the tag by accessing the `pageContext`. The contents of the tag should be evaluated only if someone is logged on, in which case a variable should also be set so that the contents of the tag can access the user.

```
package com.wrox.webshop.tags;

import java.rmi.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

import com.wrox.webshop.ejb.customer.*;
import com.wrox.webshop.ejb.cart.*;

public class GetUserTag extends TagSupport {

  /**
   * Method called at start of Tag
   * @return either a EVAL_BODY_INCLUDE or a SKIP_BODY
   */
  public int doStartTag() throws javax.servlet.jsp.JspException {

    // if no cart is available or noone logged in, don't evaluate body
    int returnCode = SKIP_BODY;

    // get cart.
    Cart cart = (Cart) pageContext.getSession().getAttribute("cart");

    if (cart != null) {
      try {
        Customer customer = cart.getCustomer();
        if (customer != null) {
          pageContext.setAttribute(getId(), customer);
          returnCode = EVAL_BODY_INCLUDE;
        }
      } catch (RemoteException e) {
        throw new JspException(e.getMessage());
      }
    }
```

```
      return returnCode;
    }
  }
```

A mentioned, this tag will need to introduce a new variable for access by the JSP content of the tag. Because of this we also need a `TagExtraInfo` class that will tell the JSP compiler about this variable:

```
package com.wrox.webshop.tags;

import javax.servlet.jsp.tagext.*;

public class UserExtraInfo extends TagExtraInfo {
  public UserExtraInfo() {}

  public VariableInfo[] getVariableInfo(TagData data) {
    return new VariableInfo[] {
      new VariableInfo(data.getId(),
                       "com.wrox.webshop.ejb.customer.Customer",
                       true, VariableInfo.AT_BEGIN)
    };
  }
}
```

Finally, the tag descriptor has to be written. The descriptor for this tag needs to contain a reference not only to the `Tag` class, but also to the `TagExtraInfo` class for the tag. We also want to specify that the content of the body of the tag is JSP, and that we have one attribute, called `id`, which is required. Based on this specification, the tag descriptor looks like this:

```
  <tag>
    <name>getUser</name>
    <tagclass>com.wrox.webshop.tags.GetUserTag</tagclass>
    <teiclass>com.wrox.webshop.tags.UserExtraInfo</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>getUser</info>
    <attribute>
      <name>id</name>
      <required>true</required>
    </attribute>
  </tag>
```

We merge the individual tag descriptors into a tag library descriptor (`.tld`) file, which becomes the deployment descriptor for our tag extension library.

We will also want to package the tags into one `carttags.jar` file for easy distribution and use in other projects. For this we need this file structure:

```
META-INF/
  Taglib.tld
com/wrox/webshop/tags/
  Tag classes
```

When we have this structure, we can easily create the `carttags.jar` using the `jar` tool included in the Java 2 SDK.

Now we have implemented the tag library, we have everything that is needed for the JSPs to function, so let's get on to those.

Implementing the JSPs

We implement the JSPs in the same order as we designed them. I chose to describe three JSPs in detail here: the product browsing page, the toolbar/cart page, and the logout page. The rest can be found online at http://www.wrox.com.

The Product Browsing Page

When we designed this page, we decided to use this structure for the page:

```
<title>
  Welcome to the Professional JavaServer Pages Book Store
</title>
<items>
  <item name="foo" price="$200" image="foo.gif"/>
</items>
```

The items were to be fetched from the database via the entity bean `Product`, and we want to use the `Product` EJB to generate a list of `<item>` tags. To do this we can use a tag extension library for using EJBs that is supplied with the Orion Application Server, and is available for free so that it can be used with any J2EE application server. You can download it at http://www.orionserver.com/tags/. This tag library contains tags that allow us to easily get hold of the home interface of beans, and to iterate through the existing beans. In this case this is done in the following way:

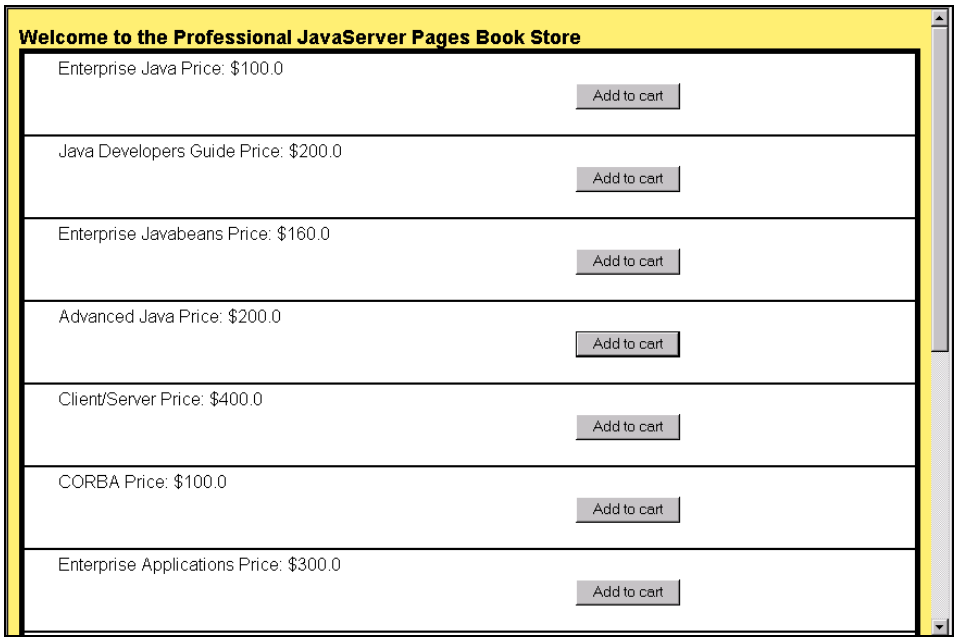
```
<ejb:useHome id="productHome"
    type="com.wrox.webshop.ejb.product.ProductHome"
    location="Product" />
<ejb:iterate id="product" type="com.wrox.webshop.ejb.product.Product"
    collection="<%=productHome.findAll()%>" max="-1">
  <item isbn="<jsp:getProperty name="product" property="ID"/>"
      name="<jsp:getProperty name="product" property="name"/>"
      price="<jsp:getProperty name="product" property="price"/>"/>
</ejb:iterate>
```

Together, this gives us the JSP:

```
<?xml version="1.0"?>
<?xml-stylesheet href="/items.xsl"?>
<%@ taglib uri="ejbtags.jar" prefix="ejb" %>
<doc>
  <title>
    Welcome to the Professional JavaServer Pages Book Store
  </title>
  <items>
    <ejb:useHome id="productHome"
        type="com.wrox.webshop.ejb.product.ProductHome"
        location="Product" />
    <ejb:iterate id="product" type="com.wrox.webshop.ejb.product.Product"
        collection="<%=productHome.findAll()%>" max="-1">
      <item isbn="<jsp:getProperty name="product" property="ID"/>"
```

```
            name="<jsp:getProperty name="product" property="name"/>"
            price="<jsp:getProperty name="product" property="price"/>"/>
      </ejb:iterate>
    </items>
  </doc>
```

The result of this page will look something like this (assuming the database has been populated with books and assuming that an XSL file has been written.):



The Toolbar/Cart Page

This page was designed to have an information structure that looks like:

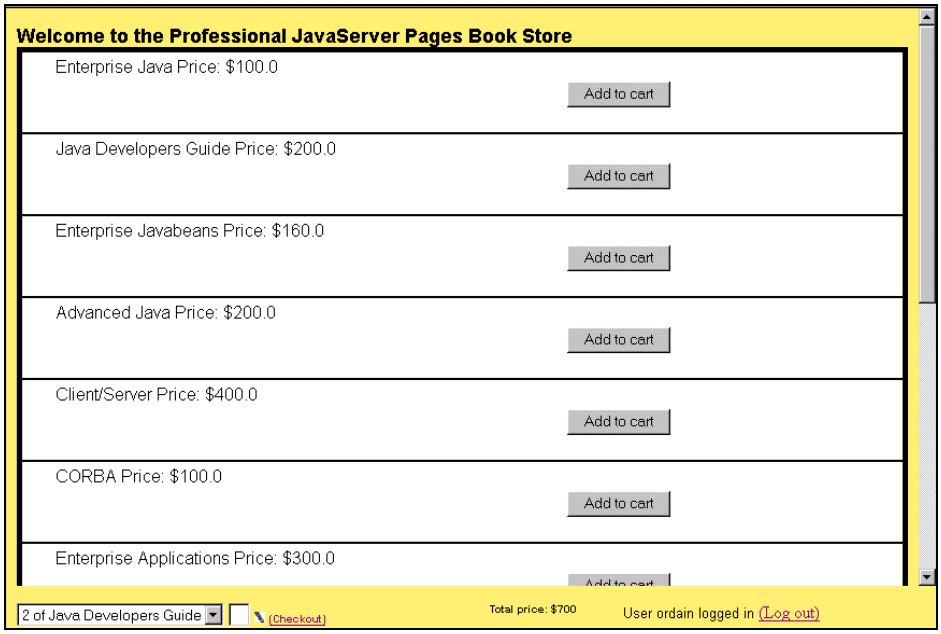
```
<cart>
  <item name="foo" price="$200" count="2">
  <item name="bar" price="$300" count="1">
</cart>
<user name="baz"/>
```

We want to implement this as a HTML JSP, to show how this looks compared to an XML JSP. This means that the JSP gets a fair bit larger, but we can still keep nearly all logic out of it by using tag libraries.

```
<%@ taglib uri="carttags.jar" prefix="cart" %>
<%@ taglib uri="ejbtags.jar" prefix="ejb" %>
<%@ page import="com.wrox.webshop.ejb.cart.*"%>
<%@ page import="com.wrox.webshop.ejb.customer.*"%>

<html>
  <body bgcolor="#ffee77">
    <cart:updateProduct product="<%= request.getParameter("product") %>"/>
```

```
      <table width="100%" border="0" cellspacing="0" cellpadding="0">
        <tr>
          <td valign="top">
            <form action="cart.jsp" method="get" name="">
              <select name="product">
                <cart:iterate id="product" countid="count">
                  <jsp:useBean id="product" scope="page"
                      type="com.wrox.webshop.ejb.product.Product"/>
                  <jsp:useBean id="count" scope="page"
                      type="java.lang.Integer"/>
                  <option value="<%= product.getID() %>" selected>
                    <%= count %> of <%= product.getName() %>
                  </option>
                </cart:iterate>
              </select>
              <input type="text" name="count" size="1">
              <input type="image" border="0" name="imageField"
                  src="update.gif" width="10" height="10">
              <b><font face="Arial, Helvetica, sans-serif" size="-2">
                  <a href="checkout.jsp" target="_top">(Checkout)</a>
              </font></b>
            </form>
          </td>

          <td valign="top">
            <b><font face="Arial, Helvetica, sans-serif" size="-2">
              Total price: $<cart:totalCost/>
            </font></b>
          </td>

          <td valign="top">

          <cart:noUser>
            <form action="login.jsp" method="get" name="">
              <div align="right">
                <font face="Arial, Helvetica, sans-serif" size="--">
                  Name
                </font>
                <b><font face="Arial, Helvetica, sans-serif" size="-2">
                  <input type="text" name="Name" size="10">
                </font></b>
                <font face="Arial, Helvetica, sans-serif" size="-1">
                  Password
                </font>
                <b><font face="Arial, Helvetica, sans-serif" size="-2">
                  <input type="password" name="Password" size="10">
                  <input type="submit" value="Log in">
                </font></b>
              </div>
            </form>
          </cart:noUser>

          <cart:getUser id="user">
              <font face="Arial, Helvetica, sans-serif" size="-1">
                User <%= user.getName() %> logged in
              </font>
            <a href="logout.jsp?page=cart.jsp">(Log out)</a>
          </cart:getUser>
        </tr>
      </table>
    </body>
  </html>
```

The result of this page looks something like this, with the product browsing page in the top frame and the cart in the lower frame:



The Logout Page

The third kind of JSP we use is a pure functional JSP without any presentation, and some people might say that this should be a Servlet and not a JSP. This being a book about JSP, I still want to show how this could be made using a JSP in a nice, clean way. Since this logout function might be used from many places, I include the possibility to supply a page parameter that specifies what file to load after the logout has been performed.

```
<%@ taglib uri="carttags.jar" prefix="user" %>

<user:logout/>
<jsp:forward page=" <%= request.getParameter("page") %>"/>
```

With all our JSPs implemented, we finally need to implement the XSL file we use for the product browsing page.

Implementing the XSL

The XSL transformation file for the product page mainly contains HTML. The key statements that get the information from the JSP and insert them into the HTML output are highlighted:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
                xmlns="http://www.w3.org/TR/xhtml1">

  <xsl:output method="xml" indent="yes"/>

  <xsl:template match="doc">
    <html>
```

```
        <body bgcolor="#ffee77">
          <font face="Arial, Helvetica, sans-serif" size="+1"><b>
            <xsl:value-of select="title"/>
          </b></font>
          <table width="100%" border="0" cellpadding="1" bgcolor="#000000">
            <tr>
              <td>
                <table width="100%" border="0">
                  <xsl:apply-templates select="items"/>
                </table>
              </td>
            </tr>
          </table>
        </body>
      </html>
    </xsl:template>

    <xsl:template match="items">
      <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="items/item">
      <tr bgcolor="#ffffff">
        <td>
          <form target="cart" action="cart.jsp" method="get" name="">
            <table width="100%" border="0">
              <tr>
                <td width="3%"></td>
                <td width="71%">
                  <font face="Arial, Helvetica, sans-serif">
                    <xsl:value-of select="@name"/>
                    Price: $<xsl:value-of select="@price"/>
                  </font>
                </td>
                <td width="26%"></td>
              </tr>
              <tr>
                <td width="3%"></td>
                <td width="71%">
                  <div align="right">
                    <input type="Submit" name="action" value="Add to cart"/>
                    <input type="Hidden" name="product" value="{@isbn}"/>
                  </div>
                </td>
                <td width="26%"></td>
              </tr>
              </table>
          </form>
        </td>
      </tr>
    </xsl:template>
  </xsl:stylesheet>
```

Packaging the Web Application

As well packaging the EJBs into `webshop-ejb.jar` and the tag library into `carttags.jar`, we want to package our entire web application as an archive. To do this we prepare a file structure for the web application:

```
WEB-INF/
web.xml
  lib/
    carttags.jar
    ejbtags.jar
    utiltags.jar

cart.jsp
checkout.jsp
index.html
items.jsp
items.xsl
login.jsp
logout.jsp
purchase.jsp
```

The directory `WEB-INF` will contain one file, called `web.xml`, which is the deployment descriptor for the web application.

When we have this file structure, all we have to do to create a web-application archive is to package it up as `webshop-web.war` using the `jar` tool.

Packaging the Enterprise Application

We now have two files, a file for the EJBs, `webshop-ejb.jar` and a file for the web application, `webshop-web.war`. We want to make this into one single distributable Enterprise application archive file, `webshop.ear`. To do this we need an application deployment descriptor, a file is called `application.xml` which must be located in the `META-INF` directory in the root of the `webshop.ear` file. This means we have a structure with three files:

```
META-INF/
  application.xml
webshop-ejb.jar
webshop-web.war
```

The `application.xml` deployment descriptor will basically just contain a name and links to contained modules; it looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN" "http://java.sun.com/j2ee/dtds/application_1_2.dtd">

<application>
  <display-name>Wrox webshop</display-name>
  <module>
    <ejb>webshop-ejb.jar</ejb>
```

```
    </module>
    <module>
      <web>
        <web-uri>webshop-web.jar</web-uri>
        <context-root>/</context-root>
      </web>
    </module>
  </application>
```

Summary

J2EE is a far too large a subject to be fully described in one chapter. Please see this chapter as a short introduction, and go on to read up on J2EE elsewhere. J2EE is becoming a very big thing in the industry, and has the possibility of bringing standard reusable components to the server and revolutionizing the whole industry. Only the future can tell whether this will happen or not.

Further Reading

For more detailed descriptions of how to use J2EE technologies to construct enterprise applications, refer to the J2EE home at Sun Microsystems at http://java.sun.com/j2ee/.


J2EE
JSP