Developing a J2EE Application –

Web Auction

Gerald Mo

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the University of Wales

Supervisor: Chris Loftus

University of Wales, Aberystwyth

31st March 2004

DECLARATIONS

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed …… (**Gerald Mo**)

Date ……

This dissertation is being submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Signed …… (**Gerald Mo**)

Date ……

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references to the bibliography. A bibliography is appended.

Signed …… (**Gerald Mo**)

Date ……

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organizations.

Signed …… (**Gerald Mo**)

Date ……

Acknowledgements

The author would like to acknowledge the help and support of the many people whose hard work, guidance, friendship and understanding which were crucial to the production of this dissertation.

The support and guidance of my supervisor, Chris Loftus

The support and understanding of my family and friends

Lastly, my fellow course mates for creating a support and friendly environment during my study for my Master course at the University of Wales, Aberystwyth.

Abstract

The aim of this dissertation is to build a scalable and distributable “online” web auction application using the Java 2 Enterprise Edition specification. J2EE is a Java platform designed for large-scale computing typical of large enterprises. It’s designed is to simplify application development in a thin client tired environment. J2EE simplifies application development and decreases the need for programming by providing reusable modular components and by enabling the tier to handle many aspects of programming automatically. The programming strategy centers on EJBs, which are at the heart of most enterprise-level Web applications. EJB integrate data management, session management, and business logic, and coordinate among all the tiers of the application. The application design goal is separated into two major components: presentation logic and business logic. Client use Web browsers to interact with the presentation layer. The presentation layer is responsible for relaying client request to the business logic layer and rendering the business logic’s response into HTML responses. The business logic layer handles the application’s logic and communications with back-end systems such as a database.
The presentation layer is implemented with Web components including JSP pages and tag libraries. WebAuction implements the business logic layer with EJB components, JMS, JavaMail, and JDBC.
Separating the presentation and business logic allows for parallel development, which a less important factor for the purpose of the dissertation. However, the separation means a much more maintainable application by using appropriate interface which promotes information hiding, minimizing exposure of internal information to other areas of the system. This minimizes dependencies: components may be redesigned or re-implemented without affecting other parts of the system.

Table of Contents:

1. J2EE overview

Java enables developer to write codes that are platform independent, any computer with a Java virtual machine (JVM) can run Java byte-code. Java objects can be broken down and sent across a network and reconstituted as object within a remote JVM, this greatly enhance interoperability across different platforms. This can be particular advantageous when trying to integrate legacy systems.

The J2EE specification provides sophisticated set of distribution APIs, with vendors and Open Source organizations providing implementations.

J2EE also defines an application server framework comprising of two kinds of containers, responsible for managing specific kinds of J2EE component. Web container manages the life cycles of serlvets and JSPs, EJB containers manages the life cycle for EJBs. These containers perform much of the work that would normally have to be coded explicitly within an application., code that is often dependent on the environment the application operates within, which typically requires to be updated relatively often. An application server simplifies this process via XML declarations. This minimizes accidental changes to the business logic rather than distribution behaviors as intended.

The J2EE application server allows development of components that consist mainly of business logic code. These components can be reusable business objects that can be configured for use in multiple applications. The J2EE framework allows developers to separate the development of graphical user interface components supported within a Web tier (Web Container) and the business logic supported within an EJB tier (EJB containers). The decoupling the presentation from business logic and data is seen as a good practice, described by design patterns, such as Model – View – Controller[1].

[1] Crawford and Kaplan, (2003) J2EE design patterns. O'Reilly

J2EE consists of a set of specification and application programming interfaces that build on top of the J2SE platform. J2SE provides APIs that are appropriate for the development of standalone applications, simple networked applications, or two-tier applications, for example client to database interaction. J2EE adds to the existing J2SE APIs by providing explicit support for developing server-side applications. These are applications that have the following characteristics[2]:

- They are multi-tiered. Each tier has a particular responsibility, allowing both the separation of concerns and the physical separation of different aspects of the application. Physical separation provides for improved security, performance optimization and increased reliability.
- They are often accessed from clients via HTTP, supporting customer to business model of interaction or business to business model of interaction.
- J2EE builds on the security model provided in J2SE, via authentication, authorization and encryption of data in transit.
- Supports both synchronous and asynchronous communication.
- Able to handle high volumes of incoming client requests, supporting concurrency and threading issues.
- Adhere to the ACID (Atomicity, Consistency, Isolation, Durability) transaction principles, and within a distributed context.

1.1 Presentation logic

Presentation logic is the server-side code in a server application that determines the client-side response to a specific request.

1.1.1 Java Servlets

[2] Allamaraju, Subrahmanyam (2001) Professional Java server programming J2EE. Wrox Press Inc.

The Java servlet is a server-side technology that accepts HTTP requests from a Web browser and returns HTTP responses. Servlets, which can be multi-threaded, have performance advantages over CGI for coding presentation logic for a Web client.

Servlets operates on the request-response model. Requests come into the servlet engine. The server then executes the appropriate servlet and returns a response to the client.

The most commonly used servlet type is the HTTP servlet designed to fill HTTP protocol requests. HTTP servlets provide the following core features:

- HttpRequest objects capture request details from requests submitted via Web page forms, including data availability, protocol types, security levels, and so forth
- HttpSession objects specific to each user handle user session information in the server.
- HttpResponse object capture response details. The servlet developer can output everything that is sent back to the client making the request. The servlet engine handles the rest.

1.1.2 JavaServer Pages (JSPs)

The JSP technology gives developers a simple, HTML like interface for creating servlets. JSP can contain HTML code, Java code, and code modules called JavaBeans. The JSP technology provides the same functionality as servlets, but the development interface is easier to use. When a JSP page is requested for the first time, the application server compiles that page into a servlet. This servlet is then executed to serve further request. In this way, the servlet engine and the JSP engine are intimately tied together.

The benefit of JSP pages is their simplicity: they look like typical HTML pages, which allows standard Web composition tool to edit JSP pages.

1.1.3 JavaBeans and Tag Libraries

JavaBeans (which are different from EJBs) are Java components (classes) that developers use in Server applications to encapsulate data, either for display or for actions against the database. Developers create classfiles with a number of methods, which are typically used to get and set values. JSP pages have special tags for including JavaBeans and automatically populating them with values. The JSP page calls methods on those JavaBeans to help create its HTML output.

Tag libraries supply custom HTML-like tags for use in JSP pages. Tag libraries abstract Java code into tags that can be easily manipulated by Web editors and designers. To build a tag library, a developer creates classfiles and a file called a Tag Library Descriptor that lists the available tags from the tag library.

JavaBeans and tag libraries manage the data and Java code that interacts with the data sources available via JDBC and EJB.
JavaBeans and tag libraries perform a valuable service by enabling Web application developers to keep explicit Java code out of JSP pages and servlets. This modularization minimizes the chance of accidental damage to the JSP page during an HTML editing session, and permits presentation logic to be changed independent of the JSP page[3].

1.2 Database and Transaction Support

Database and transaction support is provided by Java Database Connectivity (JDBC) and the Java Transaction API (JTA). The high level interface to database use is provided by Enterprise Java Beans (EJBs).

1.2.1 Java database Connectivity (JDBC)

[3] Weaver, Mukha and Crume (2004) Beginning J2EE 1.4 From Novice to Professional. APRS

JDBC is the Java standard for database connectivity[4]. The JDBC specification provides everything needed to connect to databases from a standard set of Java APIs. Vendors supply JDBC "drivers" that map this standard set of Java APIs to the specifics of the underlying database.

JDBC is the bridge that connects the application Server with the database, from a programming standpoint. This functionality is transparent to the programmer: It's provided by EJB. The developer does not program JDBC directly, except in special cases

The typical Server application relies on a database for key e-commerce application functionality such as transaction support, support for concurrent data access, and data integrity features. Relational databases support a common declarative language for access called Structured Query Language (SQL).

1.2.2 Java Transaction API (JTA) Support

JTA gives Web application developers access to the transaction functions in database systems, or any legacy data store. Transactions coordinate single-database and multi-database operations to ensure that all data resources remain accurate and consistent and that operations against the database are repeat able and durable. Transaction management is essential for enterprise-level e-commerce applications, which need to be Web-based and fault-tolerant.

JTA defines a high-level transaction management specification for resource managers for distributed applications. Application Servers transaction services provide connectivity and support for database transaction functionality, most notably two-phase commit (2PC) engine used to manage multi-database transactions.

[4] http://java.sun.com/products/jdbc

1.3 Object Registry and Remote Method Invocation (RMI)

this section discuss the usage of Remote Method Invocation and Java Naming and Directory Interface in a distributed web application.

1.3.1 Java naming and Directory Interface (JNDI)

JNDI is the Java standard for the "central registry" of naming and directory services. JNDI manages references to the core components needed to build distributed applications. When a developer builds an application that accesses a remote object, JNDI provides the application with a way to locate that object. The JNDI technology is the interface to naming and directory services, and acts as a central registry for named application and data objects.

The JNDI services help assure the proper level of uniqueness in the names of application components, and help prevent, diagnose, and treat naming conflicts that might arise.

The usage pattern of JNDI is relatively simple. Application developers do an initial lookup to find the object that they require in the application server deployment. Application services will return everything the application Entity needs to access that object.

1.3.2 Remote Method Invocation (RMI)

RMI (Remote Method Invocation) is a way that a programmer, using the Java programming language and development environment, can write object-oriented programming in which objects on different computers can interact in a distributed network. RMI is the Java version of what is generally known as a remote procedure call (RPC)[5], but with the ability to pass one or more objects along with the request. The object can include information that will change the service that is performed in the remote computer. Sun Microsystems, the inventors of Java, calls this "moving behavior." For example, when a user at a remote computer fills out an expense account, the Java program interacting with the user could communicate, using RMI, with a Java program in another computer that always had the latest policy about expense reporting. In reply, that

[5] Oberg (2001) Mastering RMI: Developing Enterprise Application in Java and EJB.

program would send back an object and associated method information that would enable the remote computer program to screen the user's expense account data in a way that was consistent with the latest policy. The user and the company both would save time by catching mistakes early. Whenever the company policy changed, it would require a change to a program in only one computer.
Sun calls its object parameter-passing mechanism object serialization. An RMI request is a request to invoke the method of a remote object. The request has the same syntax as a request to invoke an object method in the same (local) computer. In general, RMI is designed to preserve the object model and its advantages across a network.
RMI is implemented as three layers:
A Stub program in the Client side of the Client/Server relationship, and a corresponding skeleton at the server end. The stub appears to the calling program to be the program being called for a service. (Sun uses the term proxy as a synonym for stub.)
A Remote Reference Layer that can behave differently depending on the parameters passed by the calling program. For example, this layer can determine whether the request is to call a single remote service or multiple remote programs as in a multicast.
A Transport Connection Layer, which sets up and manages the request.
A single request travels down through the layers on one computer and up through the layers at the other end.

1.4 Enterprise JavaBeans (EJBs)

EJB is the enterprise Java standard for building server-side business logic in Java[6]. Whereas presentation logic automatically handles the type and format of information to be displayed to clients, business logic is used for operations such as funds transfers, product orders, and so forth. Developers build EJBs that take advantage of services provided by the EJB Server container.
This container provides services including transaction support and security, and handles concurrency issues. All of these services are required for scalable, secure, and robust electronic commerce applications.

[6] http://java.sun.com/products/ejb

There are four basic types of EJBs:

- Entity
- Message-driven
- Stateful session
- Stateless session

The EJB specification defines an API for developers to create, deploy, and manage cross-platform, component-based enterprise applications. The EJB component model supports three types of components:

- Session beans, which capture business rules and methods that persist for the duration of a session
- Entity beans, which encapsulate specific data items from a database
- Message-driven beans, which integrate EJBs with the Java Message Service (JMS)

1.4.1 Entity EJBs

Entity EJBs {entity beans) are the enterprise Java standard for representing data. They are standard Java language objects that reside in the Web Server container. In most cases, entity beans represent data from a database, although they also can represent data stored in other locations. Objects such as entity beans need to be mapped to the relational structure of a relational database management system (DBMS).

1.4.2 Session EJBs

The enterprise Java standards specify two types of session beans: stateless and stateful. Stateless beans receive requests via RMI but do not keep any data associated with the client they are serving internally. Stateful beans, on the other hand, keep data specific to

the client they are serving. From a developers perspective, these two types of session beans are similar in construction.

WebLogic Server session beans handle requests that arrive via RMI[7]. Typically they provide services to other Java objects. This is in contrast to servlets and JSPs, which are focused primarily on responding to requests from Web clients such as Web browsers (whose requests arrive via HTTP). The objects that initiate requests to session beans can be any arbitrary object that is able to access the appropriate RMI client classes.

1.4.3 Message-Driven Beans (MDBs)

Message-driven beans integrates session beans or entity beans with the Java Message Service. In both cases, a synchronous programming model is used. Clients make requests to the EJB and wait for work to be completed on their behalf. Using MDBs, the EJB is not attached to a client. Instead, it is attached to a message queue or topic defined in the JMS. When a message arrives, a method on the EJB is executed.

MDBs introduce an asynchronous processing paradigm to enterprise Java applications. Task can be queued and made available for processing when resources are available.

1.5 Java Message Service

The JMS specification provides developers with a standard Java API for enterprise messaging services such as reliable queuing, publish and subscribe communication, and various aspects of push/pull technologies. JMS is the enterprise Java standard for messaging. It enables applications and components in Java to send and receive messages.

There are several paradigms for messaging in JMS, including:

[7] Zuffoletto (2003) BEA Weblogic Server Bible

- Queue model
- Topic-based, publish-subscribe system

The queue model enables JMS clients to push messages onto a JMS queue. Clients can then retrieve these messages. The topic-based model enables publishers to send messages to registered subscribers of the JMS topic.

1.6 Java Mail

The JavaMail API provides classes that support a simple email and messaging service, as well as connection to any standard email system. The JavaMail interface provides a standard, object-oriented protocol for connection to many different types of email systems.

1.7 Security

The J2EE security Model is still evolving. The Java Authentication and Authorization service (JAAS) provides the framework for authenticating clients and authorizing differential access to application resources.

2. WebLogic Server

BEA WebLogic Server is a Java™ application server that supports enterprise-level, multi-tier, fully distributed Web applications. WebLogic Server is widely recognized as the market leader and de facto industry standard for developing and deploying Java e-commerce applications[8]. BEA WebLogic Server:

[8] Heaton (2003) BEA Weblogic Server for Dummies.

- Maintains and manages application logic and business rules for a variety of clients, including Web browsers, applets, and application clients.
- Supports software clustering of WebLogic Servers for running both Web and Enterprise JavaBeans (EJB) services to ensure reliability, scalability, and high performance.
- Provides the application services necessary for building a robust, scalable. Web-based application.

• Provides a current and complete implementation of the protocols of Sun Microsystems' Java 2 Platform Enterprise Edition (J2EE).

With its emphasis on maximizing efficient use of system resources such as client and database connections, BEA WebLogic Server can support e-commerce application for millions of users and hundreds of thousands of request per hour.

2.1 WebLogic Server Strengths; Component Support and Scalability

WebLogic Server provides several important APIs and extensions to J2EE APIs that help provides reliable, scalable performance for a distributed application[9]. The WebLogic Server clustering technology permits interconnection of several WebLogic Server instances(one per CPU) on a LAN , so that WebLogic Servers in a cluster can distribute workload and provide fault-tolerance as application demands increase.

WebLogic Servers implementation of the J2EE server-based programming strategy centers on EJBs, which are at the heart of most enterprise-level Web applications. EJBs integrate data management, session management, and business logic, and coordinate among all the tiers of the application. For example, you use entity beans to represent data from the database. You use session beans to implement business logic that is either too

[9] Zuffoletto (2003) BEA Weblogic Server Bible

complex or too sensitive to be managed with presentation logic, and you use message-driven beans to set up asynchronous data processing.

Within the WebLogic Server container, components are given connection and communications services, transactional support for multi-user operations, and the capability to replicate, or cluster, to provide better performance and scalability.

To the container and component framework, WebLogic Server added several important mechanisms for clustering to ensure high availability and scalability of distributed applications. A BEA WebLogic Server cluster is a group of WebLogic Servers that coordinate their actions to provide scalable, highly available services in a transparent manner. WebLogic Servers in a cluster can run on a heterogeneous mix of hardware and operating platforms: They interoperate through their Java-based, platform-independent APIs. WebLogic Server clustering technologies transparently support replication, load balancing, and failover for both Web page generation (presentation logic) and EJB components (business logic).

3. Requirement Specification for the WebAuction Application

For WebAuction, It is to be a model of a "web auction" site similar to existing auction sites on the Internet. Functionally, the WebAuction application supports:

1. Secure user registration
2. Email validation of user's login credentials
3. Browsing, for both registered and unregistered users
4. Browsing by category for auction items (with some minor searching functionality)
5. Placing bids
6. Email confirmation of bids
7. Viewing open bids

4. Application technology Requirements

Based on the functional specification, the WebAuction technology requirements include services for:

- Storing data in a database. User account information should be stored persistently, in a database, on disk. In the event of a power failure, we do not want to lose any user information.
- Handling concurrency issues. Concurrency becomes an issue when multiple threads of execution try to access the same resource at the same time. For example, users might try to bid on a given item simultaneously. The developer should enclose data access operations in transactions, which can help prevent or resolve concurrency issues. Both the database and the WebLogic Server container provide support for transactions.
- Handling user sessions. Because the WebAuction application needs to handle many concurrent users, session information must be maintained for each user. A session begins when the user attempts to log in. The application must determine that a user is valid before allowing that user to access the auction area. If the user is not a registered user, logic should exist to point the user to a "create user account" page.
- Representing database data. The WebAuction application s data is stored in a relational database. However, the application uses that data in the context of the object-oriented Java environment. The technology requirements therefore include mechanisms to represent relational data in a way that is consistent with the J2EE environment.
- Updating data from the Web interface. In the WebAuction application, users can update their personal inventory and data on items for auction. Technology must support updating the database from a Web form.
- Querying data. Many features of WebAuction require searching and querying data. For example, one feature requires that users be able to browse auction items by category, such as "all the books available for auction."

- Securing user information. It must be possible to protect user information and accounts so that other users cannot access them. It must be impossible to access another users
- information inside of the WebAuction application, and for someone to view the auction information that belongs to a given user while it is in transit to that user.

5. JavaServer Pages (JSPs)

The JSP standard is a result of the recognition by the J2EE community that there are typically two distinct groups working together to create a Web application: the programming team and the Web design team. Programmers build the code that executes business and presentation logic[10]. The Web design team builds the HTML pages and associated graphics.

JSP enables the separation of the two task effectively, enabling programmers to develop business and presentation logic, while the Web development team focuses on site composition. The Java classes that implement business and presentation logic are simply called from a standard HTML page, using special tags and syntax.

JSP is the main technology used in the WebAuction application, JSP is chosen over servlets for presentation as it facilitates the following:

- WebAuction could be upgraded to a more graphics extensive web pages, JSP are suitable for building HTML pages that are not trival (having more than a few lines and with advance features such as tables). Servlets uses the out.print line to produces response web page, this is not ideal for the WebAuction application.
- JSPs enables easy HTML code changes, the presentation can be change relatively quickly and easily.

[10] Asbury (2001) Developing Java Enterprises Application.

JSP supports two different mechanisms for embedding code and using more complicated logic. JavaBeans for JSP are not the same as the EJBs. JSP JavaBeans are basically special Java classes that is use to capture business logic for methods called from a JSP page. JavaBeans are used by creating a generic Java class with methods that include your Java code. JavaBeans can be used with JSP by using the useBean tag: <jsp:useBean…>

JavaBeans are used primarily for encapsulation data, rather than business logic, to modularize the Java code for reuse in something other than JSP and lastly to minimise the chance of having the HTML designers accidentally modifying the Java code.

Custom tag libraries are a second mechanism for specifying business and presentation logic in JSP pages. Custom tag libraries enables developer to have a custom define HTML-like tags that can be used by Web developers.

JSP is a sequence of HTML interleaved with special tags that the JSP container uses to generate responses to requests. There are three categories of JSP tags.

- Directives, which are messages that our page sens the JSP container.
- Scripting elements, which are variable declarations, Java code to be executed, and expressions
- Actions, which are messages to the JSP container that affect how responses are handled by the JSP container

There are three main elements that make up a JSP, Directives, Scripting element and implicit objects and finally Expressions. The characteristics and usage of each are discussed below:

5.1 Directives

Directives can be specified to a JSP container. For example, an error page could be specified if there are problems with the JSP page. Alternatively, Java classes could be specified to include during execution.

Directives are specified by tags that use the <%@ and %> characters. For example, a page directive that tells the JSP container to import all the classes in the java.util package is:

<%@ page import = "java.util. " %>*

There are several types of directive for JSP:

- Page directives, which are specific to the current page.
- Include directives, which are specifying how to include another file in a JSP.
- Tag library directives, which specify how to include and access custom tag libraries, this enables developers to create custom tags for Web developers to use.

5.2 Page Directives

A page directive tag begins with the characters <%@ page and then includes directives types and values in the body of the tag. For example, the following tag is a page directive that tells the JSP container that the JSP page should be activated to work with sessions.

<%@ page session = "true" %>

One of the most commonly used directives in the WebAuction application is the include directive. The include directive includes any arbitrary text or JSP page code when the JSP is executed. The include directive refers to a file specified by a path relative to the current location of the JSP page.

The include directive is delimited by the <%@ and %> characters. For example, if you want to instruct the JSP container to include the contents of a text file named MyDisplaytext.txt in the output of the JSP, the following code is embedded in the web page:

<%@ include file = " MyDisplaytext.txt" %>

Include directives are very powerful in the context of a J2EE web application. They enable the modularization of JSP pages, making development and maintenance much simpler. For example, a navigation bar can be place into every JSP page by creating a single page called, for example, navigation.jsp. The output of this JSP page would be the navigation bar that is shown to every user. The same technique can be extended to page headers and footers, hence different JSP pages can be added together to create the entire site. If a change in one of the element or component which needs to be reflected on all pages, only the relevant JSP page needed to be modified, and the updated version would be reflected any subsequent JSP which it is included in. The WebAuction site uses this technique for the header and for the navigation bar.

5.3 Implicit Objects and Scripting Elements

Scripting elements are a way to specify arbitrary Java code for the container to execute. The major convenience of using JSP instead of arbitrary application code is that JSP contains many predefined implicit objects, so no code is required to generate an output stream, instantiate an HTTP response object and so forth, it is all handled by the container.

There are three types of scripting elements for use with JSP. These are:

- A declaration, which enables the declaration of methods, variables etc in a JSP page. They are used to declare functions and variables to be used during the execution of the page.

- A scriptlet, which is a Java code fragment to be executed when client request the JSP page. Scriptlets define the logic of the JSP pages.

- An expression, which is a Java code expression that is evaluated, converted to a string and then sent to requesting client. Expressions are used to quickly embed dynamic values into the JSP pages.

5.4 Declarations

Declarations are scripting elements that declare methods, variables, or both in a JSP page. The special tags that designate declarations begin with <$! And end with %>. Consider the following:

<%! Int MyVariable = 3; %>

The integer variable MyVariable has been declared and assigned the value of 3. This variable, like all declarations, has a scope limited to the current page. A variable declared in a particular page can be access by any scripting element on that JSP page but not by any other JSP page by default.

5.5 Scriptlets

Scriptlets are essentially code fragments that exist in the JSP page and are executed by the container to service client requests. Scriptlets are designated using the <% and %> characters. The example below shows a basic scriptlet:

```
< %
        myVariable = inc (int myVariable);
%>
```

the above scriptlet increments the myVariable. As with declarations, scriptlets by default have a scope limited to the current page.

5.6 Expressions

Expressions are scripting elements that contain valid expression in Java. At execution time for the JSP page, the expression is evaluated, converted to a string, and then placed in the implicit object out. These expressions can be any valid expression in Java. To designate an expression in a JSP page, the <% = and %> is used to signify the beginning and end of the expression.

5.7 Actions

Actions are special JSP tags for use with implicit objects and other server-side objects, and for use in defining new scripting variables. Actions are defined using XML syntax only. Actions typically take the form of <jsp: action name/>. For example, the following actions specifies to the JSP container to include the output of a JSP page name mypage.jsp in the output of the JSP page:

```
<jsp: include page = "mypage.jsp" />
```

Actions are used for certain types of operations: for example, to include the output of another JPS page in the current JSP page. One instance in which this is helpful is if the single JSP page that dynamically generates the copy-right information for each page of the Web site, alternatively the capability to forward a user to another JSP page. All this can be done with the standard actions tags which enable this to be done in a line of code.

There are generally two types of actions available to developers:

- Standard actions are available, by definition, in every JSP container. No importing of any special classes are necessary to use them
- Custom actions are available to be "plugged-in" through the use of a JSP feature called tag libraries.

Standard Actions

There are seven standard actions available in a JSP 1.1 compliant container; these have varied functions and uses:

<jsp:include>

The include standard action provides for the inclusion of static and dynamic resources in the current JSP page. The tag specifies the relative URL of the resource to be included in the following format:

<jsp:include page = " myWebPage.html " />

the include tag is able to refer to any piece of content including HTML pages, JSP pages, servlets etc. the only requirement is that the relative URL specified with the page attribute must be a valid resource type for the JSP container or Web server and located in the same application context.

<jsp:forward>

The forward action allows for the dispatch of the current JSP request to another resource. The tag specifies the resource to which the request is forwarded using the page attribute. The forward tag begins with <jsp:forward and is completed with / >. For example, the following tag forwards the JPS request to the resource named myWebPage.jsp:

<jsp: forward page = "myWebPage.jsp" / >

The forward tag is useful to redirect misdirected request. In the WebAuction application, the forward tag is used to check that a user still has a valid account, if no, they are forwarded the user to a page which requires them to jsp page which requires them to login again.

It is important to note that a JSP forward is different to an HTTP redirect. With an HTTP redirect, a message is sent back to the Web browser, instructing it to forward to another resource. For example: response.sendRedirect(myNewURL);

This causes the new URL to display in the Web browser. It also requires another communication between the server and the client browser. On the other hand, the JSP forward is processed on the server. The JSP forwards handling of the response to another resource. As a result of this JSP forwards is prefer over HTTP redirect, expect in cases where data processing is required and exist concerns if a user click s the Reload function on the Web browser, thus resending the data.

<jsp:param>

when the forward or include actions are used, the included page or forwarded page sees the original request object. The original parameters of the request can be augmented with new parameters, as specified by the param tag. The param action provides name/value

information pairs to other actions. The format of the param tag begins with < jsp : param and is completed with / >. The following param tag specifies a name/value pair of phoneType and mobile:

<jsp: param name = " phoneType" value = " mobile" / >

The param element is used in the included, forward, and plug-in actions to provide information to other pages in the form of request parameters. These parameters are additive: each parameter added in the request takes precedence over existing parameters.

<jsp:plugin>

the plug-in action instructs the client browser to download the Java plug-in for executing a client-side job. The capability to invoke a plug-in helps the application developer overcome compatibility and versioning issued caused by the wide variety of Web browsers. By using this tag, it can be sure that the applet or JavaBeans executes on the appropriate JVM.

5.8 Java Beans and JSPs

The JSP specification enables the inclusion of JavaBeans. As mentioned earlier, JavaBeans encapsulate presentation logic or rudimentary business logic in JSP pages. By encapsulation logic in a JavaBean, the compromising or corrupting of Java code accidentally by Web page developers could be avoided.

The integration of JavaBeans with JSP pages is done through the use of three JSP actions:

<jsp:useBean>

the useBean tag enables a JSP developer to specify a JavaBean to be included in the JSP page. The useBean tag attempts to instantiate a JavaBean, and gets any parameters that are specified. There are three major parameters that affect eh capability of the container to instantiate the JavaBean. These are: the scope of the JavaBean, an ID field that represents the bean's name as it should be referenced in the application and the class name of the JavaBean.

Consider the following complete syntax:

<jsp: useBean id = "mybean" class = " com.myapp.mybean" scope = "page">
</ jsp: useBean>
<jsp: useBean
id = "beanInstanceName"
scope = "page | request | session | application"
{ class = "package. class " |
type = "package. class" |
beanName = " { package.class | <% = expression %> }

there are a number of possible attributes for the useBean tag:

- id – represents the bean's name as it should be referenced in the application, and the class name of the JavaBean. This is the name which Java code in the JSP page uses to access the JavaBean instance.

- class - specifies the complete class name representing the JavaBean. It does not include the class extension.

- Class – the fully qualified name of the class that defines the implementation of the object.

- BeanName – the name of the JavaBean.

- Scope – specifies the scope in which the JavaBean is available. The four possible values:
- Page: the JavaBean is available for the current page. Your JavaBean is discarded upon completion of the current request for the JSP.

- Request – the JavaBean is available from the current page's ServletRequest object. This is useful when using forward and include another JSP page using the page directives described above.

- Session – the JavaBean is available for the duration of the user session.

- Application – the JavaBean is available indefinitely and is stored in the current page's ServletContext object as defined for Web applications.

Properties allow for convenient setting and retrieving of parameters in a JavaBean. They are useful for streamlining the retrieval of values in a JavaBean.

<jsp:setProperty>

The setProperty tag lets the JSP developer set the properties in a given JavaBean.

<jsp:getProperty>

The getProperty tag allows a JSP page to query a JavaBean for a given property. It is the opposite of the setProperty tag, getProperty puts the value of the property in the JSP's out object for display back to the client.

The most powerful use of JavaBeans is when they are automatically populated with request parameters. Requests contain parameters such as those included in a Web form.

Each of these parameters contains a name and a text value. The JavaBean model with JSPs allows for a JSP with the appropriate get<name> and set<name> methods to be automatically filled.

In the WebAuction application, a suite of JSP pages are used to handle the user interactions. This is demonstrated using browseitem.jsp and its interaction with the JavaBean ItemBean that encapsulates the Java cod so that it does not appear in the JSP page.

The browseitems.jsp is called when a user wants to look at the current auction items. There are different categories of items up for auction including books, computers and clothing. When a request comes in to view items of a certain type, the ItemBean queries the appropriate EJB and outputs the data. The following section describes the construction and functions of the browseitem.jsp in more detail.

browseitems.jsp

firstly, the document type is defined, with HTML header, and body style:

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en"
<html>
<head>
        <title> Webauction:Browse Items</title>
<head>
<body text = " #000000" bgcolor = "#FFFFFF">
```

The above describes the content type of this HTML document. Subsequent tags specify the HTML document title and set the colors of the body of the HTML document. This is then followed by a page which specifies that this page is visible only in the context of a session, and error page is also specified.

```
<%@ page session = "true" errorPage = "error.jsp" %>
```

The page also uses a JavaBean. The following tag directs the web server to instantiate an instance of WebAuction.jsp.Itembean and makes that available for the JSP page under the identifier "itembean":

```
<jsp:useBean  id = "itembean"  scope = "page"
  class = "Webauction.jsp.Itembean" / >
```

Next, text and table for the links for the different categories:

```
<H2>Select a category to browse for Items</H2>
<CENTER>
<table width = "100%" bgcolor = "#000ff" fgcolor = #FFFFFF">

<TR>
```

the links contained in the page enable users to browse through the categories of auction items. If a user click on one of those links, a parameter is created name cat, which is short for category. Links are created that pass this parameter automatically when clicked by adding the parameter and value to the URL. For example, a relative URL of "browseitems.jsp?cat = books" passes the parameter cat with a value of books:

```
<TD ALIGH =  "CENTER"><A
href = "browseitems.jsp?cat=books"><H4>Books</H4></A></TD>
<TD ALIGH =  "CENTER"><A
href = "browseitems.jsp?cat=clothing"><H4> Clothing </H4></A></TD>
<TD ALIGH =  "CENTER"><A
href = "browseitems.jsp?cat=computers"><H4> Computers </H4></A></TD>
<TD ALIGH =  "CENTER"><A
href = "browseitems.jsp?cat=electronics"><H4> Electronics </H4></A></TD>
```

```
</TF>
</table>
</CENTER>
```

When a user chooses to browse a category by clicking on a link, the following scriptlet invokes the JavaBean and deals with it appropriately. First, the category is recognized by looking for it in the request using the getParameter() method on the implicit object request. If the category exists, a method on the JavaBean is called to request all the items in the category. In addition, the current user's name is located in the session object and sent to the JavaBean.

```
<%
        String category =  request.getParameter("cat");
        If(category ! = null) {
        String userName = (String) session.getAttribute("username");
        Itembean.outputItemsInCategory(out,category,userName);
        }
%>
```

Finally, the JSP page is finished by adding the appropriate closing HTML tags:

```
</body>
</html>
```

it is noted that there is very little actual Java code encapsulated in the JSP page, this mages the page more maintainable and present less of the Java code in a form that Web developers can see and change accidentally. The JSP page relies on a JavaBean Itembean to do the real work.

The ItemBean JavaBean encapsulates the Java code for accessing the back end resource to locate auction items. ItemBean has two functions: locate all items in a given category, and also add a new item to the auction.

In the case of browseitem.jsp, the Itembean is used only to view items up for auction. The capability to enter items is coded in different JSP pages. The ItemBean can be reused among multiple JSP pages in the WebAuction application. The ItemBean also relies on EJBs, which are accessed by standard Java calls. EJBs will be discussed in more details in subsequent sections.

To build ItemBean, declaration of package name and import of necessary classes are done:

```
Package Webauction.jsp;

import java.io.Writer;
import java.sql.Date;
import java.util.Calendar;
import java.util.Collection;
import java.util.Iterator;
```

the class and the constructor for this JavaBean:

```
public final class ItemBean {

  public ItemBean ( ) {

  }
```

5.9 Implicit objects

Implicit objects are objects automatically provided to JSPs by the JSP container. For example, user session information: The JSP container provides an implicit object that can be use with Java code to get and set the user session information. Other objects are available for servlet development.

Implicit object: out

Out is a subclass of the standard Java extension class javax.servlet. JspWriter, which enables information to be printed to the requester. Whenever results of an operation needs to be send back to the client browser, the out object, provided by the JSP container, is used.

Implicit object: request

Request is a subclass of javax.servlet.HttpServletRequest that includes all information provided by the request objects. The object is used to access the parameters and the respective values included in the request.

Implicit object: session

Session is an instance of the javax.servlet.http.HttpSession class. It represents the current session information for the user session. Every session is given a unique, randomly generated session identification number. The application server uses this number to track sessions. The server automatically handles the assignment of session ID numbers and transparently places session information in the client browser. This information is a place holder to match the browser to the HttpSession object that is automatically created for each user session.

Implicit object: exception

Exception is an instance of java.lan.Throwable that encapsulates the error message received if the page created is an error-handling page. Error-handling pages are those that include the page directive isErrorPage = true. To access the error messages, a number of methods are available:

Public String getMessage ( ) and public String getLocalizedMessage ( ), which return the message contained in the exception object and a localized version of the message, respectively. Localized messages are those intended for different languages. To print out the message of the exception to the user, the following code can be used in the JSP:

out.print ("exception:" + exception.getMessage ( ));

public void printStackTrace ( ) prints out a complete stack trace of the error. This is very helpful to see where the error originated and how it propagated through the system. By default, this method outputs the stack trace to the standard error output stream, which is logged in the application server log on most platforms.

6. Java Database Connectivity (JDBC)

SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database. Although SQL is both an ANSI and an ISO standard, many database products support SQL with proprietary extensions to the standard language[11]. Queries take the form of a command language that lets you select, insert, update, find out the location of data, and so forth. There is also a programming interface.

[11] Taylor (2002) JDBC: Database Programming with J2EE

JDBC is the J2EE standard for accessing an application's database resource. The JDBC standard specifies a Java API that enables you to write SQL statements that are then sent to the database.

JDBC is one of the oldest enterprise Java specifications: the earliest drafts date back to 1996. JDBC addresses the same problems as the Open Database Connectivity (ODBC) standard developed by Microsoft: to provide a universal set of APIs for accessing any database, using the database- specific driver. Without JDBC or ODBC, developers must use a different set of APIs to access each database: one for Oracle, one for Informix and so on. With JDBC or ODBC, a single set of APIS can access any database using the drivers specific to that database. Developers are able to write applications to a single set of APIs and then plug and play different database drivers, depending upon what resource type they are accessing. With JDBC, it is possible to migrate an enterprise Java Application from one database to another with only marginal adjustments, because no custom Java APIs are used.

While SQL is generally portable across multiple databases, the actual protocols that hose databases use to communicate (and some database- specific features) are not portable. For that reason, the JDBC specification supports products that map the calls in a JDBC-based application to the appropriate calls specific to the database. This is called a JDBC driver.

There are JDBC drivers specific to commercial databases such as Oracle, Sybase, and others. Accessing a database of a specific type requires the specific database driver for that database.

6.1 Database Connection Pooling

When an application server starts, it creates what are called connection pools to the database resource. Connection pools contain connections that are kept open to the

database resource by the application server. When the application needs to access the database, it grabs a connection from the connection pool and uses it to communicate with the database. Once the work being done with the database for a given user is completed, the database connection is released back to the database connection pool.

There are a number of reasons to pool connections to the database:

- Creating a new connection for every individual client that visits the site is very expensive. Using connection pools is much more efficient than creating a new database connection for each client, each time.

- Hard code details such as the database management system(DBMS) password in the application is not needed. This is particularly beneficial in the case of J2EE services such as JSPs, which typically store the source code with the application.

- The database system used can be changed with out changing the application code.

- Databases are most effective when the number of incoming connections is limited. With connection pooling, a limit on the number of connections to the DBMS can be place.

6.2 JDBC DataSource

To simplify the process for acquiring a connection to the database, the JDBC DataSource concept was introduced in the JDBC 2.0 specification. A DataSource object is a factory for Connection objects. To use a DataSource, one must specify a connection pool to provide connections to the DataSource in the Server's Java naming and Directory Interface (JNDI) (discuss below), which is a registry of user and application variables and values.

Using JDBC to Read Data

There are five basic operations for JSBC reads in the application server. These are:

- Establishing a connection to the database
- Sending a query to the database
- Getting results
- Handling results
- Releasing the connection

6.3 Establishing a Connection to the Database

Connections to the database are represented by instances of the java.sql.Connection object. Each instance of this object represents an individual database connection. Access of connections is done by calling factory methods on instances of the javax.sql.DataSource class.

The code snippet below demonstrates how a connection is made to a database:

```
Connection myConn =  null;

Try {

  Context ctx = new InitialContext ( )'

  javax.sql.DataSource ds = (javax.sql.DataSource)ctx.lookup("example-datasource-pool");
  java.sql.Connection myConn = ds.getConnection ( );

} catch (SQLException sqle)
```

{

}

The code first locates the application server JNDI naming services. Then, it creates a new instance of the java.sql.Connection class and assigns the object returned by ds.getConnection( ) to it. By calling getConnection with the appropriate DataSource as defined in the Server configuration, abstraction out is achieved from the database configuration.

There are other methods that are no as efficient and hence not chosen as the method to access the database the WebAuction application. For example, the DriverManager could be used to access the database. The problem with the DriverManager method is that it is a synchronized class, which means that only one thread of execution can run at a single time, where as the DataSource technique of accessing database is multi-threaded.

6.4 Sending a Query to the Database

The process of interacting with the database centers on the java.sql.Statment class. First, an instance of the Statement class by calling a factory method, createStatement ( ), on the instance of the Connection class that was created. The code snippet below demonstrates this:

Statement stmt = myConn.createStatement ( );

6.5 Executing SQL

The statement object can be used to execute SQL queries against the database. This is accomplished via the execute ( ) method. The result from the query to the database is handled by the java.sql.ResultSet datatype.

The statement class includes a method getResultSetType ( ), which returns query results as instances of the java.sql.ResultSet class. To get the result set for the query, the code snippet below is used.

ResultSet rs = stmt.getResultSet ( );

This creates an instance of the ResultSet class and returns the results of the query. A shortcut, single line method could also be used to for simple queries. This method is within the statement class: executeQuery (String SQL). For example:

ResultSet rs = stmt.executeQuery ("String SQLstring")

6.6 Accessing the Results

The ResultSet (rs) is a virtual table of data representing a database result set. The data returned form the query using the Statement class is encapsulated in an instance of the ResultSet class. Methods on the ResultSet is used to access the data. ResultSet is accessed very much like an Enumeration. A ResultSet object keeps a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next ( ) method moves the cursor to the next row. Because it returns false when there are no more rows in the ResultSet objects, it can be used in a WHILE loop to iterate through the result set.

ResultSet maps the data from the database to instances of Java objects. A relational-object mapping is required for object-oriented Java programs to be able to use relational data. Usually, results map directly from the SQL types that are defined by the database.

There are three basic operations for JDBC updates. These are:

- Established connections to the database

- Executing statements
- Releasing connections

The method establishing a connection to the database is the same as describe above.

Executing Statements

An instance of the Statement class I required to execute database updates. The createStatement ( ) method in the Connection class is used for SQL statement execution. The code snippet bellows demonstrates this:

Statement stmt = myConn.createStatement ( );

An instance of the Statement class is created for the specific connection to the database currently being used.

Three types of database updates are available in SQL:

- INSERT – to insert rows of data into tables
- UPDATE - to modify data in tables
- DELETE – to remove rows from tables

These statements are use via JDBC by inserting the SQL string in an executeUpdate ( ) method call on the instance of the Statement class. For example:

stmt.executeUpdate ("INSERT INTO mytable VALUES ('data1','data2')");

releasing connections

after updating the data in the database, release the connections and objects by running the close ( ) method on each of these objects:

stmt.close ( );
myConn.close ( );

6.7 Transactions

Transactions are a means to guarantee that a series of operations against a database completes accurately[12]. Using the WebAuction application as an example, transaction enables a user to place a bid on a given item available at auction. Transaction prevent multiple simultaneous users from winning the same auction, or from placing identical bids on a given item. Transactions represent a unit of work. Either all the work succeeds or noon, its all or nothing.

The transaction is not complete unless all of its operation are successful. The application server and database must ensure that transactions have the four essential properties known by the mnemonic "ACID". These prosperities are:

- Atomicity – the actions that make up the transaction must either all complete successfully, or none be executed at all.
- Consistency - a transaction must leave its environment and any data that it processes in a state that does not endanger integrity.
- Isolation – all of the actions in the transaction must result in the same values as if they were all run serially (one at a time).
- Durability - all results of the actions of the transaction must be persistently stored.

JDBC is used substantially in the WebAuction application. However, it is not used explicitly during development. Instead, as in the context of most commercial application, the JDBC access is encapsulated in an EJB. The WebAuction application uses Container Managed Persistence (CMP), which automatically generates JDBC code to map the relational data in the database to an object form.

[12] Perrone (2002) J2EE Developer's Handbook.

There is one instance in which explicit JDBC is used in the WebAuction application. It is necessary to use the database to keep a sequence for the WegAuction application to generate new ID numbers for auction items. An EJB is used to encapsulate this SQL.

7. Java Naming Directory Interface (JNDI)

A naming service is an integral piece of distributed systems. With the explosion of the Internet, naming services have become commonplace. While computers communicate with raw UIP addresses, humans prefer symbolic names that are easy to remember.

Like the Internet, distributed Java programs require a naming service to locate distributed objects. JNDI enables servers to host objects at specified names. Remote clients can perform a lookup in the JNDI service and receive a reference to the specified object.

The JNDI architecture consists of a common client interface and a set of JNDI providers that define the back-end naming system. JavaSoft defines an SPI interface through which new JNDI providers can be plugged into the JNDI system.

A JNDI client interacts with the JNDI system through the classes in the javax.naming package. The javax.naming.Context interface is the fundamental JNDI object. The Context interface has methods for a client to add, remove, and look up objects in the naming service[13].

The javax.naming.InitialContext class implements the Context interface. Clients use this class to interact with the JNDI system. Clients create an InitialContext object with the code snippet (as demonstrated in previous sections):

Context ctx = new InitialContext ( );

[13]Barish (2002) Building scalable and high-performance Java Web Applications using J2EE technology.

When the InitialContext method is created in the application server, the caller receives an InitialContext that reference the JNDI service on the local server. Remote clients can also create InitialContext references, but a client must let the JNDI client know the location of the application server.

With the InitialContext objects, the JNDI client can store objects in the naming system. Like RMI parameters, an object stored in JNDI must either implement java.io.Serializable or java.rmi.Remote. the bind method of the InitialContext object is used to establish a name to object mapping in the JNDI tree.

8. Enterprise JavaBeans (EJBs)

JavaSoft defined the Enterprise JavaBeans (EJB) specification to give Java Developers a foundation for building distributed business components. EJBs are Java components that implement business logic and follow a contract designated in the EJB specification[14]. EJBs live inside an EJB container, which provides a set of standard services, including transactions, persistence, security and concurrency. This means that the application programmer is freed from developing these services from scratch.

In EJB 2.0, as mentioned earlier there are four types of EJBs:

- Stateless session beans provide a service without storing a conversation state between method calls. The advantage of a stateless session bean is that a small number of instances can be used to satisfy a large number of customers. Each instance has no identity and is equivalent to any other instance.

[14] Matena (2003) Applying Enterprise JavaBeans 2.1: Compenet-Based Development for the J2EE Platform.

- Stateful session beans maintain state each instance is associated with a particular client.
- Entity bean represent an object view of persistent data, usually rows in a database. Entity beans have a primary key as a unique identifier. There are two operational styles for entity beans: container-managed persistence (CMP) and bean-managed persistence (BMP). In a CMP entity bean, the EJB container automatically generates code to persist the entity bean to a database. In a BMP entity bean, the bean writer must write the data access code. Generally, this involves writing JDBC code to insert, remove, and query the entity bean in the database. The advantage of BMP is it offers the bean writer complete flexibility about the entity bean's persistence. Because the bean writer is writing the data access code, almost any persistent store can be used. The main advantage of CMP is that it relieves the bean writer from having to write the data access code to persist the entity bean to a relational database. Instead of writing tedious JDBC code, CMP automates this process. In addition, EJB 2.0 CMP offers standard mapping for relationships between entity beans. This enables the container automatically to manage the interaction between business objects. Because the container has more control over data access in CMP, the performance of EJB 2.0 CMP beans is usually better than with BMP entities.

- Message-driven bean is the integration between JMS (Java Message Service) and EJBs, are used to perform asynchronous work within the server. Unlike other EJBs, clients never directly call a message-driven RJB. Instead, the client posts a message to a JMS destination. When a message arrives at the JMS destination. A MessageDrivenBean's onMessage method is called to process the message. Message-driven EJBs generally are used to perform asynchronous work within the server.

In an EJB, there are three main components:

- The remote interface

- The home interface
- The bean class

The remote interface lists the business methods that are available to clients of the EJB. Because this is an interface, the EJB writer does not implement these methods. The EJB container is responsible for supplying concrete implementation of the methods in the remote interface. The remote interface only stipulates the contract between the client and the EJB.

This is illustrated in the code snippet example below:

```
package com.example;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;
public interface ExampleEJB extends EJBObject {
        public String exampleEJB ( ) throws RemoteException;
}
```

The above remote interface is used to expose business logic to the client. In this case, the EJB offers a single exampleEJB ( ) method.

The home interface is the EJB's factory. Clients use the home interface to create, find and remove instances of an EJB. Like the remote interface, the EJB writer only defines the signature of the methods in the home interface. Below is an example of this:

```
Package com.example;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
```

```
public class ExampleEJBHome extends EJBHome {
        public ExampleEJB create ( ) throws CreateException, RemoteException;
}
```

The ExampleEJBHome interface contains a single create method. This create methods is a factory that produces reference to the Example EJB. The return type is the Example interface. The return type of create methods is always the EJB's remote interface.

The bean class implements the business logic that is exposed to the client through the remote interface. For instances, the bean class must implement the ExampleEJB business method:

```
package com.example;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class ExampleEJBBean implements SessionBean {
        private SessionContext ctx;
        public void setSessionContext(SessionContext c) {
        ctx = c
        }
        public String ExampleEJB ( ) {
        return " Place your message here";
        }
        public void ejbCreate ( ) {}
        public void ejbRemove( ) {}
        public void ejbActivate( ) {}
        public void ejbPassivate ( ) {}

}
```

the remaining methods in the ExampleEJB class (except for ejb Create ( )) are inherited from the javax.ejb.SessionBean interface.

In order for clients to make use of the ExampleEJB, it must be deploy it into the server. First, the ExampleEJB is compiled to produce the class files. The next step is to create the deployment descriptors. The when the EJB container deploys the EJB, it reads configuration parameters and metadata from the deployment descriptor. For instance, the container uses the deployment descriptor to determine what type of EJB is being deployed, the name of the home interface, and other vital information.

The standard deployment descriptor begins by declaring its XML document type. EJB deployment descriptors are XML documents that must use the structure defined in the standard EJB DTD (document type descriptor). The descriptor gives the EJB container the names of the home interface, remote interface, and the EJB (bean) class. The ejb-name parameter is a logical name for this EJB. It is used throughout the deployment descriptor to refer tot his EJB. The XML also includes a session-type parameter. This informs the container that this deployment is a stateless session EJB.

In addition to the standard deployment descriptor, the WebLogic Server also requires a WebLogic-specific deployment descriptor. This deployment descriptor enables the EJB writer to configure parameters that are specific to the WebLogic implementation. The deployment descriptor contains only two pieces of information. First, it uses the ejb-name tag to specify that these parameters are for each of the EJB in the application. The second tag is the JNDI name, the EJB container binds the home interface into the JNDI tree using the <jndi-name> specified in the server deployment descriptor. Clients can then find RJB by using the JNDI to look up the EJB.

Entity EJBs present an object view of persistent data. The fields in entity beans correspond to underlying data in a persistent store - usually, a relational database. An entity bean’s state is transactional. When a client updates field within a transaction, the

updates are only permanent if the transaction commits. When a transaction rolls back, the entity bean's state returns to its last committed state.

In a multi-tier e-commerce application, back end persistence is provided by one or more databases. The Web engine uses HTML for static content, and servlets and JSPs for dynamic presentation logic. EJBs provide the business logic between Web tier and the database.

Session beans can take advantage of container services such as transactions, security and concurrency. However, session beans cannot directly represent persistent data. Java is an object-oriented language, but databases store data relationally, as rows in tables. Session beans using JDBC cannot easily represent data as first-class objects. Moreover, session bean do not share some of the defining characteristics of persistent data: multiple clients do not share them, and they do not generally survive server reboots or crashes.

The EJB specification provides entity beans as a persistent, transactional and shared component, so that business data can be simultaneously used by many clients and persistently stored until it has been explicitly deleted.

Entity beans consist of a home interface, remote interface, bean class, primary key class, and deployment descriptors.

8.1 Home Interface

The home interface extends the javax.ejb.EJBHome interfaces and contains create methods, remove methods, finder methods, and home methods.

An entity bean's create methods calls the corresponding ejbCreate method on the bean class. The responsibility of the create method is to create the persistent representation in the backing store. The is usually implemented as a database insert.

The entity bean's home interface must define a remove method that takes a primary key as a parameter. This method removes the entity bean instance with the corresponding primary key from the persistent store. Usually this represents a database delete operation.

8.2 Primary keys and identities

Entity beans have identities, a business method in the remote interface must be called against a specific entity bean instance. The entity bean client receives the entity bean reference by creating, finding, or using an EJB handle. A bean either has identity (it has a unique identifier such as a primary key) or it is anonymous (no primary key has been attached).

Each entity bean reference is associated with a particular primary key. When calls are made against that reference, they are dispatched to a bean instance with the same primary key.

8.3 Primary key classes

All entity beans must include a primary key class. The primary key class identifies the entity bean instance: its value must be unique for the entity bean type. The primary key class can be either a java primitive type such a java.lang.String or java.lang.Integer, other user may write a custom primary key class. The primary key class maps to one of more fields in the entity bean. A primary key with multiple fields is known as a compound primary key.

8.4 Finder Methods

Finder methods enable the client to make queries and receive references to entity beans that satisfy query conditions. Every entity EJB must have a findByPrimaryKey method in its home interface. This special finder method returns an EJB reference that has the corresponding primary key. Bean writers may also define more complex finders that return many entity reference that match the finder's condition.

8.5 Home Methods

Entity beans also can have home methods. Home methods are business methods that do not apply to a particular instance. Instead, the container merely chooses an available instance and calls the home method on it.

The Bean Class and Bean Context

Like session beans, the entity bean's remote interface extends the java.ejb.EJBObject interface and contains the signatures for business methods. The actual implementation of these methods is provided in the bean class.

The entity bean's implementation class implements the javax.ejb.EntityBean interface. Like the javax.ejb.SessionBean interface, the EntityBean interface contains the signatures for callbacks from the EJB container to the bean instance. The setEntityContext method instance is called immediately after the bean's constructor and passes the bean instance the EntityContext. The EntityContext is generally stored in a member variable and is used by the bean instance to make some standard calls into the EJB container. The setEntityContext method may be used to acquire some basic resources such as DataSource references that are not specific to a particular primary key.

When setEntityContext is called, the EJB container has not yet assigned a primary key to this bean instance. The entity bean interface also has a corresponding unsetEntityContext method that is called before the bean instance is destroyed.

The WebAuction application utilizes the CMP entity bean for persistent storage of data (see following sections for more detail). CMP entity bean is preferred over BMP in the application as CMP offers many advantage over BMP. Instead of writing cumbersome JDBC code, the CMP bean writer provides only the business logic and deployment descriptors. CMP can offer faster development time and better performance than BMP entity beans. The CMP entity bean class is abstract. This enables the EJB container to implement persistence logic by generating a class that extends the bean class.

8.6 Container-Managed Fields

Every container-managed entity bean has a set of container-managed fields, which are saved and loaded from the database. Generally, each container-managed field corresponds to a column in a relational database.

The bean provider cannot declare container-managed fields. Instead, the bean writer declares abstract get and set methods for each container-managed field. For instance, instead of declaring a private String name in the bean class, the bean provider uses public abstract void setName (String name); and public abstract String getName ( );. These get and set methods are public abstract because the EJB container provides the actual implementation. This enables the EJB container to detect when fields are read and written. This enables the EJB container to optimize the calls to the database.

Each container-managed field must be declared in the ejb-jar.xml deployment descriptor. This enables the container to match the container-managed fields with the set and get methods in the bean class. The bean provider then includes the database mapping in a separate CMP deployment descriptor name WebLogic-cmp-rdbms.xml, which contains the database table name and a mapping between each container-managed field and its corresponding database column.

A CMP entity bean must set the values of the primary key fields in its ejbCreate methods. Then the ejbCreate methods always returns null. The EJB container determines the primary key value by extracting the primary key fields after the ejbCreate has returned. The bean needs to se the primary key fields in ejbCreate because the container does the database insert after it calls ejbCreate.

The entity beans used in the WebAuction application use CMP so do not include JDBC code: the EJB container generates code that provides automatic persistence. The EJB container calls the ejbCreate methods before the bean has been inserted into a database. The ejbCreate method generally uses its parameters to initialize the entity bean's fields.

Entity beans exist in a persistent store until they are deleted by either the entity bean's remove method or by a direct database delete. Therefore, the entity lifecycle must accommodate instances that exist before the EJB server is started and that continue to live after the EJB server is halted.

Entity bean instances exist in tow states: anonymous and identity. An anonymous entity bean is similar to a stateless session bean. It has no associated identity: one anonymous instance is as good as any other. An identified bean has an associated primary key that uniquely identifies the instance. Through its lifetime, the entity bean transitions between these states in response to callbacks from the EJB container.

8.7 Message-Driven EJBs

Enterprise Messaging with the Java Message Service's messaging service allows an asynchronous model. Instead of waiting for the server's response, the client sends a message to a JMS destination and returns. In addition to scalability benefits, this model enables client programs to continue without waiting for server operations to complete.

EJBs can be used to send messages, but session and entity beans cannot be used as JMS message listeners. The problem is that session and entity beans instances live within the EJB container, and the EJB container determines their lifecycle. With the EJB 2.0 specification, there is a new EJB type that integrates EJB and JMS. This new EJB type is the message-driven EJB.

Message-driven EJBs are the integration between EJB and the JMS. Like other EJB types, message-driven EJBs live within an EJB container and benefits from EJB container services such as transactions, security, and concurrency control. However, a message-driven EJB does not interact directly with clients. Instead, messages-driven EJBs are JMS message listeners. A client publishes messages to a JMS destination. The JMS provider and the EJB container then cooperate to deliver the message to the message-driven EJB

Because message-driven EJBs do not have clients, they do not require home or remote interfaces. A message-driven EJB is a bean class that implements the javax.ejb.messageDrivenBean and the javax.jms.MessageListener interfaces. The messageDrivenBean interface includes only two methods: setMessageDrivenContext and ejbRemove. The MessageListener interface contains only a single method, onMessage. In addition to implementing these three methods, the bean writer provides a single ejbCreate methods with no parameters. One of the best features of message-driven EJBs is their simplicity: this single bean class has only four methods. Message-driven beans include a single class, which implements the javax.ejb.MessageDrivenBean and javax.jms.MesssageListener interface.

9. WebAuction Design

9.1 Presentation logic

The WebAuction application is an e-commerce application that uses nearly all of the J2EE APIs: It uses JavaServer Pages (JSP), Java Database Connectivity (JDBC), Java Message Service (JMS), Java Naming and Directory Interface (JNDI), Enterprise JavaBeans(EJB), and JavaMail.

9.1.1 WebAuction Subsystems

The WebAuction has two layers: a presentation layer and a business logic layer. Clients use Web browsers to interact with the presentation layer. The presentation layer is responsible for relaying client request to the business logic layer and rendering the business logic's responses into HTML.

The business logic layer handles the application's logic and communication with back-end systems such as databases. The database, which provides a persistent repository for the application's data, usually resides on a separate server. The presentation layer is implemented with web components including JSP pages and tag libraries.

The business logic of the WebAuction system uses EJB components, JMS, JavaMail, and JDBC. The WebAuction's entity beans use container-managed persistence(CMP), the EJB container handles the data access code.

9.1.2 Interfaces

One of the aim of the design of the WebAuction system is to ensure good interfaces between components. Interfaces promote information hiding, which enables software components to minimize exposure of internal information to other areas of the system.

This in turn minimizes dependencies, which allows components may be redesigned to re-implemented without affecting other parts of the system.

9.1.3 Separating the presentation and business logic

A clear separation between the presentation layer and the business logic tier is important. The design goals of the presentation layer's user interface are clean and clear web pages[15], and providing a user-friendly website.

An advantage for separating the presentation and business logic is to allow for parallel development. This enables development teams to work parallel to each other with the web page designer and experts to focus on the presentation layer, allowing business logic developers to concentrate on the database, messaging and transaction issues.

Separating presentation logic can also allow security features to be implemented at the presentation layer. This could be done by ensuring privileged access to first access a form-based login page. This enables the application to change security policies without modifying the back-end logic. The WebAuction application uses JSP to access its presentation logic. JSP pages can include Java code therefore possible to access business logic directly from the JSP page. However, this design structure is not used; in fact the JSP used in WebAuction contains little Java code. Instead the WebAuction system uses JSP with tag libraries or small snippets of code to access the business logic layers. This design would enable the web page layout to be altered by graphic designers easily without affecting the remainder of the system. Another advantage of using tag libraries is their syntax resembles other HTML documents. This is beneficial for Web page designers who are more familiar with HTML than Java code. Tag libraries structure facilitates reusability of existing tags in other application components, which would be require more effort for in the case of JavaBeans and servlets.

[15] Crawford (2002) Java Enterprise in a nutshell

The WebAuction's presentation layer also uses JavaBeans as value objects when interfacing with the business logic. JavaBeans are simple objects with get and set methods for each field. The value objects pass information such as the new account profile to the business logic layer. The business logic layer also uses JavaBeans to return information to the web tier. The web tier renders pages by retrieving the information hel din the value objects. However, the downside to this approach is that extra objects must be created to encapsulate information that already exists within the business object layer. For instance, when the JSP page needs to show the current item available for bid, the WebAuction application must find the appropriate items and then create a value object holding the information for each item. While this creates additional objects, the advantages of this approach is that the presentation layer sees only the value objects, sot he persistence(explain what this means) layer can be changed without affecting JSP pages or the tag libraries.

9.1.4 Tag Library-to-Business Logic Interface

The WebAuction's presentation layer interfaces with the business logic layer by means of JavaBean value objects. The tag libraries contain Java code to access the business logic layer. In order to decouple persistence, transactions, and messaging (the business layer) from the presentation layer, entry points into the business logic layer needed to be minimized. This was achieved by tag library using either the bids JMS queue or the WebAuction stateless session bean to communicate with the business logic layer.

Synchronous vs. Asynchronous design

When a WebAuction's customer submits a bid, the bid information is encapsulated in a JMS message and sent to a JMS queue. Once the information is on the queue, the JSP page displays a message stating the bid has been received by the system. This is asynchronous design: the bidder need not wait for the system to process the bid. It is only necessary to wait for the bid's information to be entered in the bid. It is only necessary to wait of the bid's information to be entered in the persistent queue. A JMS

listener handles the actual bid processing in the background. Synchronous design where bidder waits until the bid processing completes is possible, however, as every bids requires to be validated, and the WebAuction application updates several database tables when a bid it entered. This might introduce significant delays.

The advantage of a asynchronous approach aids scalability, for example, a server cluster can listen on the queue and do all of the processing in parallel.

The user's bids are entered in to the system from bid.jsp with the enter-bid tag:

```
<Webauction:enterBid itemId= "<% = bidId %>"
  username = "<%= request.getRemoteUser ( )%>"
  bidamount = "<%= amount %>"
/>
```

The EnterBid. Java tag library sends a message to the JMS queue with the values passed in the tag's parameters:

```
bidMsg.setInt("item_ID", itemId);
bidMsg.setString("User_Name", username);
bidMsg.setDouble("Amount", bidAmount);

qsender.send(bidMsg);
```

in addition to the bids JMS queue, WebAuction's business logic layer exports a synchronous interface to presentation layer with the WebAuction stateless session bean. Because the WebAuction bean is stateless, the tag libraries create a reference when they are initialized and make all business method calls against the reference, as holding a stateless session bean reference does not tie up resources in the server. Whenever a call is made against the WebAuction remote interface, the EJB container selects a pooled

bean instance to handle the call. If a stateful session or entity bean were used, the tag library would have to include more lifecycle code to manage the EJB state.

The WebAuction bean's remote interface exposes utility methods that are used by the tag libraries. The methods return a BidValueHolder or the ItemValueHolder JavaBeans.

```
Public interface Webauction extends EJBObject {

  Bid ValueHolder [] getBidsForUser(String username)
    Throws NoSuchUserException, RemoteException;

ItemValueHolder getItemWithId(int id)
  Throws NoSuchItemException, RemoteException;

ItemValueHolder [] getItemsInCategory(String category)
  Throws RemoteException;

}
```

the GetBidsForUser tag library keeps a reference to the WebAuction stateless session bean in the WebAuction member variable. The getBidsForUser method call returns an array of BidValueHolder objects, which are then assigned to the bids variable in the JSP page.

```
BidValueHolder [] bids = Webauction.getBidsForUser(userName);
PageContext.setAttribute("bids",bids);
```

The currentbid.jsp page uses the GetBidsFOrUser tag library to receive the array of BidValueHolder objects:

Finally, the currentbid.jsp renders the bids into the HTML page. Each bid is shown as a row in a table. Note the use of the wl:repeat tag. This is a standard tag included in the WebLogic server's tag library. It iterates through the elements of an array or java.util.Collection and applies the tag body to each element. In this case, it iterates through the bids array and assigns each value in the array to bid variable.

9.1.5 Security

The WebAuction security model is based on user accounts. Users register with the WebAuction site and create user accounts with password and an associated email address. Protection of web pages are not always necessary as in the case of general browsing by web users which should be allowed without having the user logging in or creating an account. This can minimize resources as no session tracking is required until the user has logged in.

All security checking is done at the presentation layer as the business logic layer assumes that presentation layer has satisfied all security constraints. This is a compromised between security and simplicity. In addition, security checks can affect performance, so avoiding unnecessary access constraints helps scalability[16].

Security constraints in the Deployment Descriptor

The web.xml deployment descriptor specifies the security constraints that restrict access to a protected page. The server would redirect the browser to the login page if a user has not logged in before accepting the request.

The security constraint restricts the newitem.jsp page to users in the auction_user role.

```
<security-contraint>
  <web-resource-collection>
    <web-resource-name>New Item</web-resource-name>
```

[16] Berg (2003) Designing Secure J2EE Application and Web Services

```
    <url-pattern>/newitem.jsp</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>auction_user</role-name>
  </auth-constraint>
</security-constraint>
```

the web.xml file also declares the abstract role named auction_user:

```
<security-role>
  <role-name>auction_user</role-name>
</security_role>
```

The WebLogic.xml maps the auction_user role name to the user principal.

```
<security-role-assignment>
  <role-name>auction_user</role-name>
  <principal-name>user</principal-name>
</security-role-assignment>
```

All WebAuction users is a member of the group named user and can access the newuser.jsp page.

The WebAuction application uses the servlet 2.2 specification's form-based authentication to certify user names and passwords. The login.jsp page includes a form with the j_security_check as the action. The user name is passed as j_username and the password is j_password. The password field uses the input type of password to prevent the password characters from being echoed to the screen. The login.jsp page is served through HTTPS to ensure that the password is encrypted on the network.

9.1.6 Creating New User Accounts

New user accounts are created by the newuser.jsp Web page. This page includes a FORM element that contains all of the required account information. WebAuction requires that all fields in the new user form be completed, by including a JavaScript onSubmit element that instructs the browser to run validateNewUserForm before accepting the submission. The JavaScript ensures that every field has a value.

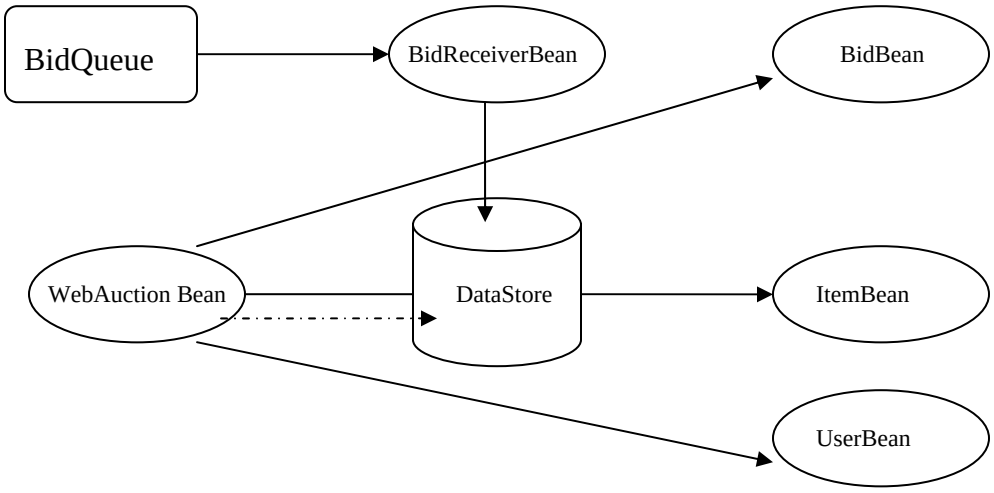
```
<form method = "post" name = "NewUser" action = " newuser.jsp"
        onSubmit = "return validateNewUserForm( )">

<SCRIPT LANGUAGE = "JavaScript">
  <! – Hide code from non-javascript
 function validateNewUserForm ( ) {
        newUserForm = document.NewUser;
        if ( (newUserForm.userName.value = = "") ||
            (newUserForm.password.value = = "") ||
            ...and the rest of the user account fields…
            (newUserForm.email.value = = "")  {
        alert ( " You must fill out all fields to create a Webauction account.");
        return false;
} else {
        return true;
}
}
</SCRIPT>
```

9.2 Business Design

The business logic layer implements the application logic as well as maintaining the transactional integrity and persistence of the application data. The presentation layer contains no code for managing transaction or accessing the database, as this would violate modularity and limit scalability.

The business logic layer is entered either via the bids JMS queue or via the WebAuction stateless session bean. The BidReceiverBean is a message driven bean that listens on bids JMS queue. When a new bid arrives, BidReceiverBean receives the message and updates the persistent representation. Both the BidReceiverBean and the WebAuction stateless session bean must update the persistent state in the database. WebAuction stores its persistent state in tree entity beans: the UserBean, BidBean, and ItemBean.



The UserBean stores account information that is entered when the account is created. This information should exist until the account is removed from the system. This persistent information is stored in the database and modeled with an entity bean.

The ItemBean represents a WebAuction item that is available for auction. The item information is populated when the item is entered into the auction, and it must be persistent.

The BidBean is the persistent representation of a user's bid. When the BidReceiverBean de-queues the bid message, it validates the message and creates a new BidBean entity bean with the corresponding bid information.

9.2.1 WebAuction Stateless Session Bean

The WebAuction stateless session bean is the synchronous interface into the business logic layer for the entity beans and their persistent data. The WebAuction bean is responsible for reading the persistent information and populating the value objects. This design strategy was chosen over another design strategy which was to return the entity beans to the presentation layer. This design is rejected because it exposes the persistent layer to the presentation layer, this would make it increasingly difficult to make changes as the application grows. Creating value objects also helps performance, as entity beans are transactional objects, and their state is refreshed from the database on transaction boundaries. If the presentation layer accessed the returned entity beans in another transaction, it would cause another round of database hits to refresh the returned information.

The WebAuctionBean's getBidsForUser method returns all the bids for a given user name. this method uses the BidBean'sfinder method to return the appropriate bid reference. It then iterates through each bid and creates a matching BidValueHolder.

```
public BidValueHolder [] getBidsForUser (String userName)
throws NoSuchUserException
{
  try {
    User user = userHome.findByPrimaryKey( userName);
    Collection bids = bidHome.findBidsForUser( user);
    Int size = bids.size( );
    If(size  = = 0) {
```

```
      // no bids for this user
   return null;
  } else {
      //build an array of java bean value objects
   Iterator it = bids.iterator ( );
   BidValueHolder [] bidValues = new BidValueHolder [ size];

   For ( int i = 0; i< size; i++) {
    Bid b = (Bid) narrow(it.next( ), Bid.class);
    bidValues[i] = new
     BidValueHolder(b.getItem( ).getDescription ( ), b.getAmount  ( ) );
   }

  return bidValues;
}
```

9.2.2 Transaction Flow

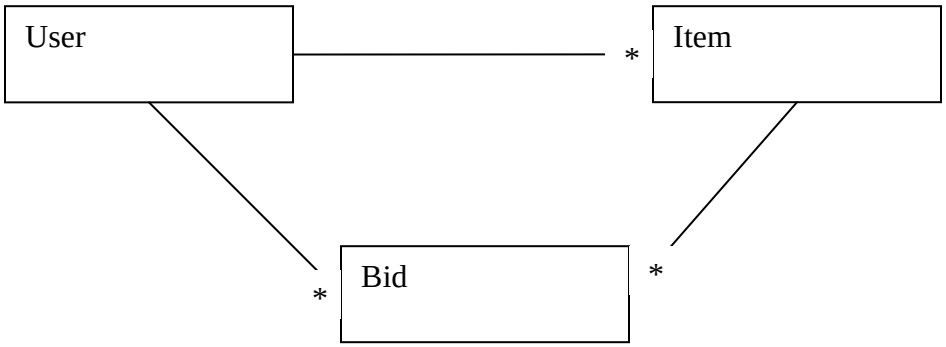
All transaction demarcation in the WebAuction application is handled in the business logic layer, in container-demarcated transactions of the EJBs. The WebAuction stateless session bean is deployed with the Required transaction attribute to ensure that the container starts a transaction before running the business methods. This minimize database access frequency, this could be demonstrated using the getBidsForUser example above. Here the CMP engine does a single database query to retrieve the associated bids. The calls to the BidBeans do not incur database hits because the CMP engine has already prefetched the associated data in this transaction. If these methods ran in a separate transaction, each bid.getAmount () call would be another trip to the database.

The BidReceiver message-driven bean also demarcates transactions when it receives a message from the bids queue. The BidReceiverBean is deployed with the Required

transaction attribute. The transaction starts with the JMS message receipt, and it flows into the entity beans called by BidReceiverBean to process the new bean. If the transaction aborts, any entity bean updates are rolled back, and the message is returned to the JMS queue. This ensures that each bid message is processed and committed, at most, once. Also should the server goes down while processing a new bid, the bid is not lost.

9.2.3 Entity Bean Relationships

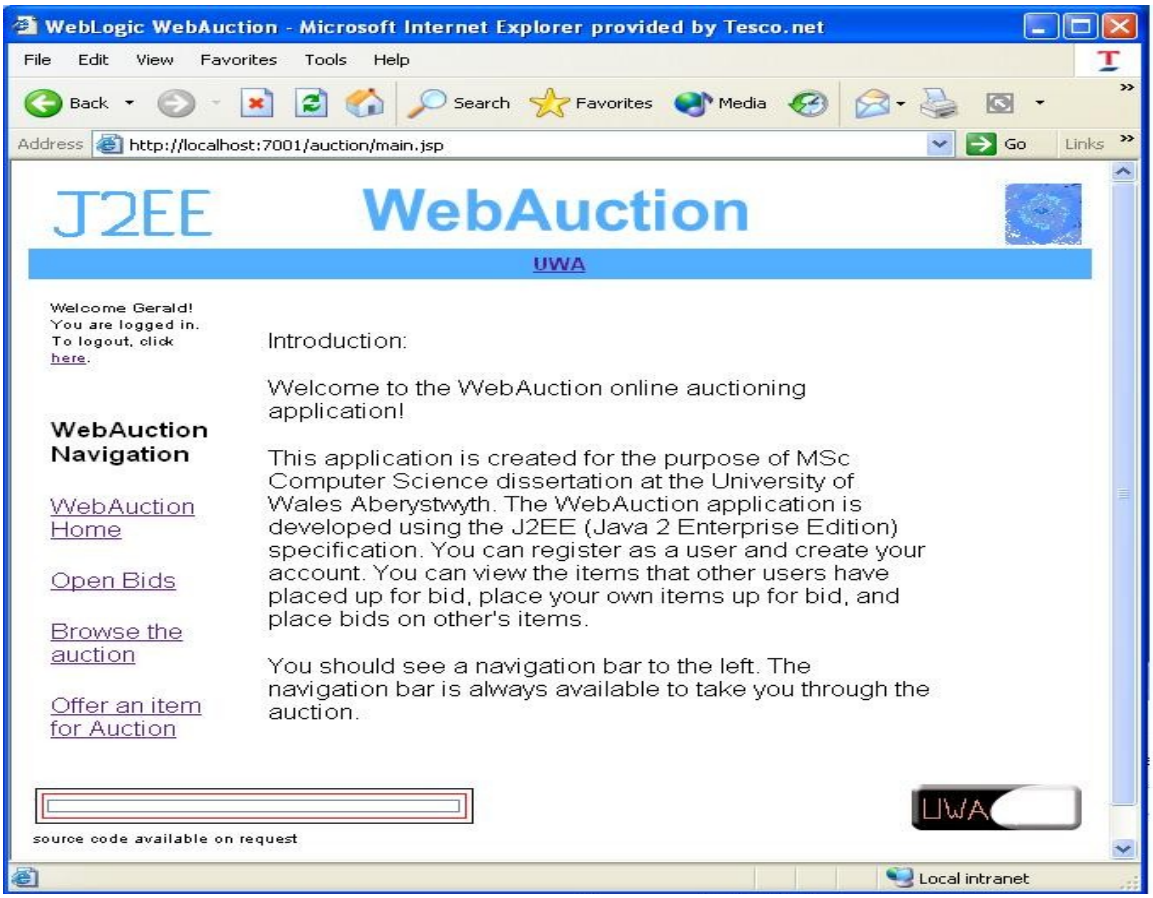
The WebAuction application maintains relationships among its persistent data. There is a 1:N (one to many) relationship between a user and his bids; a 1:N relationship between user and his items; and a 1:N relationship between and item and its bids.



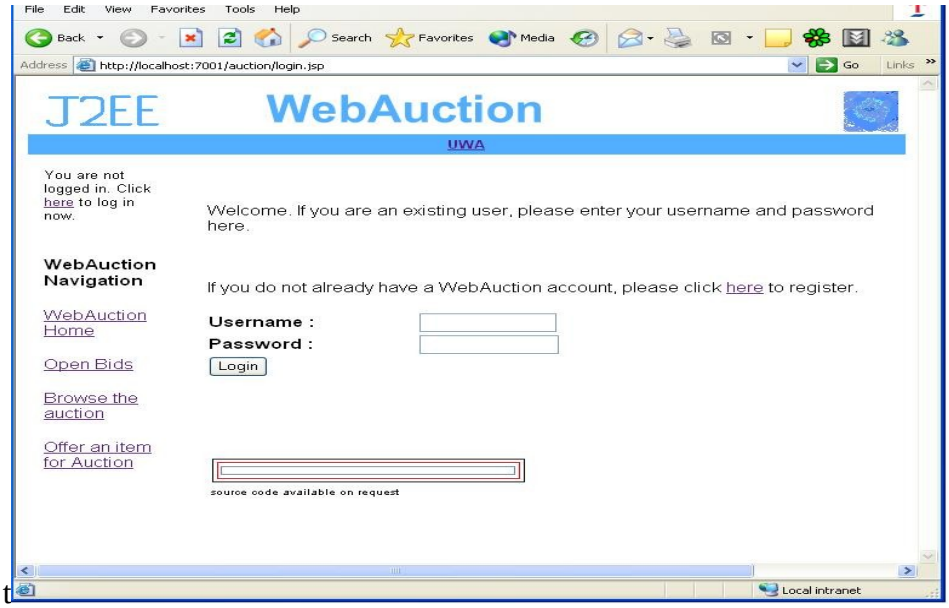
10. The WebAuction Application Demonstration

The following shows a demonstration of the WebAuction Application.

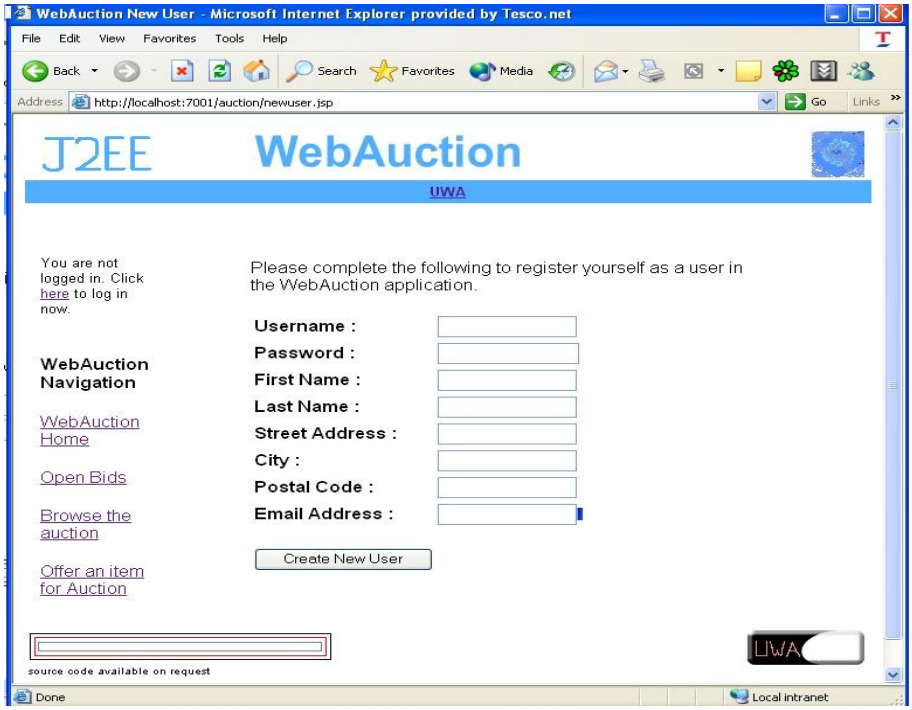
The WebAuction Home Page:



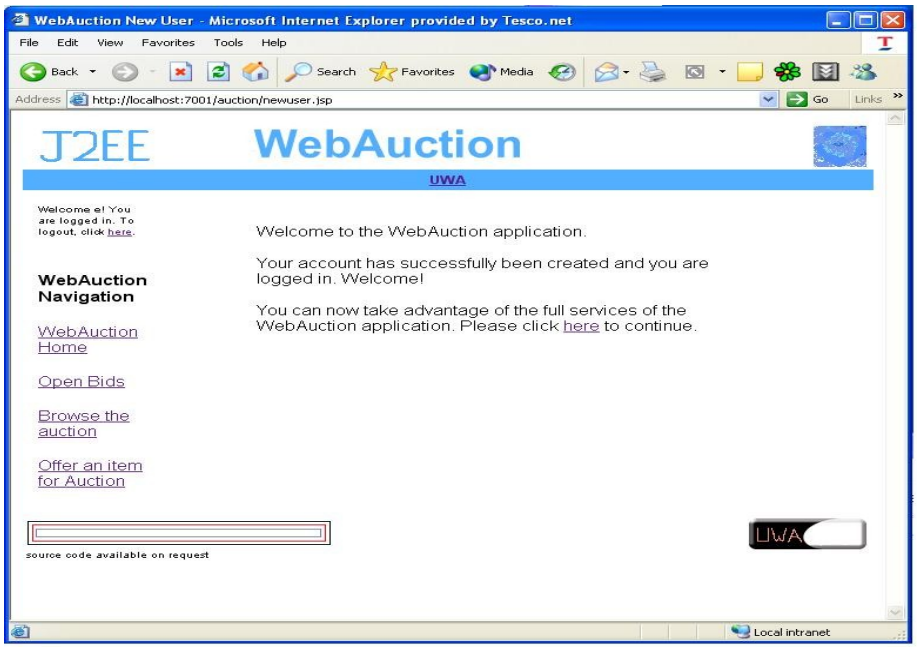
The user can log on to the application or register if no account exists:



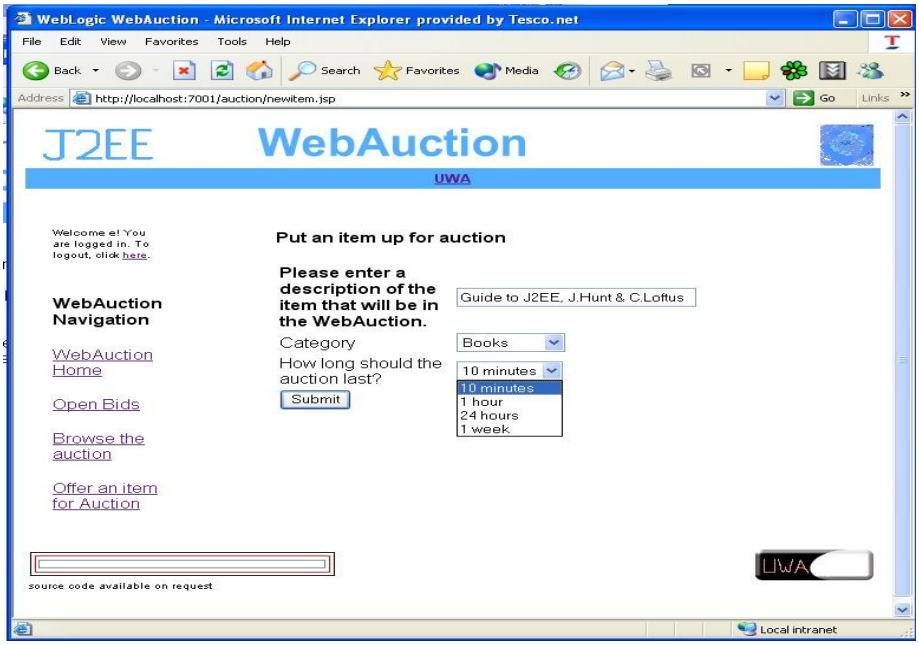
This page enables the user to create a new user account by entering the appropriate details:



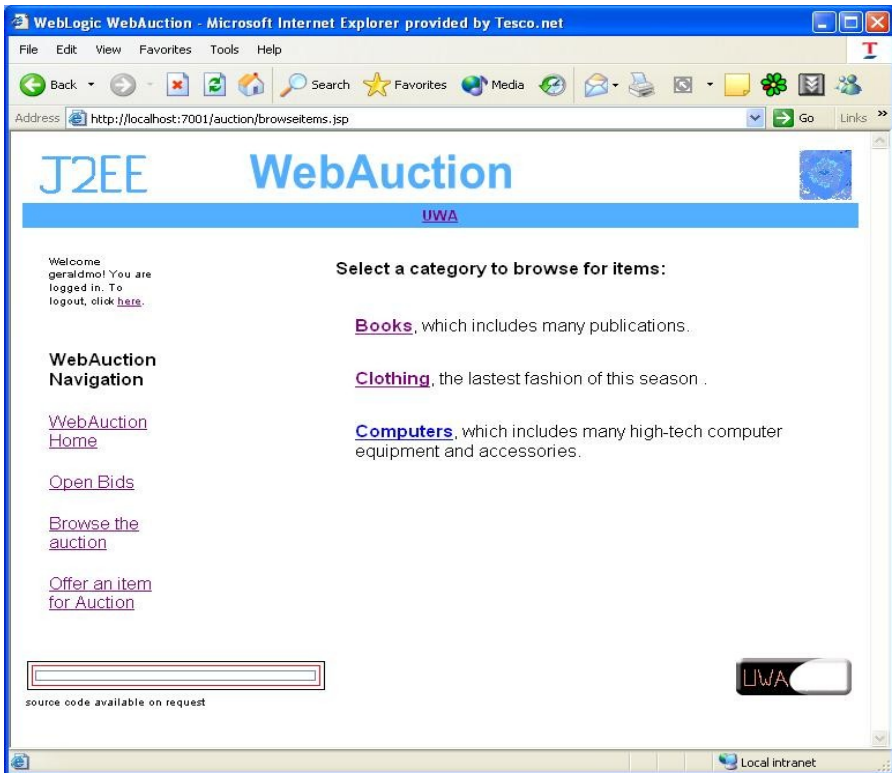
After clicking on the “Create New User” button, the WebAuction application confirms that a new user account has been create:



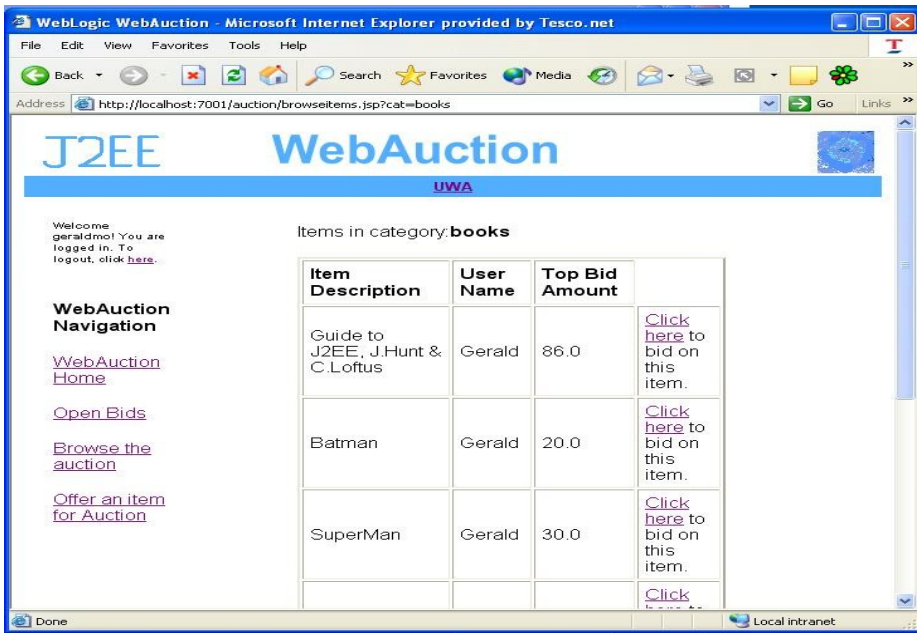
Having register, the user can offer items up for auction:



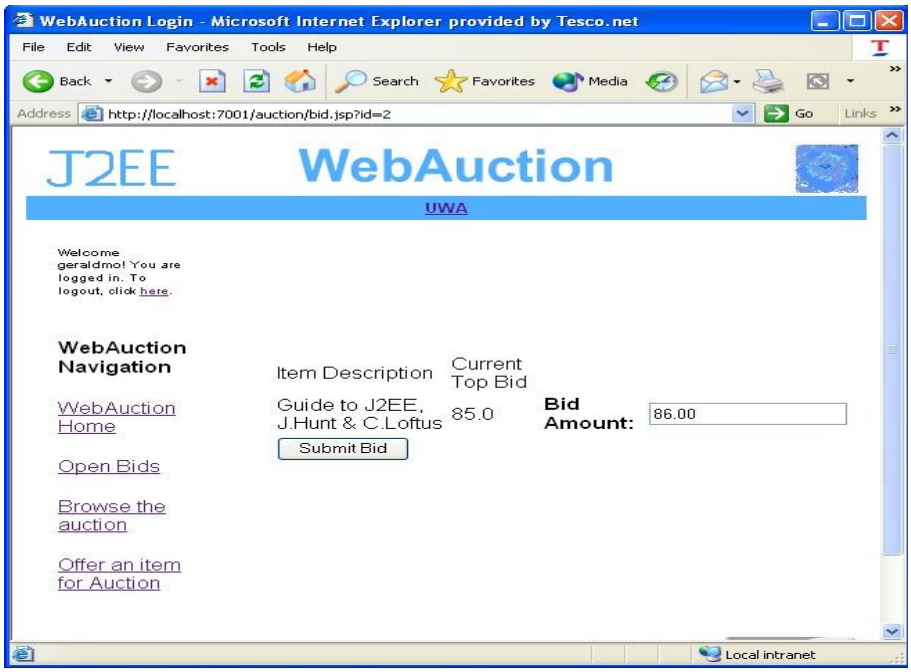
The user can browse the auctions that are up for auction:



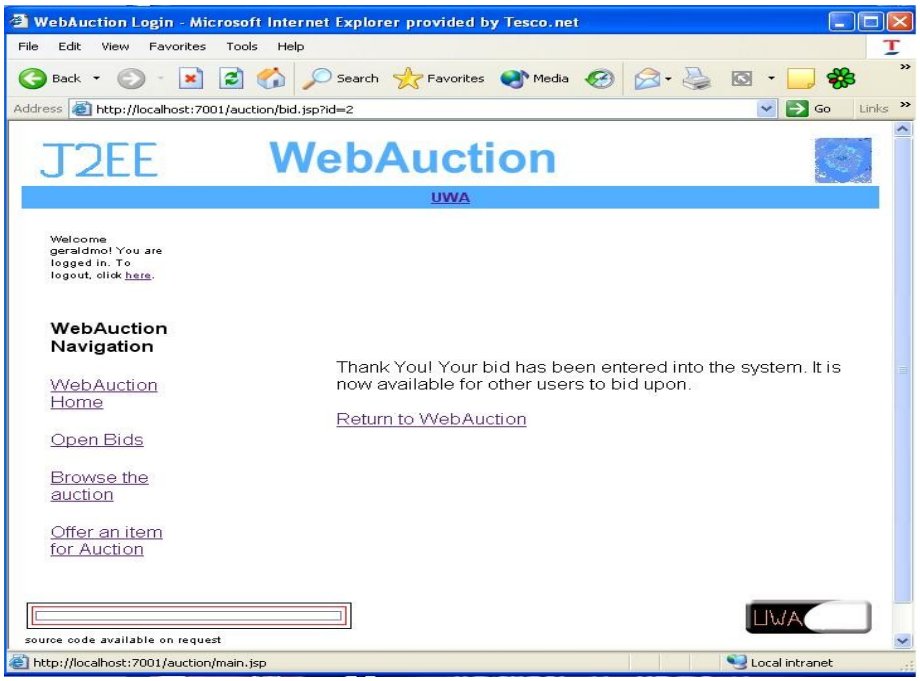
The user chooses the item to bid on:



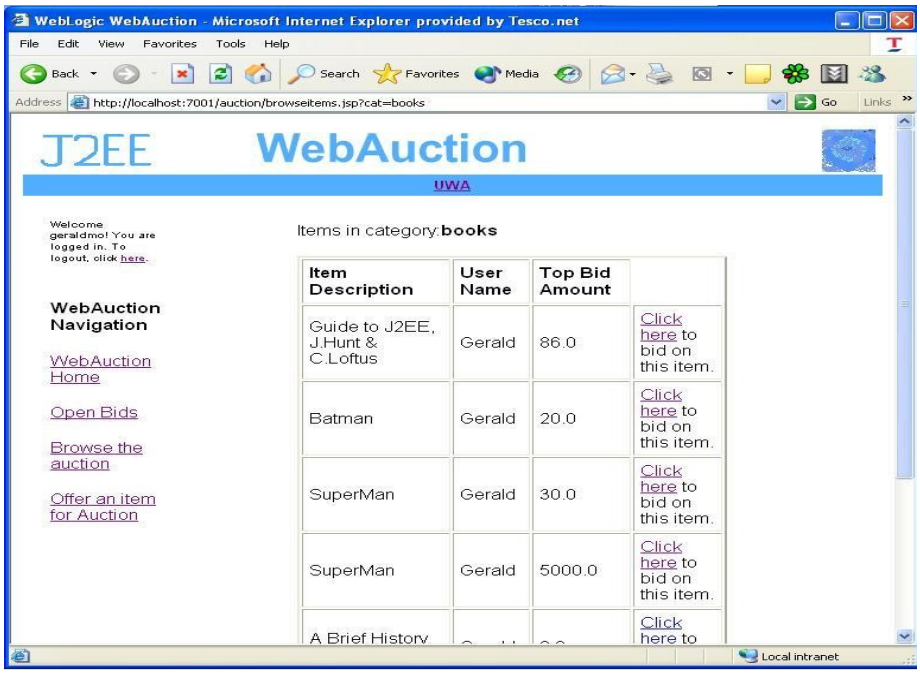
The user can place a bid on the item:



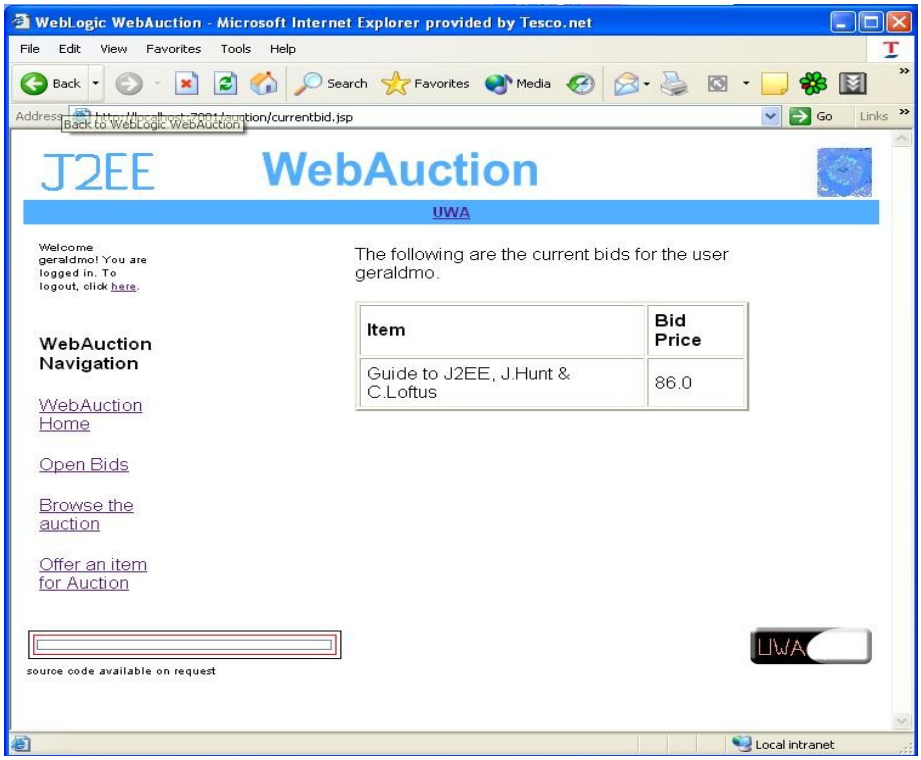
The WebAuction application confirms that a bid has been placed:



To confirm that the Top Bid Amount field has been updated for the item by browsing the appropriate category:



The user can check their bidding history on all items:



11. Testing

There were two types of testing carried out on the WebAuction application to ensure that it runs as expected.

- Functional testing, which ensures that the application functions as expected
- Stress and performance testing, which ensures that the application performs as expected under heavy and real world conditions. Note that Stress and performance testing is not carried out as it is un-realistic to perform the test on a non-server class hardware. The idea behind stress testing is discussed below.

11.1 Functional Testing

The tests are carried out manually over a web browser. The tests are design to ensure each functional requirements (FR) are met.

FR 1. User registration and user log-in:

Test Description	Pass/Fail	Comments
User is able to enter User account details:	Pass	Although no checks on made to ensure that email address format appears to be valid or not
User is unable to create an account using user name that already exist	Pass	An error message is displayed informing the user.
User needs to ensure all fields are fill out before submitting the form	Pass	An error message is displayed informing the user.
User is able to log into the WebAuction application (using existing user account)	Pass	A greeting message personalized to the user.
Logged in user can logout	Pass	

FR 2. Email validation of user's login credentials

An email is sent to User's email address	Fail	Mailsession server in Web Server configuration is required
Information in confirmation email is correct	Fail	Mailsession server in Web Server configuration is required

FR 3. Browsing, for both registered and unregistered users

Unregistered user is able to browse all category	Pass	Unregistered user is able to browse all available category
Registered user is able to browse all category	Pass	

FR 4. Browsing by category for auction items

All users can browse all available category	Pass	
If category is empty, a information message is displayed	Pass	An message enabling the user to following a link to another category
All items placed submitted into a category is displayed accordingly	Pass	

FR 5. Placing bids

Users not logged in cannot place bid on item	Pass	User is forwarded to a user log-in page
Logged in users can place bids on items	Pass	User is forwarded to a page to enter bid
Bids placed cannot be negative number	Pass	Error message informs user that
Bids placed should conform the format of XX.XX	Fail	Values of XX.XXX is possible

Bids placed value should be more than the current top bid	Fail	However, the top bid amount remains accurate.
A higher bid would over write the current top bid	Pass	

FR 6. Email confirmation bids

Email is sent to confirm that a bid has been placed	Fail	Mailsession server in Web Server configuration is required
Email contains correct information of the bid	Fail	Mailsession server in Web Server configuration is required

FR 7. Viewing open bids

Logged in user can view all bids places history	Pass	A list of the current bids for the user is displayed
Users not logged in cannot view open bids	Pass	User is forwarded to the Log-in JSP

11.2 Stress and Performance Testing

The functional testing was used to assure that the WebAuction application performed as expected. It is also important to make sure the application performs as expected in high-stress situation. As there can be many users connecting to the application during real deployment simultaneously, the deployment must scale to handle the increased demand. To verify this, stress and performance testing attempts to replicate the load that will be placed on the application.

There are four major steps in stress and performance testing with the Application Server Web applications:

Define scope of testing. Definition is required on how the users are expected to interact with the application. The purpose of this step is to develop an estimation of how users

will interact with the application in a real-world situation. For example, a simplified interaction model for the WebAuction application could be:
95% of all interactions will either involve no database access (using only main.jsp) or simply will browse the items up for auction
5% of all interactions will involve updates to the database (using newuser.jsp for example)

Design tests. After the definition of the scope of the testing, the representation of the interaction model should be designed. For example, in WebAuction, test should address the features listed previously:

- Simulate many users browsing items simultaneously
- Simulate many users registering for the site simultaneously
- Simulate both user interactions by performing both reads and updates in a single test run.

Implement and execute test. The execution parameters of the test should be designed to locate possible bottlenecks or limitations in the architecture of the application and deployment.
For example, test should be executed and monitored to determine whether an application waste CPU cycles, requires special WebLogic Server tuning parameters, requires database tuning etc. Limiting factors, such as CPU usage for example should be isolated one at a time in testing your application.

The following are some factors that could be tested one at a time:

- CPU utilization
- Memory utilization
- Server Execute
- Database utilization and connectivity

Review results and repeat. After the tests have been executed, a feedback process that builds on results from the tests that were run. In the previous step, tests located possible

bottlenecks or limitations in the architecture and design. In this step, the limitations identified should be analyzed in order to improve the performance of the application.

12. Limitation

The following section highlights some limitations of the WebAuction application and gives suggestions for improvement.

The WebAuction application uses simple entity bean finder methods to query the database. This may require further refinement for a commercial application. For instance, the ItemBean includes a finder that returns all the items in a given category. On a real auction site this could return hundreds or thousands of rows, this would be too many to display on a Web page, and it is expensive to return this many items in a result set from the database. A limit in the number of results returned by a finder could resolve this problem. A possible solution to this is by including an additional condition in the finder clause that limits the query to a range of IDs. So the finder could initially return items in with ID ranges between 100 to 200. The web page would have a next button to enable the user to ask for more results. The next page could run the query again asking for the next batch of IDs.

Many common business objects, including WebAuction's Items and Bids, have no natural "primary key" that would uniquely identify them in the database. Instead, a new primary key value is generated for every new item and bid. The only requirement is that primary key values be unique within the items or bids. WebAuction uses a separate IDGenerator stateless session bean to produce these unique IDs. Before an item or bid is created, the caller uses the IDGenerator's getNextValue ( ) method to receive a new ID. The item or bid is then created with the generated ID.

The advantages of using a stateless session bean for unique ID generation is that the algorithm can be modified without affecting other code. WebAuction uses a database

sequence value to generate unique IDs. This scheme takes advantage of the database's existing ID generation support, however, this involves an extra database round trip on each create to read the next sequence value.

The WebAuction application JSP forms should provide field checking for user inputs, for example, checking should be done to ensure that the format of email address is valid, or that the bid amount place cannot be beyond 3 decimal places.

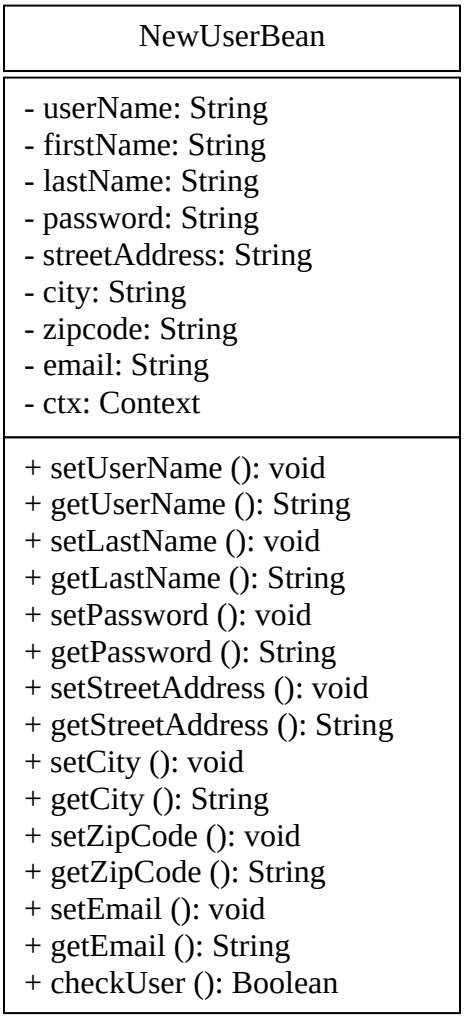
Potentially, WebAuction application could support multiple languages. WebAuction could facilitate internationalization by separating the presentation JSP pates from the presentation logic in JSP tag libraries. The WebAuction application could be internationalized by moving the printed messages from the pages in to separate message catalogs for each language. A user entering the internationalized WebAuction site could select a language, and the preference would be noted in the servlet session. The JSP pages would be reformatted to check the session for a language preference and access the appropriate message catalog.

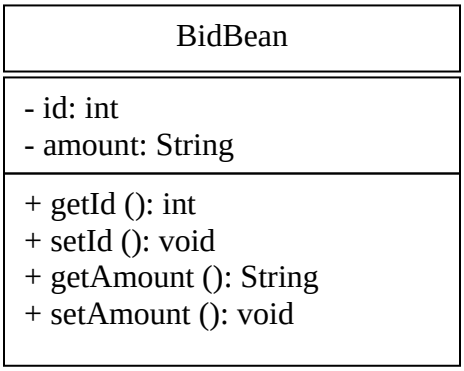
13. Bibliography

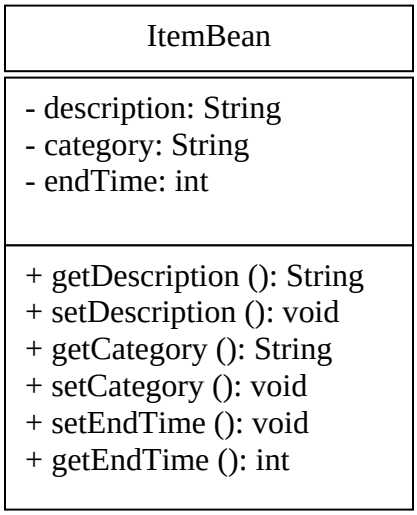
1. Allamaraju, Subrahmanyam. (2001) *Professional Java server programming J2EE 1.3 edition.* Wrox Press Inc.
2. Alur, D., Crupi, J., and Malks, D. (2001) *Core J2EE patterns: best practices and design strategies.* Prentice Hall PTR.
3. Arlow, J., Neustadt, I. (2001) *UML and the Unified Process: Practical Object-Oriented Analysis and Design.* Addison Wesley Professional.
4. Asbury, S. and Weiner, S. (2001) *Developing Java enterprises applications.* J. Wiley, 2001
5. Barish, G. (2002) *Building scalable and high-performance Java Web applications using J2EE technology.* Addison-Wesley.
6. Berg, C. (2003) *Designing Secure J2EE Application and Web Services*. Prentice Hall PTR.
7. Bergsten. H. (2004) *JavaServer Pages.* O'Reilly.
8. Booch, G., Jacobson, I., Rumbaugh, J. (1998). *Unified Modeling Language User Guide (Object Technology).* Addison Wesley Professional.
9. Broemmer, D. (2002) *J2EE Best Practices: Java Design Patters, Automation, and Performance*. Wiley.
10. Chiu. (2004) *Mastering Bea WebLogic Server: the Complete Reference Guide: The Complete Reference Guide.* John Wiley & Sons Inc
11. Crawford, W. and Flanagan. D. (2002) *Java Enterprise in a nutshell: a desktop quick reference.* O'Reilly.
12. Crawford, W. and Kaplan, J. (2003) *J2EE design patterns*. O'Reilly
13. Falkner, J., Jones, K. (2003) *Servlets and JSP: The J2EE Web Tier.* Addison-Wesley Pub Co.
14. Hammell, T., Gold, R., Snyder, T. (2004) *Test-DrivenDevelopment: A J2EE Example.*
15. Hanna, P. (2003) *JSP 2.0: the complete reference.* McGraw-Hill/Osborne.
16. Heaton, J. (2003) *BEA WebLogic Server for Dummies.* John Wiley & Sons Inc
17. Horton, I. (2001) *Beginning Java 2*. Wrox Press Inc.
18. Hunt, J. and Loftus, C. (2003) *Guide to J2EE: enterprise Java*. Springer.

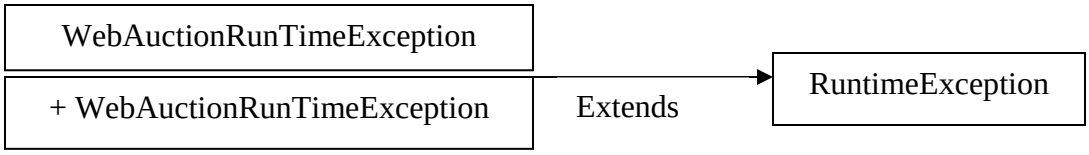
19. Johnson, R., Hoeller, J. (2004) *Expert One-on-One J2EE Development without EJB.* Wrox Press Inc.
20. Kulak, D., Guiney, E. (2003) *Use Cases: Requirements in Context (2nd Edition).* Addison-Wesley Pub Co.
21. Marinescu, F. (2002) *EJB design patterns: advanced patterns, processes, and idioms.* John Wiley.
22. Matena, V., Krishnan, S., Demichiel, L., Stearns, B. (2003) *Applying Enterprise JavaBeans 2.1: Compenet-Based Development for the J2EE Platform (2nd Edition).* Addision-Wesley Pub Co.
23. Oberg, R. (2001) *Mastering RMI: developing Enterprise applications in Java and EJB.* Wiley.
24. Perrone, O., Venkata, S., Schwenk, T. (2003) *J2EE Developer's Handbook.* SAMS
25. Saganich, A., Kaye, L., Hardy, T., (2004) *Bea WebLogic Workshop 8.1 Kick Start: Simplifying Java Web Applications and J2ee.* Sams.
26. Taylor, A., (2002) *JDBC: Database Programming with J2EE.* Pearson Education.
27. Weaver, J., Mukhar, K., Crume, J. (2004) *Beginning J2EE 1.4 From Novice to Professional (Apress Beginner Series).* APRS
28. Winder, R. and Roberts, G. Developing Java software
29. Zuffoletto, J. (2003) *BEA WebLogic Server Bible*. John Wiley & Sons Inc

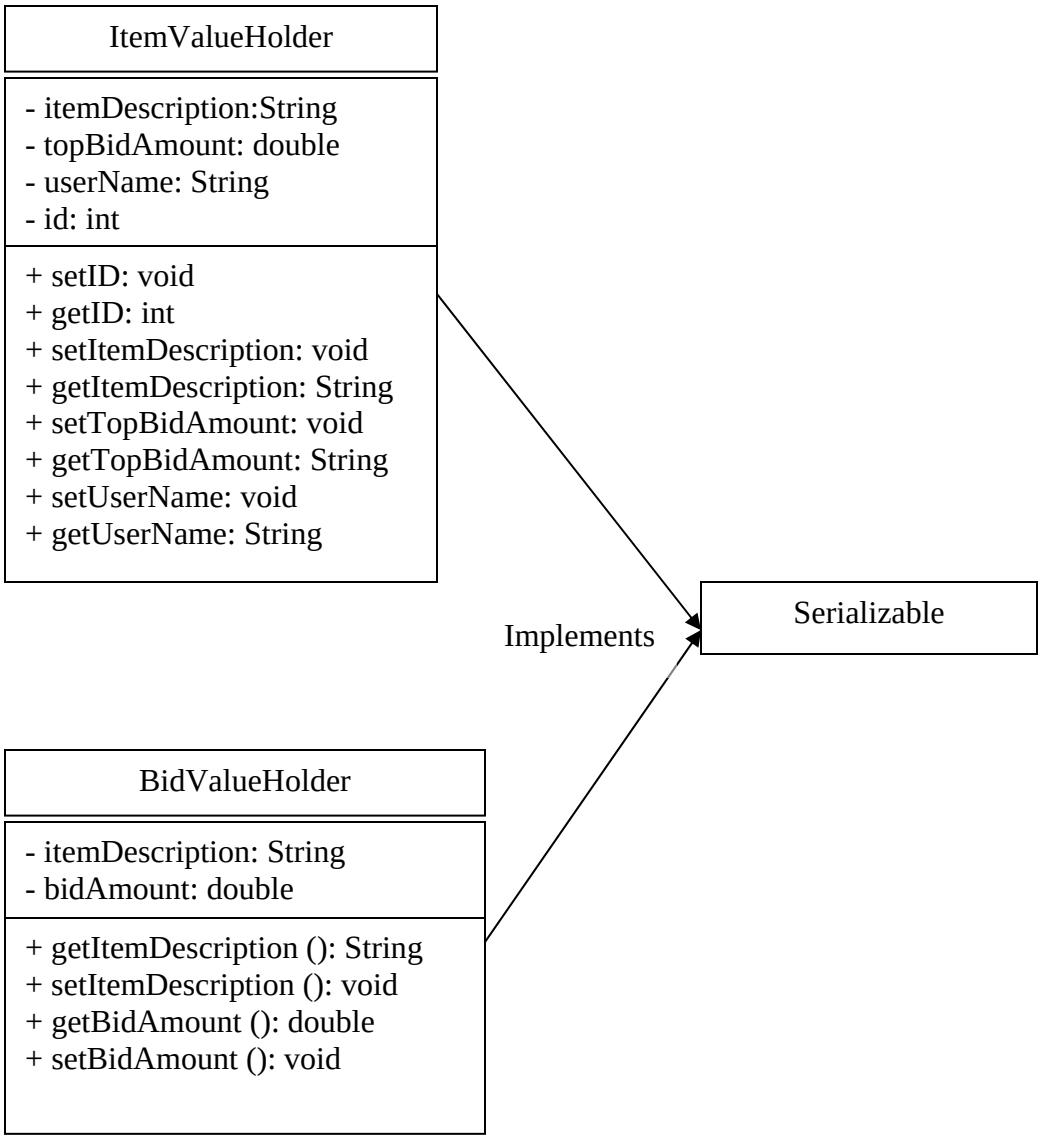
14 Appendix

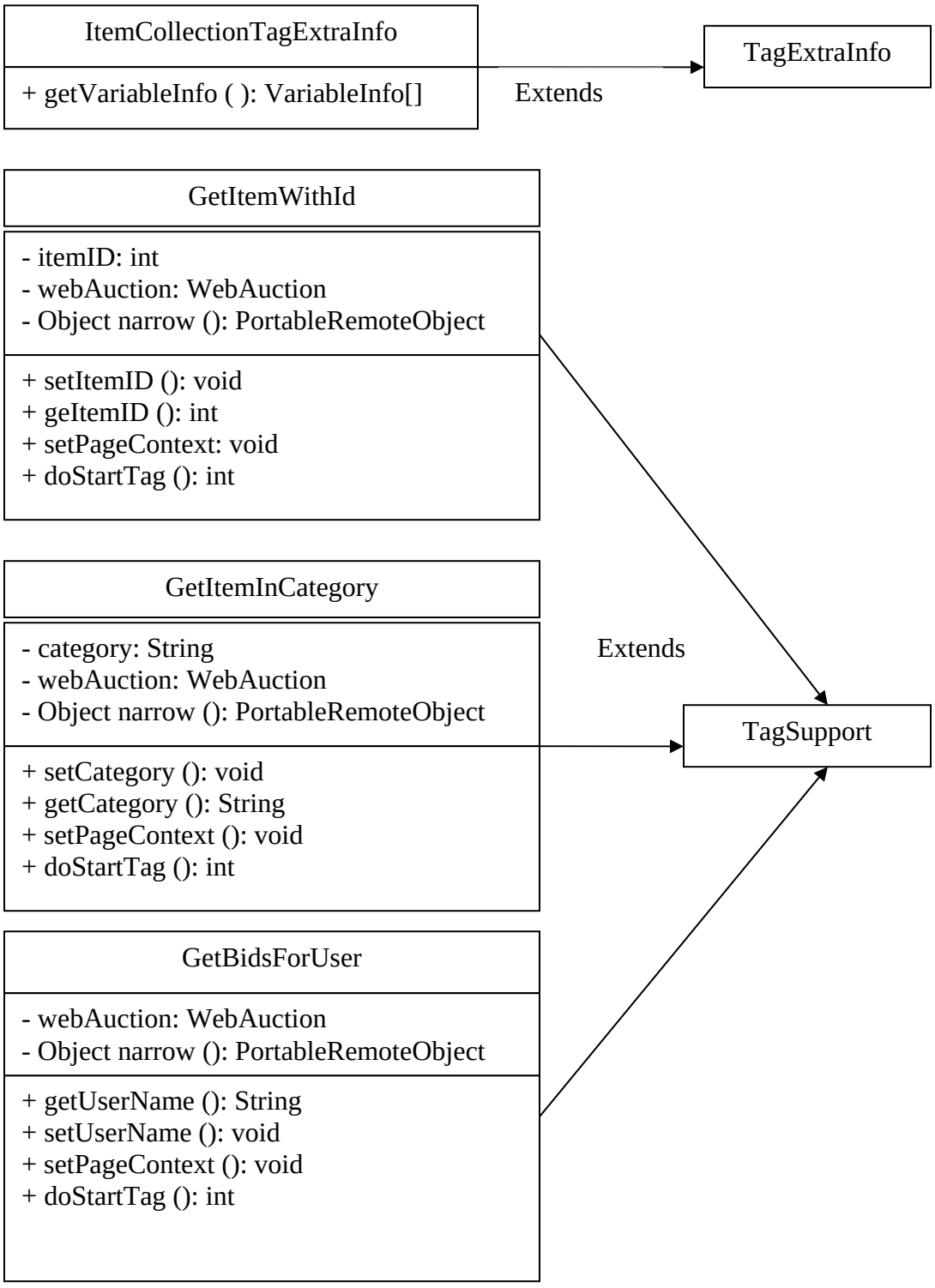

NewUserBean
- userName: String
- firstName: String
- lastName: String
- password: String
- streetAddress: String
- city: String
- zipcode: String
- email: String
- ctx: Context
+ setUserName (): void
+ getUserName (): String
+ setLastName (): void
+ getLastName (): String
+ setPassword (): void
+ getPassword (): String
+ setStreetAddress (): void
+ getStreetAddress (): String
+ setCity (): void
+ getCity (): String
+ setZipCode (): void
+ getZipCode (): String
+ setEmail (): void
+ getEmail (): String
+ checkUser (): Boolean

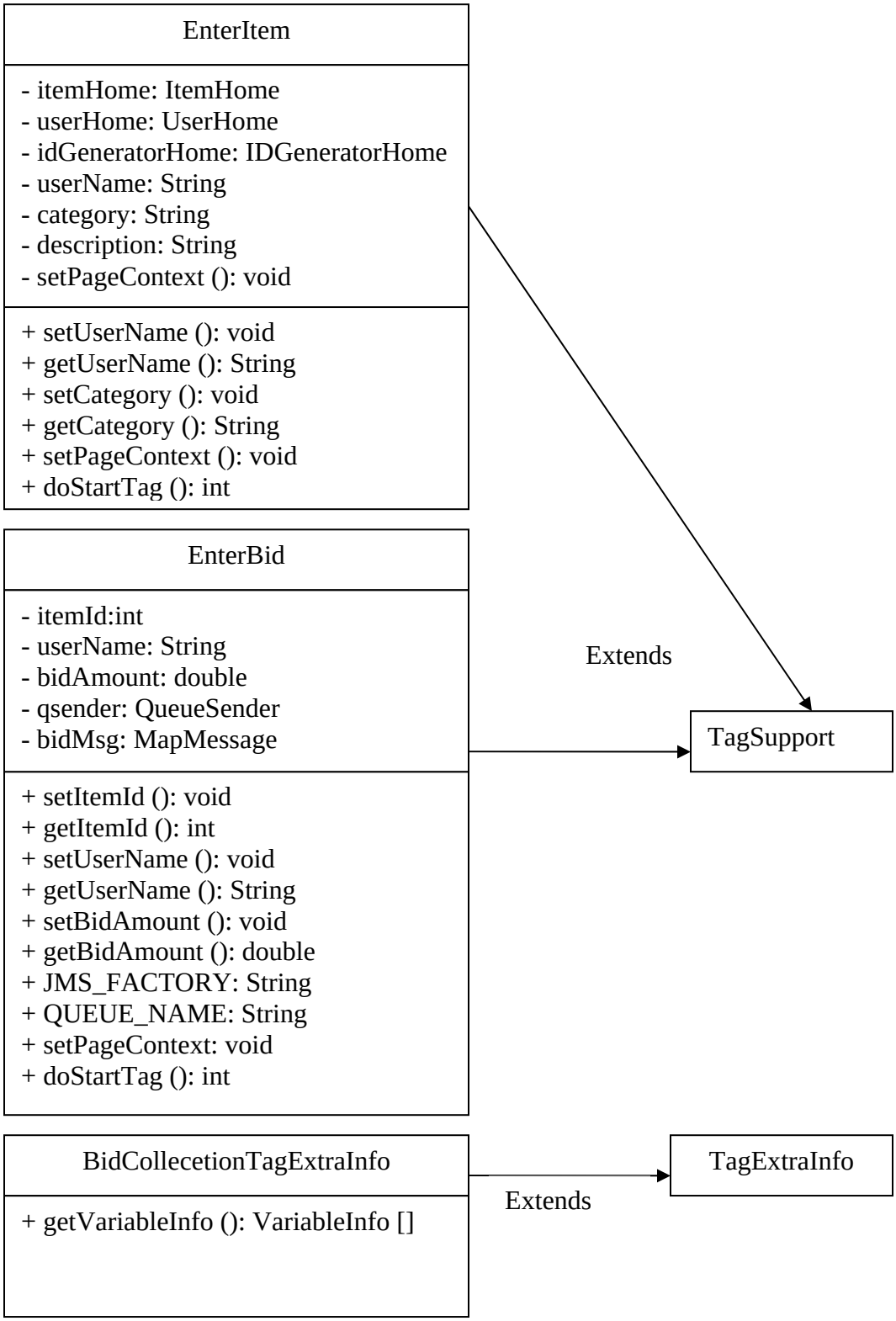

BidBean
- id: int
- amount: String
+ getId (): int
+ setId (): void
+ getAmount (): String
+ setAmount (): void

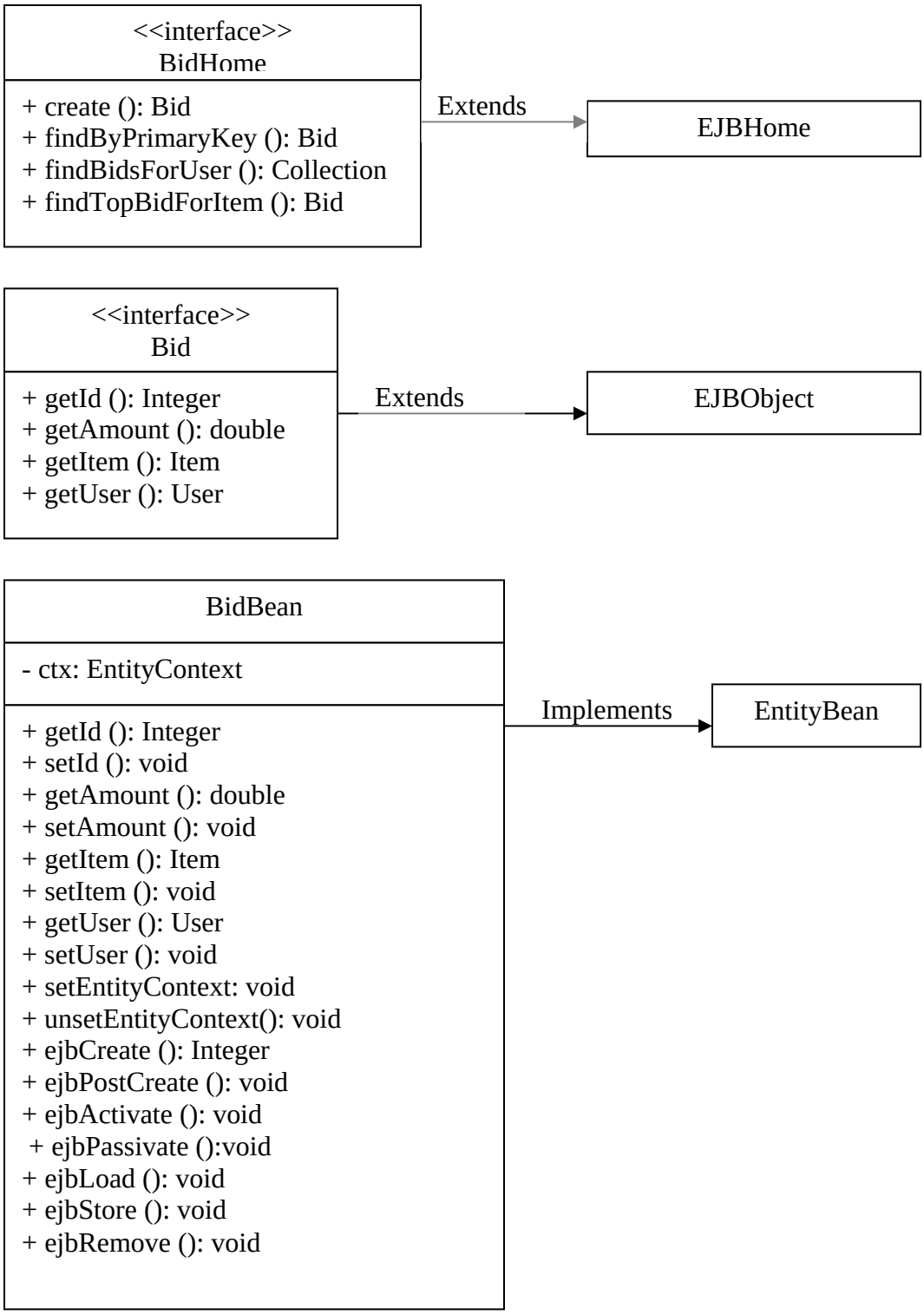

ItemBean
- description: String
- category: String
- endTime: int
+ getDescription (): String
+ setDescription (): void
+ getCategory (): String
+ setCategory (): void
+ setEndTime (): void
+ getEndTime (): int

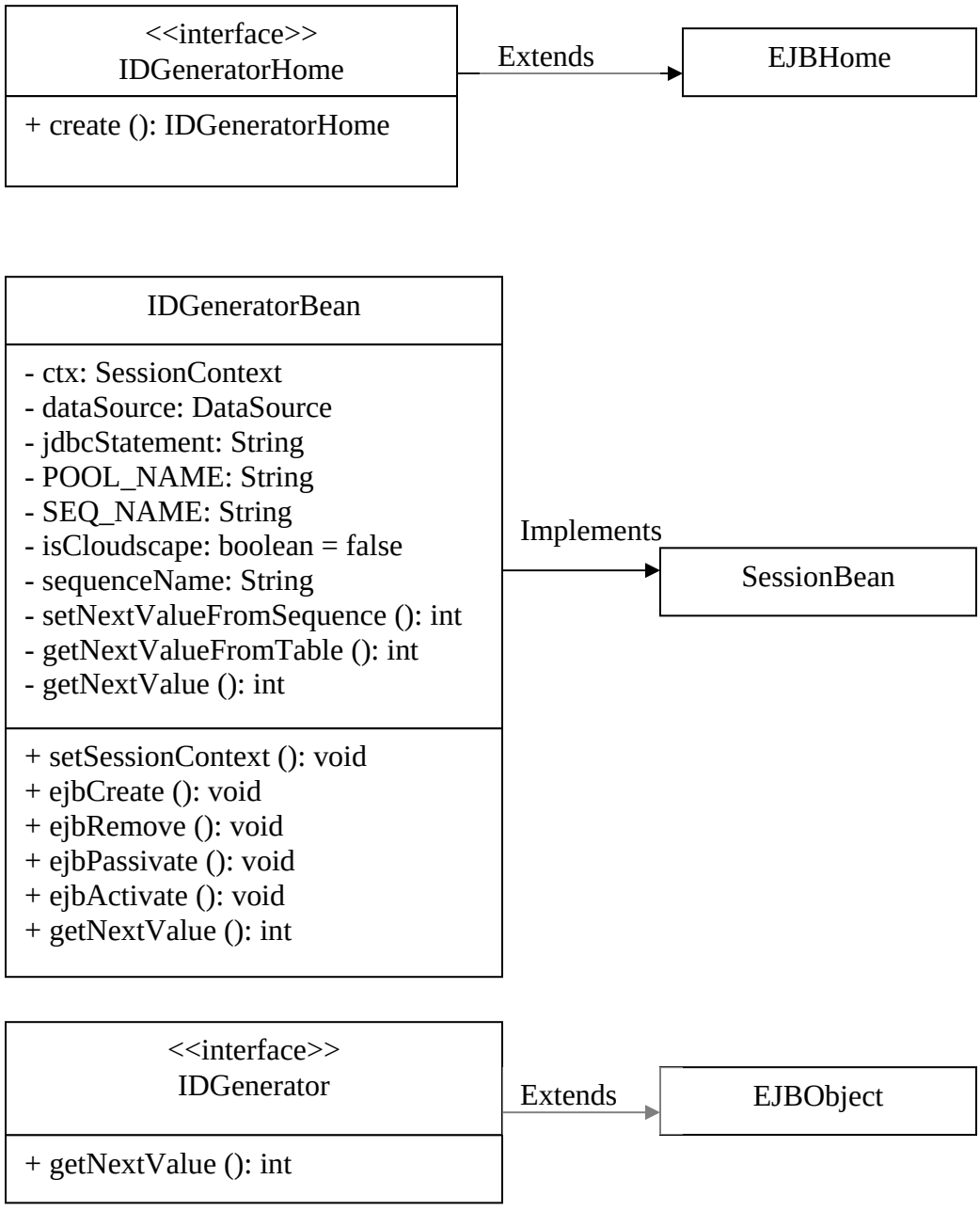

WebAuctionRunTimeException
+ WebAuctionRunTimeException
Extends
RuntimeException

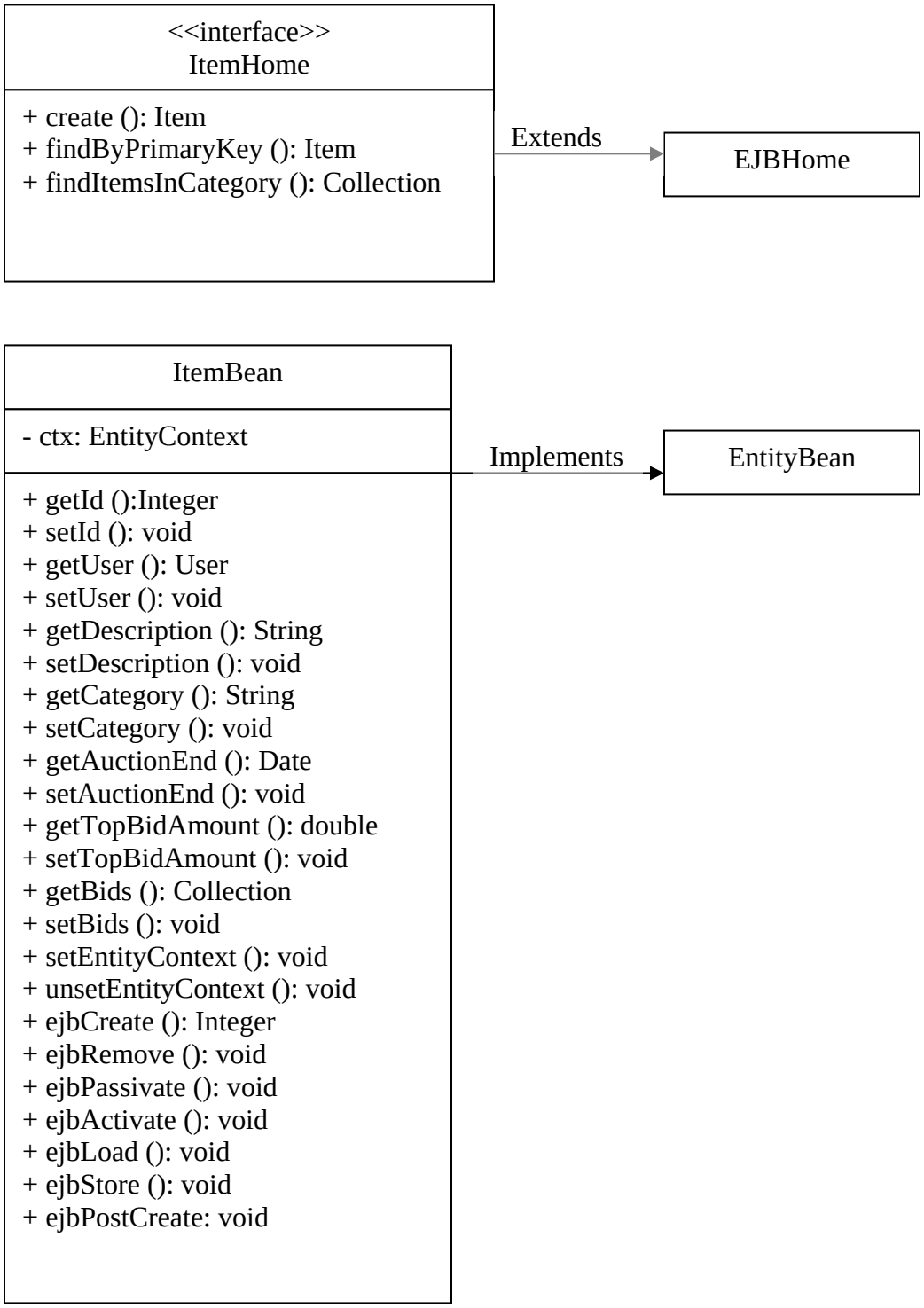

ItemValueHolder
- itemDescription:String
- topBidAmount: double
- userName: String
- id: int
+ setID: void
+ getID: int
+ setItemDescription: void
+ getItemDescription: String
+ setTopBidAmount: void
+ getTopBidAmount: String
+ setUserName: void
+ getUserName: String
Implements
Serializable
BidValueHolder
- itemDescription: String
- bidAmount: double
+ getItemDescription (): String
+ setItemDescription (): void
+ getBidAmount (): double
+ setBidAmount (): void

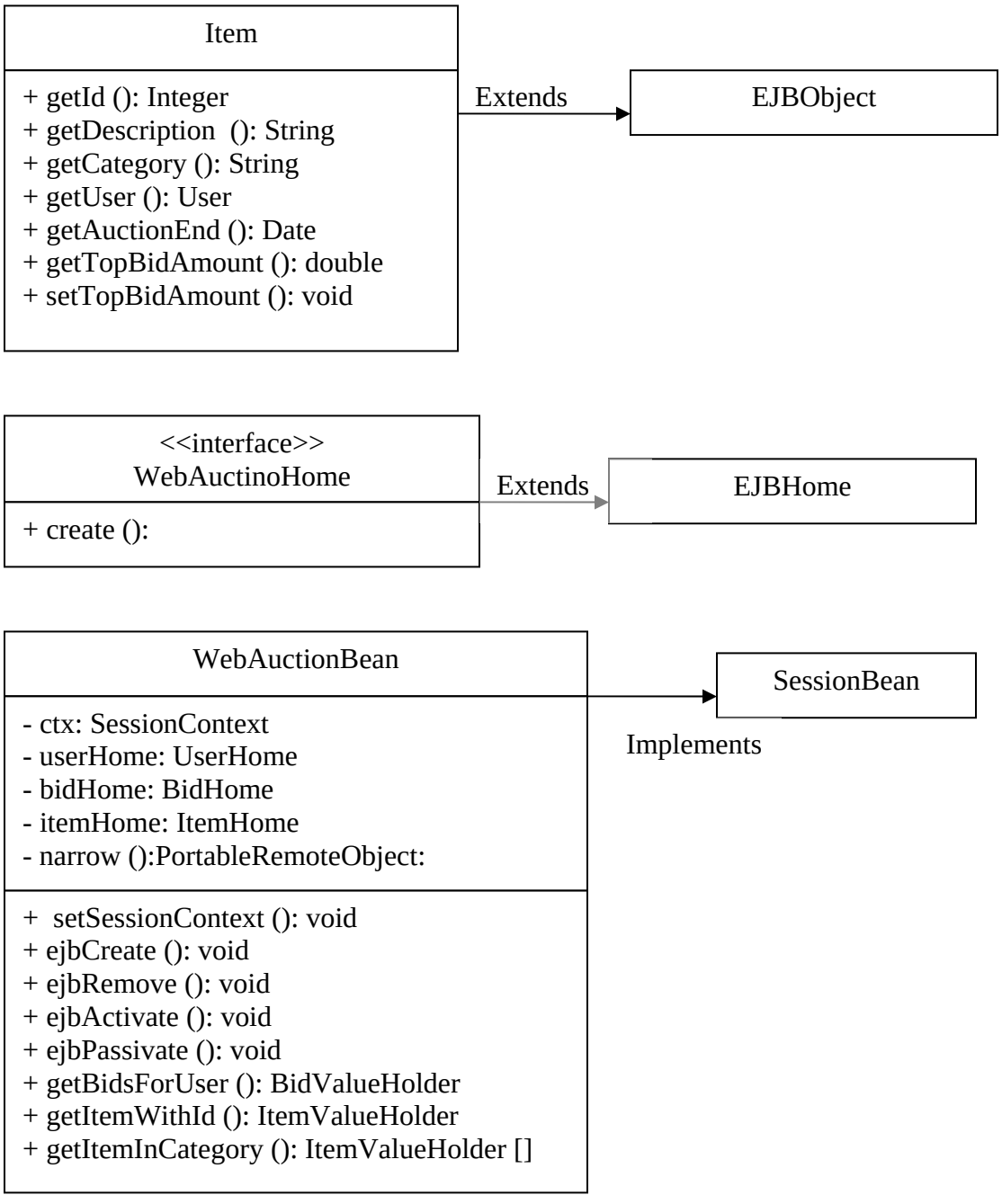

ItemCollectionTagExtraInfo
+ getVariableInfo ( ): VariableInfo[]
Extends
TagExtraInfo
GetItemWithId
- itemID: int
- webAuction: WebAuction
- Object narrow (): PortableRemoteObject
+ setItemID (): void
+ geItemID (): int
+ setPageContext: void
+ doStartTag (): int
GetItemInCategory
- category: String
- webAuction: WebAuction
- Object narrow (): PortableRemoteObject
+ setCategory (): void
+ getCategory (): String
+ setPageContext (): void
+ doStartTag (): int
Extends
TagSupport
GetBidsForUser
- webAuction: WebAuction
- Object narrow (): PortableRemoteObject
+ getUserName (): String
+ setUserName (): void
+ setPageContext (): void
+ doStartTag (): int

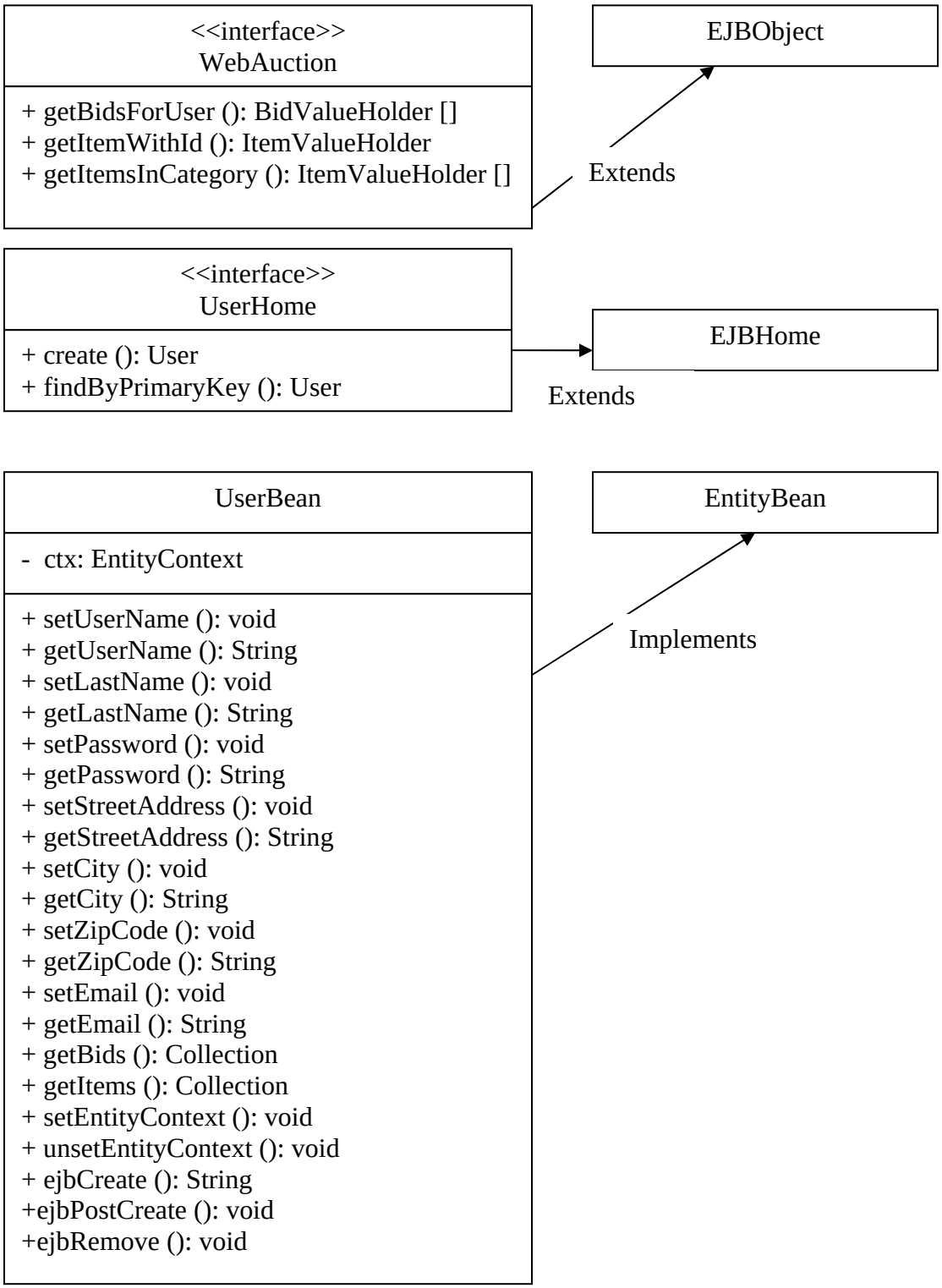

EnterItem
- itemHome: ItemHome
- userHome: UserHome
- idGeneratorHome: IDGeneratorHome
- userName: String
- category: String
- description: String
- setPageContext (): void
+ setUserName (): void
+ getUserName (): String
+ setCategory (): void
+ getCategory (): String
+ setPageContext (): void
+ doStartTag (): int
EnterBid
- itemId:int
- userName: String
- bidAmount: double
- qsender: QueueSender
- bidMsg: MapMessage
+ setItemId (): void
+ getItemId (): int
+ setUserName (): void
+ getUserName (): String
+ setBidAmount (): void
+ getBidAmount (): double
+ JMS_FACTORY: String
+ QUEUE_NAME: String
+ setPageContext: void
+ doStartTag (): int
Extends
TagSupport
BidCollecetionTagExtraInfo
+ getVariableInfo (): VariableInfo []
Extends
TagExtraInfo

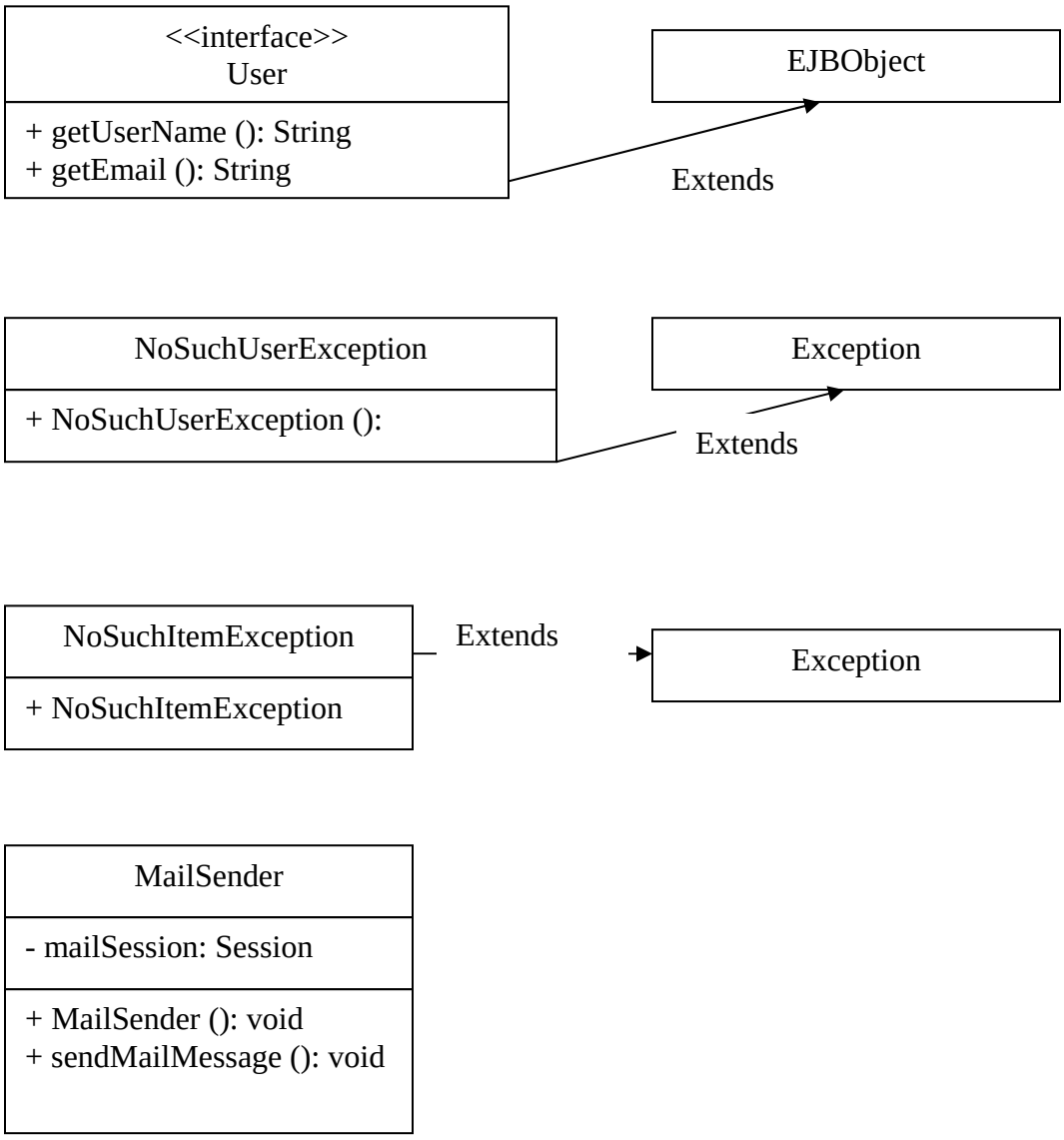

<<interface>>
BidHome
+ create (): Bid
+ findByPrimaryKey (): Bid
+ findBidsForUser (): Collection
+ findTopBidForItem (): Bid
Extends
EJBHome
<<interface>>
Bid
+ getId (): Integer
+ getAmount (): double
+ getItem (): Item
+ getUser (): User
Extends
EJBObject
BidBean
- ctx: EntityContext
+ getId (): Integer
+ setId (): void
+ getAmount (): double
+ setAmount (): void
+ getItem (): Item
+ setItem (): void
+ getUser (): User
+ setUser (): void
+ setEntityContext: void
+ unsetEntityContext(): void
+ ejbCreate (): Integer
+ ejbPostCreate (): void
+ ejbActivate (): void
+ ejbPassivate ():void
+ ejbLoad (): void
+ ejbStore (): void
+ ejbRemove (): void
Implements
EntityBean


<<interface>>
IDGeneratorHome
+ create (): IDGeneratorHome
Extends
EJBHome
IDGeneratorBean
- ctx: SessionContext
- dataSource: DataSource
- jdbcStatement: String
- POOL_NAME: String
- SEQ_NAME: String
- isCloudscape: boolean = false
- sequenceName: String
- setNextValueFromSequence (): int
- getNextValueFromTable (): int
- getNextValue (): int
+ setSessionContext (): void
+ ejbCreate (): void
+ ejbRemove (): void
+ ejbPassivate (): void
+ ejbActivate (): void
+ getNextValue (): int
Implements
SessionBean
<<interface>>
IDGenerator
+ getNextValue (): int
Extends
EJBObject


<<interface>>
ItemHome
+ create (): Item
+ findByPrimaryKey (): Item
+ findItemsInCategory (): Collection
Extends
EJBHome
ItemBean
- ctx: EntityContext
+ getId ():Integer
+ setId (): void
+ getUser (): User
+ setUser (): void
+ getDescription (): String
+ setDescription (): void
+ getCategory (): String
+ setCategory (): void
+ getAuctionEnd (): Date
+ setAuctionEnd (): void
+ getTopBidAmount (): double
+ setTopBidAmount (): void
+ getBids (): Collection
+ setBids (): void
+ setEntityContext (): void
+ unsetEntityContext (): void
+ ejbCreate (): Integer
+ ejbRemove (): void
+ ejbPassivate (): void
+ ejbActivate (): void
+ ejbLoad (): void
+ ejbStore (): void
+ ejbPostCreate: void
Implements
EntityBean


Item
+ getId (): Integer
+ getDescription (): String
+ getCategory (): String
+ getUser (): User
+ getAuctionEnd (): Date
+ getTopBidAmount (): double
+ setTopBidAmount (): void
Extends
EJBObject
<<interface>>
WebAuctinoHome
+ create ():
Extends
EJBHome
WebAuctionBean
- ctx: SessionContext
- userHome: UserHome
- bidHome: BidHome
- itemHome: ItemHome
- narrow ():PortableRemoteObject:
+ setSessionContext (): void
+ ejbCreate (): void
+ ejbRemove (): void
+ ejbActivate (): void
+ ejbPassivate (): void
+ getBidsForUser (): BidValueHolder
+ getItemWithId (): ItemValueHolder
+ getItemInCategory (): ItemValueHolder []
SessionBean
Implements


<<interface>>
WebAuction
+ getBidsForUser (): BidValueHolder []
+ getItemWithId (): ItemValueHolder
+ getItemsInCategory (): ItemValueHolder []
EJBObject
Extends
<<interface>>
UserHome
+ create (): User
+ findByPrimaryKey (): User
EJBHome
Extends
UserBean
- ctx: EntityContext
+ setUserName (): void
+ getUserName (): String
+ setLastName (): void
+ getLastName (): String
+ setPassword (): void
+ getPassword (): String
+ setStreetAddress (): void
+ getStreetAddress (): String
+ setCity (): void
+ getCity (): String
+ setZipCode (): void
+ getZipCode (): String
+ setEmail (): void
+ getEmail (): String
+ getBids (): Collection
+ getItems (): Collection
+ setEntityContext (): void
+ unsetEntityContext (): void
+ ejbCreate (): String
+ejbPostCreate (): void
+ejbRemove (): void
EntityBean
Implements


<<interface>>
User
+ getUserName (): String
+ getEmail (): String
EJBObject
Extends
NoSuchUserException
+ NoSuchUserException ():
Exception
Extends
NoSuchItemException
+ NoSuchItemException
Extends
Exception
MailSender
- mailSession: Session
+ MailSender (): void
+ sendMailMessage (): void