University Leipzig

Faculty of Mathematics and Computer Science

Developing Web Application with EJB3.0 and DB2 on WebSphere

Master Thesis

Li, Zheng

Index

1 Overview

1.1 Motivation

In modern society the e-business becomes more and more important for enterprises. Electronic business methods enable enterprises to link their internal and external data processing systems more efficiently and flexibly, to work more closely with suppliers and partners, and to better satisfy the needs and expectations of their customers.

There are many technologies and toolkits that can help enterprises to build their enterprise application platform. The most often used technology is J2EE that aims to simplify the design and implementation of enterprise applications. Since the release of EJB2.x, J2EE owns great success. According to the feedback from the developers, Sun provides EJB 3.0 that enhances the EJB specification, introducing a new plain old Java object (POJO)-based programming model that greatly simplifies development of J2EE applications .

An enterprise application must be deployed on a J2EE application server. In this work we choose WebSphere Application Server 6.1 that is the leading software platform designed by IBM. WebSphere Application Server 6.1 is a J2EE compliant application server and supports Java standard edition 1.5 [1]. With installation of EJB3 feature pack, WebSphere Application Server 6.1 provides support for EJB 3.0. WebSphere Application Server can be installed on a wide range of operating systems, so that an application can be easier exported from the develop machine to the deploy machine.

1.2 Target

For this master thesis, 5 projects: IBMWeb Basic Servlets, Xtreme Travel, ITSO bank, PlantShop and MDB Demo, are implemented, they are shown in Figure 7-1, where:

IBMWeb Servlet Demo project uses servlet technology.

Xtreme Travel project are developed with JSP technology.

ITSO Bank project is yet developed by Mr. Ronnenburger[2] with servlet and by Mr. Kumke[3] with EJB2.x technology respectively. This time the project is rewritten with EJB3.0 technology.

PlantShop project is the WebSphere Application Server 6.1's sample written with EJB2.x. The code is rewritten with EJB3 technology.

MDB Demo project is a sample for message driven beans.

All the projects are developed with Rational Application Developer V7.5 [4] and deployed on WebSphere Application Server 6.1.

1.3 Document Structure

Chapter 1 gives a brief introduction of the motivation and target. Chapter 2 introduces the Web Application Architecture. Chapter 3 introduces the servlet and JSP technologies that are used for web tier. Chapter 4 focuses on the EJB3 technology. In chapter 5, the features of WebSphere Application Server are mentioned. The Application Architecture is introduced in chapter 6. Chapter 7 explains the projects and their technical details. Chapter 8 reviews the document and mentions the future work.

2 Web Application

2.1 Static and Dynamic Web Pages

A static web page is a web page that is delivered to the user exactly as stored prepared information, such as Texts, Pictures and so on. A static web page provides all users the same information. Static web pages are often HTML documents stored as files in the file system and made available by the web server over HTTP protocol. The content in a static web page can not change itself, although it can make a interaction with users via some methods such as CSS , VBScript and JavaScript. These limitations bring developers to use server-side programming languages, and dynamic web pages.

A dynamic web page is a hypertext document that provides customized or actualized Information for each user according his individual requirement. In a dynamic web page, the content and the page layout are created separately. The contents are always stored in a database and retrieved on a web page only when a user asks for it.

2.2 Server-Side Scripts

A server-side script is one language that creates a HTML page on a server which sends the page content to client. There are many server-side scripts and methods used to generate dynamic web pages, such as Active Server Pages (ASP), Common Gateway Interface (CGI), PHP, Java Server Pages (JSP) and Java Servlets. Each method or language has its benefits and disadvantages.

In earlier day in the web world, CGI is the most performed method that is written in C, Perl and other shell scripts. Those scripts were executed by the operating system and depend on operating system. Each CGI program runs as a single process, that means, the server must create and reclaim memory space for each CGI program when it was invoked and terminated, which is quite time and resource costly [5].

Nowadays, interpretive languages are widely used to generate dynamic web pages. The today most popular web developing language such as Java Servlet, JSP, ASP and PHP are all interpretive language. Java Servlet and JSP are both based on Java. Compared with PHP, the Java programmer can easily master the syntax of JSP and Java Servlet and use a very large class library during the development of web pages. The other advantage of JSP and Java Servlet compared with ASP is the platform independence. The on Windows generated and compiled Servlets can run on any servlet engines whose Java versions are compatible [11]. To run JSP or Java Servlet we must use a servlet engine such as Tomcat[6], WebLogic[7], GlassFish[8] and WebSphere Application Server[9].

2.3 Web Application Architecture

2.3.1 3-Tier

A 3-Tier architecture is the most common model used for today's web applications. In the this model, the web browser on end-user acts as the client, an application server (such as Apache Tomcat, Sun Glassfish) acts as middle-Tier that handles the components processing business logic , data access, and a database server (such as DB2 or MySQL database servers) acts as another tier to keep data persistence. Figure 2.3-1 illustrates the 3-Tier model. The client needs only a Web browser instead of all other resource such as JDBC Driver, therefore, this client tier is also called thin client. The client can communicate with the middle tier in different ways, for example with HTTP, Remote Procedure Calls (RPC), Remote Message Invocation (RMI), Common Object Request Broker Architecture (CORBA) or Distributed Component Object Model (DCOM). The middle Tier handles the Business logic, organists and processes data and exchanges data with database server via some database query language.

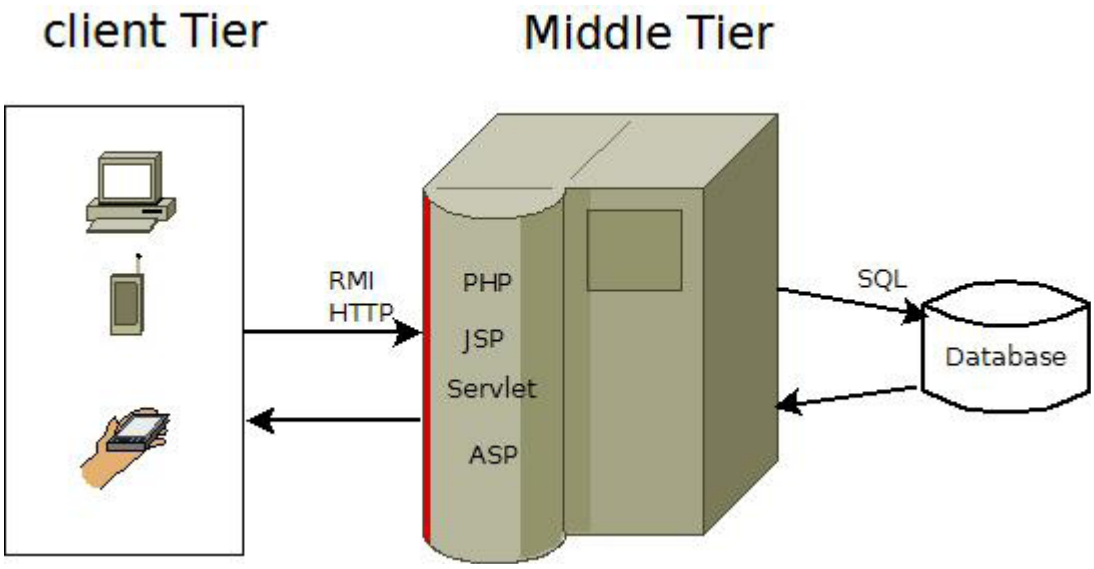


Figure 2.3-1 3-Tier

The advantages of 3-Tier architecture are mainly in the lower maintenance costs for the clients [10]. When the business logic is needed to be changed, the developers can put attention only on the middle Tier, by the separation of presentation and processing the code will be easier to change and maintain, the scalability of the system is also improved.

A disadvantage of 3-Tier architecture is that the programmer must set himself very strongly with the individual procedures on the middle tier, such as with competing access, the guarantee of ACID properties for transactions that require it, as well as authentication and authorization.

In order to overcome these problems, application server was developed. They provide the developer available via an Application Program Interface (API) for transaction management, multithreading, database access, resources bundling and load distribution. Thus the developer can concentrate himself on the business logic. However, it is bound to a specific application server, if the developer does not take it into account, to port the applications to a different application server.

The Java 2 Enterprise Edition (J2EE) as a framework was developed , which defines a standardized interface for all J2EE-compliant application server, so that the portability between J2EE-compliant application servers

is ensured[11]. The J2EE API provides e.g. APIs for the database connection (JDBC), the asynchronous message exchanged between program components (JMS), access to java naming and directory interface (JNDI) and the transaction APIs. More detailed information on J2EE can be found at [12].

2.3.2 N-Tier

Since the business logic and data management are mixed in middle Tier in 3-Tier model, it brings a problem, the programs are difficult to understand. When the business logic or data access even if the page layout to be changed, the developer must rewrite all the related programs in middle-tier, it needs that the developer must skillfully understand all the related technologies. With separation of the business logic, data management and layout management, each developer can concentrate himself only on one aspect. With the J2EE API standardization it is possible to modularize the applications even further and thus to use a 4- or 5-Tier model. Figure 2.3-2 shows a 4-Tier and 5-Tier model written in Java.

The work on the client side is thereby reduced to the representation of data(e.g. html- or wap-pages), the control of the representation takes place by a Servlet or a Java server Page running on a web server, the business logic is completely separated in Enterprise Java Beans (EJB) running on EJB container[12]. The difference between 4- and 5-tier shown in figure is that JSP and Servlet run in one or two tiers. In 5-Tier model, Servlet controls the front-side logic e.g. Cart checking and Count calculation and sends the result to JSP tier, JSP tier organizes the front-side layout e.g. setting the table according the item quantity in a cart.

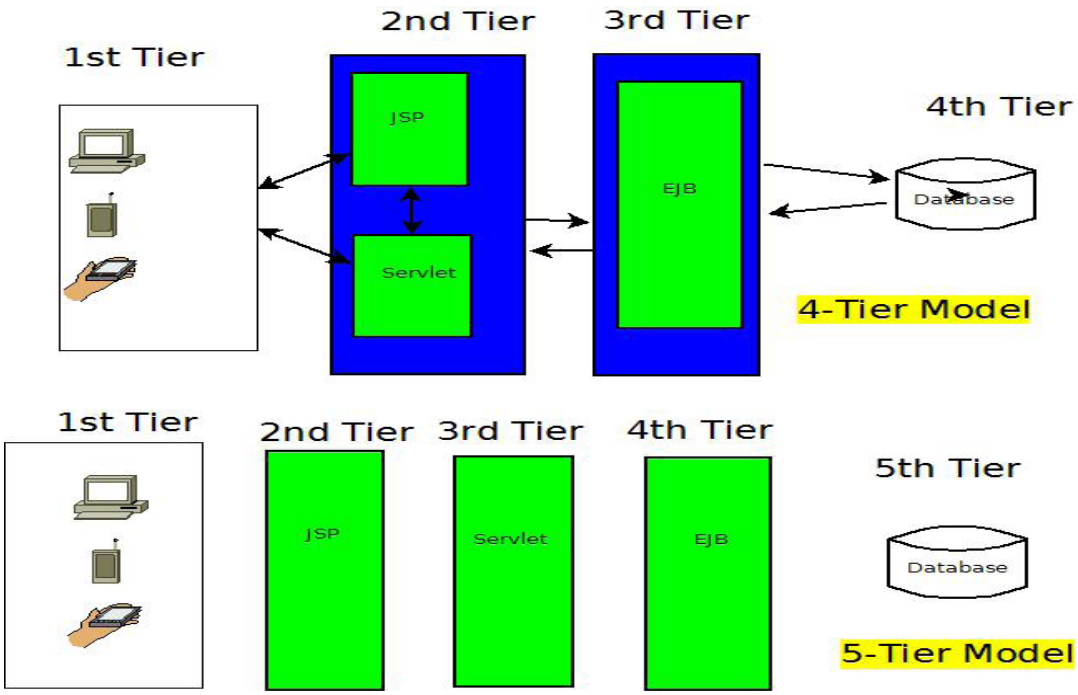


Figure 2.3-2 N-Tier

The advantages of 4- or 5-Tier model are clear – With more isolated Tiers the flexibility of application is enhanced. For example, the developers can design multiple user interface within JSP tier to fit different requirement e.g. Html, Wap without changing any business logic.

However, the system designed in N-Tier model becomes more complex, meanwhile, N-Tier application requires stronger hardware e.g. more memory requirement.

With these disadvantages, when to use N-Tier model is a question. The answer is only one word, scalability [10]. The N-tier model can scale up to extremely large systems without compromise. By large we are referring to the number of users, the number of differing user interface components, the size of the database, the structure of the network, and all of the other size issues for an application.

3 Java Servlet and JSP

3.1 Java Servlet

A Servlet is a Java Class that implements a specific interface javax.servlet.Servlet, which is defined by the Java Servlet API from Sun [12][13]. A Servlet is a pre-compiled class that receives a request from client via HTTP protocol and generates a response content based on that request, the generated content is commonly HTML, but can be other data such as XML. A servlet has no main () function, instead of it, has some standardized methods that are called by the servlet engine. A Servlet can keep state in session variables across many server transactions by using HTTP cookies, or URL rewriting.

The servlet lifecycle consists of the following steps:

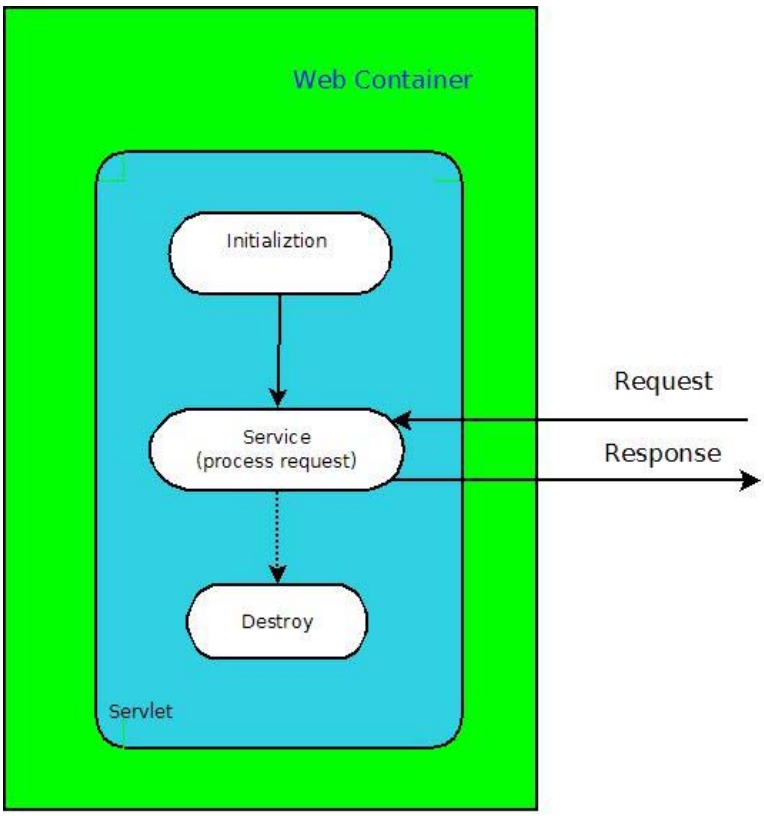


Figure 3.1 Lifecycle of servlet

1. A Servlet is loaded by a web container.
2. Web container calls init() method. It is used to initialize the servlets (e.g. for reading the configuration parameters or creating the persistent objects

or database connections). The initialization is usually performed just after the first calling of the Servlet by the client, the web container can also initialize the servlet even before that. It creates an object instance of the servlet. This instance will remain until the servlet is unloaded (e.g. with the stop of the Servlet container) or overwritten (e.g. after new compiling). Each request to this servlet will be processed by the same object instance, but be serviced in its own separate thread. Thus it is ensured that the resource consumption remains small. In the lifecycle of a servlet, the init() method is just called once.

3. After initialization, a servlet steps into service phase and stay in the phase until the destroy() method is invoked. When a client sends a request to the servlet, Web container calls the service() method of the servlet to process this request, each request is serviced in its own separate thread. With HTTP request, according to the kind of request – get or post, the service() method dispatches it to an appropriate method – doGet() or doPost() - to handle the request. The developer of the servlet must provide an implementation for these methods. If a request for a method that is not implemented by the servlet, the method of the parent class is called, typically sending a error message to the client.

4. At unloading of the servlet the method destroy () is called. It implements actions, which are necessary at the end of the Servlet lifecycle (e.g. terminating the persistent connections). Like the init () method, this method is also called only one time in the lifecycle.

Servlets can be integrated into existing html pages using Server Side Includes (SSI). The output content appears on the place where the servlet is included. In order to output data on different places in the final html-Code more servlets must be integrated.

More servlets can build a Servlet-Chain, one servlet sends its output data to an other servlet. The common Servlet-Chain is servlet filter [13].

3.2 JSP

Architecturally, JSP can be viewed as a java servlet by another name [14] . Unlike Java Servlet, a JSP file is a textual form document commonly with a file extension name .jsp. It mixes the HTML elements and JSP elements in one file where the JSP elements are expressed by tags. There are many types of tags used to present a JSP element. The following list introduces a few of them that are usually used.

- **<% %>** the commonly used tag; all in the tags enveloped codes are handled as Java code.
- **<jsp: useBean />** declares that the JSP page will use a bean that is stored within the session scope or request scope and so on according to the user setting, if the bean does not exist, the JSP page will create a new bean in the scope.
- **<c: />** declares that the JSP page will use customer tag.

The more information about JSP tags can be found at [].

A JSP page services the requests similar to a Java Servlet, therefore, the lifecycle of a JSP page is determined by Java Servlet technology. When the web container receives a request to a JSP page, it checks whether a servlet instance of the JSP page exists. If there's no instance, the web container compiles the JSP page to a servlet and loads it. Then the JSP servlet services the request as servlet. If the instance exists, the web container checks the creation time of the instance, if the instance is older than the JSP page, that means, the JSP page is changed after the creation of the instance, the web server performs the same steps that just mentioned.

Figure 3.2 illustrates the Lifecycle of a JSP page.

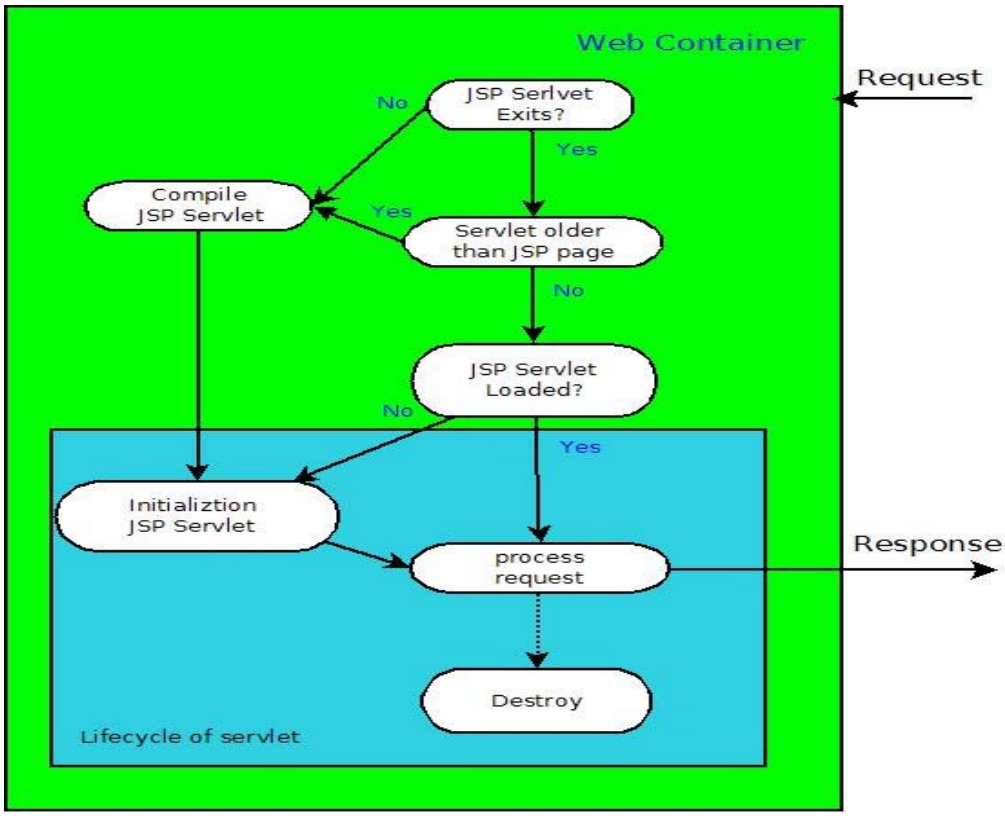


Figure 3.2 Lifecycle of a JSP page

4 Enterprise Java Beans

An enterprise bean is a server-side component that fulfils the business logic of an application and runs in an EJB container [13]. The business logic is the code that realizes the purpose of the application. There are three types of Enterprise Java Beans, session bean, entity bean and message driven bean. Table 4 lists the usage and character of them.

Type	Usage	Event processing	Interface
Session	Implements the business logic and performs the request for a client	synchronous	Remote Home
Entity	Organizes and contains the data that are stored in persistence device	synchronous	Remote Home
Message Driven	Listens a message destination and processes the message delivered from a message sender.	asynchronous	

Table 4 Enterprise Java Bean Types

The advantage of using enterprise beans is that the enterprise beans simplify the development of large, distributed applications. The beans implement the business logic of the application and provide interfaces to client, thus the client can keep up the request and layout, although the business logic is changed. With the separating of the client and beans, the client developer can focus on the data presentation or layout for client. The bean developer can concentrate on implementation of business logic. The beans can be accessed by many clients through the interface, thus depending on the existing beans the developer can build new applications quickly.

For requests, session beans and entity beans handle them as soon as they receive the requests, but message driven bean may process the requests later, because the messages may be delivered to that beans asynchronous.

A session bean and an entity bean must implement one or both of Remote and Home interface to provide the client services. A message bean has no such interface, because it runs behind the sense and keeps no contact with clients.

4.1 Entity Bean

An entity bean is a component that contains and presents the data stored in persistent storage mechanism. A storage mechanism can be a relational database, XML files and so on. Typically, each entity bean is a mapping of a table in a relational database, and each instance of the bean matches a record in that table.

Persistence

Persistence means that the state of an entity bean exists longer than the lifetime of the application. An entity bean is persistent, because its state is saved in a storage mechanism.

There are two types of persistence for entity beans:

- bean-managed persistence, for short BMP
- container-managed persistence, for short CMP

With *bean-managed* persistence, the database access method, such as opening/closing a database connection and query calls, must be written by the developer himself. If an entity bean has *container-managed* persistence, the database access methods are automatically generated by the EJB container. The bean's code contains no database access methods.

Primary Key

Unlike a table in a database, each entity bean must have a unique object identifier (also called primary key), even if the mapped table has no primary key. Then unique identifier enables the client to locate a particular

entity bean.

Relationships

Like tables in a relational database, an entity bean may have relationship with other entity beans. For BMP beans, the developer must write code himself to implement the relationships. But with CMP beans, the EJB container takes over the management of the relationships. There are 3 relationships:

- One - One
- One - Many
- Many – Many

Lifecycle

Figure 4.1–1 illustrates the lifecycle of an entity bean.

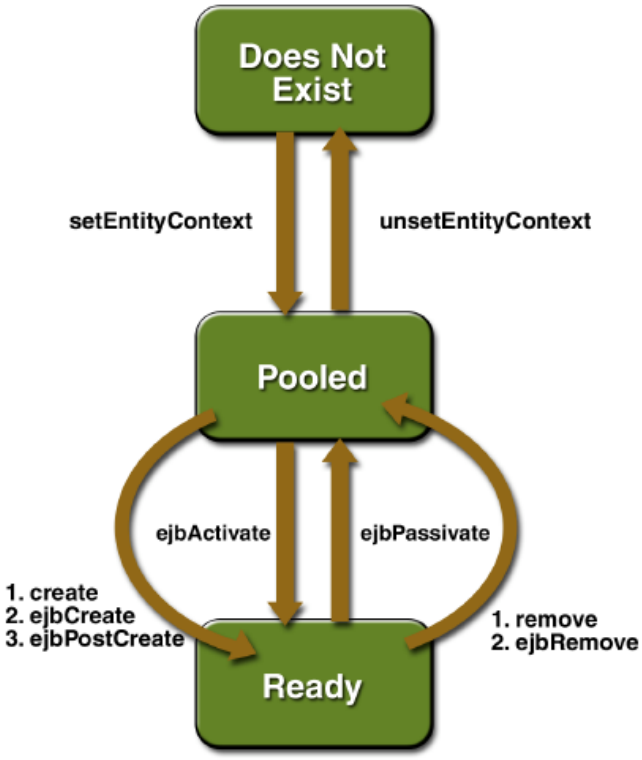


Figure 4.1-1 Lifecycle of an entity bean[13]

An entity bean has only two states during its lifecycle: pooled and ready.

After the EJB container creates an instance of an entity bean, the EJB container puts the instance into a pool of available instances. In the pooled stage, the instance has no primary key. All instances in the pool are identical. The EJB container assigns an identity to an instance when moving it to the ready stage.

There are two ways from the pooled stage to the ready stage. First, the client invokes the create() method, causing the EJB container to call the ejbCreate() and ejbPostCreate() methods. Second, the EJB container invokes the ejbActivate() method.

When an entity bean is in the ready state, it can response for the quest from client, its business methods can be invoked. There are also two paths from the ready stage to the pooled stage. In the first way a client can invoke the remove() method, which causes the EJB container to call the ejbRemove() method. In another way, the EJB container can invoke the ejbPassivate() method to put the bean into pool.

At the end of the lifecycle, the EJB container removes the instance from the pool and invokes the unsetEntityContext() method.

EJB 3.0 uses Java Persistence API (JPA) to simplify and enhance the implementation of entity beans, section 4.5 will give a simple introduction.

4.2 Session Bean

A session bean implements the business logic of an application that runs inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs the business logic for its client, shielding the client from complexity by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, EJB container creates more instances for one bean. Like an interactive session, a session bean is not persistent. (Here "not persistent" means that the data in a session bean is not saved to a database.) When the client terminates,

its session bean appears to terminate and is no longer associated with the client.

4.2.1 Stateless Session Bean

A stateless session bean does not maintain the state for the client. When a client invokes a method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of this invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Because stateless session beans does not retain the state of a client, it can be assigned to more clients, so they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

Lifecycle

Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 4.2.1 illustrates the stages of a stateless session bean. After the EJB container creates instance(s) of a stateless session bean and then invokes the setSessionContext() and ejbCreate() method, then the bean enters into the Ready state to respond the request from the client. After performing the business methods to respond the client request, the session bean is ready for garbage collection by invoking the ejbRemove() method.

Figure 4.2.1 Life Cycle of a Stateless Session Bean[13]

4.2.2 Stateful Session Bean

As its name suggests, a stateful session bean performs also the business logic and retains the state for the duration of client-bean conversation.

Lifecycle

Because the state of a client is retained, the stateful session bean has 3 states during the lifecycle: nonexistent, ready to respond request and passivating. Figure 4.2.2 illustrates the states that a stateful session bean passes through during its lifetime.

The client initiates the life cycle by invoking the create method. The EJB container creates an instance of the bean and then invokes the setSessionContext() and ejbCreate() methods in the session bean.

The bean is now in the ready stage to perform the business methods to respond the invoking from client. While in the ready stage, the EJB container may decide to deactivate or passivate the least-recently-used bean by moving it from memory to secondary storage. The EJB container invokes the bean's ejbPassivate() method immediately before passivating it. If a client invokes a business method on the bean again when it is in the passive stage, the EJB container reactivates the bean, calls the bean's ejbActivate() method, and then moves the bean in to memory and changes

the stage to ready stage.

At the end of the life cycle, the client invokes the remove() method, and the EJB container performs the bean's ejbRemove() method. The bean's instance is ready for garbage collection.

Figure 4.2.2 Lifecycle of a stateful session bean[13]

4.3 Message Driven Bean

4.3.1 Java Message Service(JMS)

The Java Message Service is a set of Java APIs that allows applications to create, send, receive, and read messages. The JMS API defines a set of interfaces and associated semantics that allow programmers write messaging components in Java that communicate with other messaging implementations.

JMS API Programming Model

- Administered Objects: *Administered objects* are preconfigured JMS objects created by an administrator that consists of two components *connection factories* and *destinations*

- Connections
- Sessions
- Message Producers
- Message Consumers
- Messages

A *connection factory* is the object for clients to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the *ConnectionFactory, QueueConnectionFactory*, or *Topic-ConnectionFactory* interface.

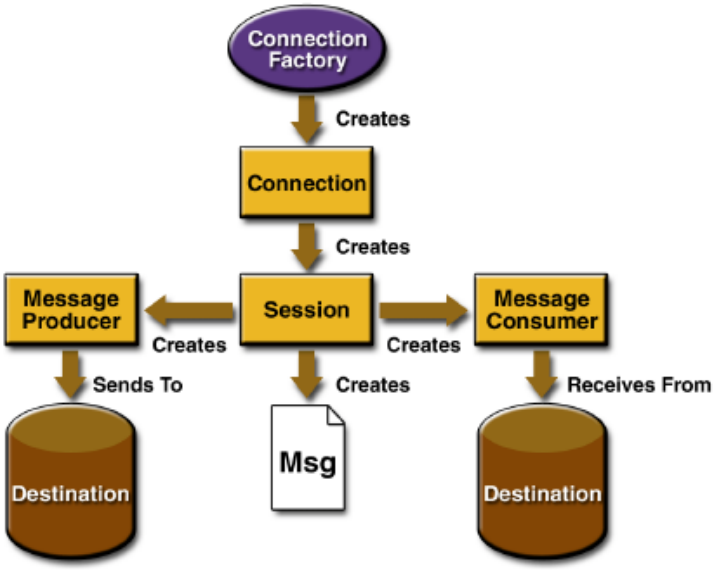


Figure 4.3.1-1 JMS Programming Architecture[13]

A *destination* is the object that specifies the target where the producers deliver the created message to and the consumers get the message from. Two kinds of destinations are *Queue* and *Topic*. In Queue model, each message must be gotten by zero or one consumer. In Topic model each message can be processed by many consumers, the message is stored in memory until all consumers have gotten it.

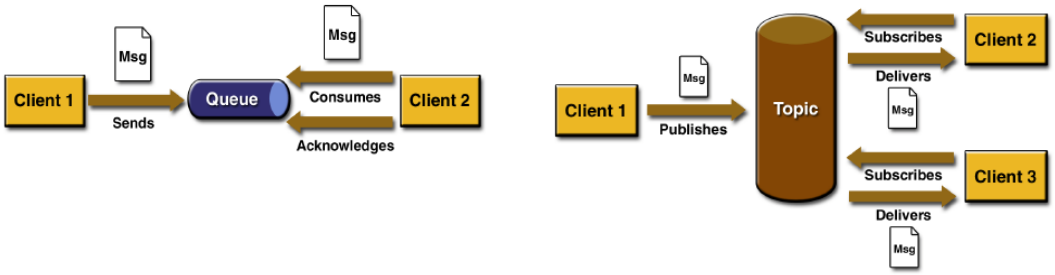


Figure 4.3.1-2 *Queue* / *Topic* destination[13]

A *connection* creates a virtual connection on open TCP/IP socket between a client and a provider service daemon.

A *session* is a single-threaded context for producing and consuming messages. The *sessions* are created by a *connection*.

A *message producer* implements the MessageProducer interface that is created by a session and used for sending messages to a *destination*.

A *message consumer* is an object that is created by a session and implements the MessageConsumer interface in order to receive messages sent to a destination which can be either a *Queue* or a *Topic.*

The purpose of a JMS application is to create and to deliver messages that can then be used by other components. A JMS message has three parts: a *header*, *properties*, and a *body*. Only the *header* is absolutely necessarily, the other 2 parts can be absence in one *message.*

A JMS message header has a number of predefined fields that are used by clients and providers to identify and to route messages. Each header field has setter and getter methods itself. Table 1 shows all the fields and the place where the fields are set.

HEADER FIELD	SET BY
JMSDestination	send or publish method
JMSDeliveryMode	send or publish method
JMSExpiration	send or publish method
JMSPriority	send or publish method
JMSMessageID	send or publish method
JMSTimestamp	send or publish method

JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

Table 4.3.1-1 Properties of Message Header

If we have additional information to set in a message for other components, we could use *Message Properties*, for an example of needing a property for a *message selector* .

The body contains the content of a message. Eevey messsage content must obey one pre-defined *message format*, also named *message type*, which allows software components to send and to receive data in different forms. Table 2 shows the *message types.*

Message Type	**Body Contains**
TextMessage	A java.lang.String object
MapMessage	A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
BytesMessage	A stream of uninterpreted bytes. This message type is for literallyencoding a body to match an existing message format
StreamMessage	A stream of primitive values in the Java programming language, filledand read sequentially.
ObjectMessage	A Serializable object in the Java programming language.
Message	Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.

Table 4.1.1-2 JMS Message Types

4.3.2 Message Driven Bean

A message driven bean acts normally as a message listener and a message consumer within the scope of an EJB container and processes

the messages asynchronous.[] A message driven bean listens in a message destination and processes the messages that are sent to the message destination. All J2EE components, a servlet, a JSP page, another enterprise bean and so on, can send messages to a message destination. Message driven beans can process any type of JMS message or other kinds of messages.

Unlike session beans and entity beans, message driven beans provide no interface to client. It can be only invoked by the EJB container through calling the onMessage() method. When a session/entity bean receives a request, it responds immediately. But for a message driven bean, it is different, EJB container decides when to invoke the message driven beans, normally when the system is not busy, therefore, message driven beans handle messages asynchronous.

In some aspects, a message driven bean resembles a stateless session bean. Message driven beans are stateless. All instances of a message-driven bean are equivalent, the EJB container can assign a message to any instance.

Lifecycle of a Message-Driven Bean

Like a stateless session bean, a message driven bean has only two states: nonexistent and ready to receipt message. Figure 4.3 illustrates the lifecycle of a message driven bean. The EJB container usually creates a pool of message-driven bean instances at startup. For each instance: the EJB container instantiates the bean and performs these tasks:

1. It calls the setMessageDrivenContext() method to pass the context object to the instance.
2. It calls the instance's ejbCreate() method.

Then the message driven bean is ready to process the message when the container calls the onMessage() method. At the end of the lifecycle, the

container calls the ejbRemove() method. The bean's instance is then removed and ready for garbage collection.

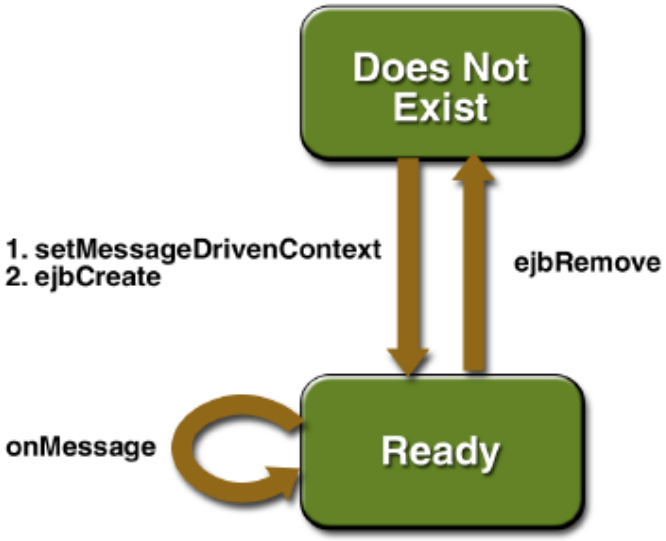


Figure 4.3 Lifecycle of a message driven bean[13]

4.4 New Features in Enterprise Java Bean 3.0 (EJB3)

According to the feedback from a large part of the J2EE community, developers think that the development with earlier versions of EJB is unnecessarily complex, although it works well. During the development, developers had to spend more time to write APIs that focused on the EJB container's requirements than that implemented the business logic [15]. For Example:

● The EJB 2.x specification requires that a session/entity bean must implement one or both of the Remote and Home interface that extend an interface from the EJB framework package. This causes a tight coupling between the developer-written code and the interface classes from the EJB framework package. This also requires the boring repeated work to implement several unnecessary callback methods (ejbCreate(), ejbPassivate(), ejbActivate()) that do not directly related to the business logic. Furthermore, the exceptions caused by the unnecessary methods must be handled.

● An XML deployment descriptors are overly verbose, complex. Much of the information required in the deployment descriptor could be set with default values.

● Resources must be accessed through JNDI.

● The container-managed persistence model is complex to develop and manage.

● The persistence mapping model was never well defined in EJB2.x , this causes that entity beans can not fit to all J2EE containers without any changing.

These negative aspects can be inferred that have the following reasons.

● J2EE development is too complex.

● Over-use of XML-based configuration.

● Persistence model is considered obsolete.

To overcome these shortages, Sun provides Enterprise JavaBeans 3.0 that is a major enhancement to the EJB and greatly simplifies the development of J2EE applications. The significant feature in EJB3 is the wide use of annotations that brings innovative techniques, such as: Metadata Annotations, Lifecycle Interceptors, Dependency Injection，EJB Injection and Java Persistence API (JPA).

Metadata Annotation

Using Metadata Annotation simplifies the steps to define a bean.

For example, we want to define a stateless session bean BankServBean, that provides different services for remote client(that can run on other JVM) and local client(that must run on the same JVM). We must define the following interfaces.

EJB component interface: Used by an EJB client to gain access to the capabilities of the bean. This is where the business methods are defined. There are two types, BankServLocal and BankServ, they extend

javax.ejb.EJBLocalObjct and javax.ejb.EJBObject respectively.

```
public interface BankServBean extends javax.ejb.EJBObject

public interface BankServBeanLocal extends javax.ejb.EJBLocalObject
```

EJB home interface: Used by an EJB client to gain access to the bean. Contains the bean life cycle methods of creating, finding, or removing. There are two types, BankServLocalHome and BankServHome, they extend javax.ejb.EJBLocalObjct and javax.ejb.EJBObject respectively.

```
public interface BankServBeanHome extends javax.ejb.EJBHome

public interface BankServBeanLocalHome extends javax.ejb.EJBLocalHome
```

At last, we must configure their relationship in the Deployment Descriptor, the stateless attribute of BankServBean is also defined in the file. The following code segment shows the configuration.

```
<ejb-jar id="ejb-jar_ID">
    <enterprise-beans>
            <session id="BankServBean">
                <ejb-name>BankServBean</ejb-name>
                <home>bank.BankServBeanHome</home>
            <remote>bank.BankServBean</remote>
            <local-home>bank.BankServBeanLocalHome</local-home>
            <local>bank.BankServBeanLocal</local>
            <ejb-class>bank.BankServBeanBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
  </enterprise-beans>
```

In EJB3 development model, the work is significantly reduced, we must only define 2 simple interfaces, whose relationship is also defined in class without any configuration in Deployment Descriptor. The following code shows the steps to define a stateless session bean in EJB 3 model.

1	Define 2 interfaces	public interface BankServ{……} public interface BankServLocal{……}

2	Define BankServBean	**@Remote(BankServ.class)**
		@Local(BankServLocal.class)
		@Stateless
		public class BandServBean **implements** BankServ, BankServLocal {
		….
		}

On step 1, we define 2 plain interfaces BankServ, BankServLocal, where we define the business logic methods.

On step 2, we define a plain java class BankServBean that implements the methods defined in both interfaces. Annotation @Stateless indicates to EJB container that the BankServBean is a stateless session bean, the annotations @Remote and @Local specify the interfaces that expose the bean on the remote and local client respectively.

If the bean is a stateful session bean, we can replace the annotation @Stateless with @Stateful.

Lifecycle Interceptor

In EJB2.x model, developers must implement several lifecycle callback methods, such as ejbPassivate(), ejbActivate(), ejbLoad(), and ejbStore(), even if they are not needed for business logic.

In EJB3 model, developers can implement only the lifecycle callback methods that are useful for the business logic, the other lifecycle callback methods are invoked automatically by EJB Container with default setting. Any method can act as the callback methods, when it has a lifecycle

callback annotation. The following code gives an example for BankServBean that has two methods, init() and cleanup() , they act as the callback methods PostConstruct() and PreDestroy() respectively. When the lifecycle event PostConstruct/PreDestroy is triggered, EJB container invokes the init()/cleanup() method automatically.

```
   @Remote( BankServ.class)
   @Local( BankServLocal.class)
   @Stateless
public class BankServBean implements BankServ, BankServLocal {
    @PostConstruct
    public void init( ){ …. }
    …..
    @PreDestroy
    public void cleanup( ) {…..}
….
   }
```

Dependency Injection

In EJB 2.x, the only way to get Java EE resources (JDBC data source, JMS factories and queues, and EJB references) was to use JNDI lookup, developers must write a piece of code. That work could become repeated, boring and vendor specific, because in many cases developers had to specify properties related to the specific J2EE container provider. The following code piece shows how to find a Datasource in EJB 2.x model.

```
    Datasource ds;
    ……..
    parms.put(Context.INITIAL_CONTEXT_FACTORY,
```

```
        "com.ibm.websphere.naming.WsnInitialContextFactory");
    InitialContext ctx = new InitialContext(parms);
        ds = (DataSource) ctx.lookup("java:comp/env/jdbc/db2");
```

EJB 3.0 adopts a *dependency injection* (DI) pattern, which is a better way to implement loosely coupled applications. It is much easier to use and more elegant than older approaches. When we want to use a database resource, we can write the code in EJB3 model using resource injection as below. But before we use the DI, we must specify the resource type, name and JDNI name in the web.xml file, the resource name can not be same as the JDNI name, because the valid scope for resource name is limited in the web module. The following code shows how to use DI in a servlet.

```
import javax.annotation.Resource;
……
@Resource(name = "jdbc/db2")
private DataSource ds;
```

Same as Resource Injection, EJB injection is also base on dependency injection, which is used for injecting session beans into a client. When a servlet wants to use a bean, the bean is so invoked as below,

```
Import javax.ejb.EJB
public class ListAccounts extends HttpServlet implements Servlet{
…..
@EJB BankServ bank;
```

Where @EJB indicates that the variable bank is an instance of enterprise bean that implements the interface BankServ. There is a special note for stateful session bean injection. Because a servlet is a multi-thread object, developers can not use dependency injection for a stateful session bean that must be explicitly looked up through JNDI. If developers use injection to define a stateful session bean object in servlet, the bean performs as a stateless session bean.

The dependency injection must be used by enterprise bean or servlet inside the management of EJB container, jsp doesn't support the technique.

4.5 JPA

In EJB2.x the mapping between CMP entity bean and database must be implemented through an XML-based deployment descriptor file, according to the feedback from many developers, the work is thought as complex and obsolete, because the deployment descriptor file is not usable in other EJB framework. To simplify the work, EJB3 model specifies a new persistence model—JPA. As a part of JSR220[17] the Java Persistence API provides a POJO (plain old java object) persistence model for object-relational mapping to replace the entity bean used in EJB2.x model. In EJB3 the concept of entity beans has also been substituted by the JPA entities to clarify the distinction.

According to the JPA specification, a JPA entity must comply with the rules:

- The entity class must be annotated with the Entity annotation or denoted in the XML descriptor as an entity.
- The entity class must have a no-argument constructor.
- The no-argument constructor must be public or protected.
- The entity class must be a top-level class.
- The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
- The class must define an attribute that is used to identify in an unambiguous way an instance of that class (it corresponds to the primary key in the mapped relational table)
- If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the Serializable interface.

In this section I will give a simple JPA introduction with help of an on-line banking system. This system has only two JPA entities: Customer and

Account.

A simple JPA Entity

```
@Entity
public class Account implements Serializable {
    @Id
    private String id;
    private BigDecimal balance;
…
}
```

The @Entity annotation indicates that the Account class is a JPA entity.

The @Id annotation indentifies the variable id that corresponds to the primary key in the mapped table

In this example the entity is persisted to a table named Account, and all the attribute names must be identical with the column names.

Mapping the table and columns

The above entity is restricted to match a table whose name is identical. To enhance the flexibility, JPA provides a user setting using annotation.

```
@Entity
@Table( schema="prak224", name="bank_account")
public class Account implements Serializable {
    @Id
    @Column(name="Account_ID")
    private String id;

    @Column(name="balance")
    private BigDecimal balance;
…
```

```
}
```

The @Table annotation defines which table and schema relate to the JPA entity.

The @Column annotation provides the column name that matches the entity property.

Relationships between entities

As explained in Chapter 4.1, a JPA entity may have relationships with other entities to reflect the table relationships in a database. In RMBMS the relationships are defined through foreign keys, in JPA the relationships are defined through object references from a source object to the target object. There are 3 relationship types defined in JPA: one-to-one, one-to-many, many-to-many. The relationship can be either unidirectional or bidirectional. Unidirection means, if a source object references a target object, it is not ensured that the target object also has a relationship to the source.

One-to-one

We consider the situation where each account must have an owner, however, a customer can have maximal one account or no account. So the account table has a foreign key customer_id that references to the id property, the primary key in customer table.

```
@Entity
public class Account implements Serializable {
    @Id
    private String id;

    @OneToOne
    @JoinColumn( name="customer_id", unique=true, nullable=false)
    private Customer customer;
```

```
    private BigDecimal balance;
    …
}
```

The @OneToOne annotation is used to identify the relationship type.

In the database, a relationship mapping means that a table has a reference to another table. The database term for a column that refers to a key (usually the primary key) in another table is a foreign key column. In the Java Persistence API, we call them join columns, and the @JoinColumn annotation is used to configure these types of columns. In this sample, the @JoinColumn annotation is used to specify the mapped column (Customer_ID) for joining the entity association.

In many situations, the relationship is bidirectional, the target object has a reference back to the source. In this case, we define a reference in target object towards the source.

```
@Entity
public class Customer {
@Id
private int id;

@OneToOne(mappedBy=”customer”)
private Account account;
....
}
```

The @OneToOne(mappedBy=”customer”) annotation specifies the relationship type, the mappedBy element is used to specify that the relationship is inverse directional, the source direction is defined in the parameter of mappedBy element, in this sample, the source directional relationship is defined in the customer property in Account entity.

One-to-many and many-to-one

We consider the situation where a Customer can have more accounts. In this case, there are a one-to-many relationship for customer entity and a many-to-one relationship for account entity. The following code piece shows how to define the one-to-many relationship.

```
/****** in Account class ********/
@Entity
public class Account implements Serializable {
    @Id
    private String id;

    @ManyToOne
    @JoinColumn( name="customer_id", nullable=false)
    private Customer customer;

    private BigDecimal balance;
    …
}

/****** in Customer classs ******/
@Entity
public class Customer {
    @Id
    private int id;

    @OneToMany(mappedBy=”customer”)
    private Set<Account> accounts;
....
}
```

In this case we set the @ManyToOne annotation on the Account entity because it holds the relationship(owns the foreign key). The Customer entity holds the inverse relationship signed with mappedBy element. Because one customer can have more accounts, we use Set<Account> to contain the accounts.

Many-to-many

If everybody of a couple has his/her private account and shares an account together, the relationship between customers and accounts is many-to-many. In this case, the relationship is always implemented through a join table in database. Figure 4.5-1 illustrates the relationship. To implement the relationship in JPA entity, we use @JoinTable and @ManyToMany anntations.

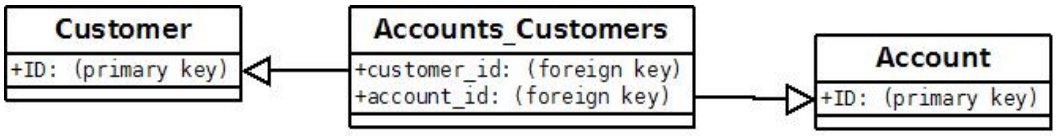


Figure 4.5-1 many to many relationship

```
/****** in Account class ********/
@Entity
public class Account implements Serializable {
    @Id
    private String id;

    @ManyToMany
    @JoinTable( name="Accounts_Customers", schema=”prak224”,
                        jionColumns=@JoinColumn(name=”customer_id”),

    inverseJoinColumns=@JoinColumn(name=”account_id”) )
    private Set<Customer> customesr;
```

```
        private BigDecimal balance;
        …
}

/****** in Customer classs ******/
@Entity
public class Customer {
        @Id
        private int id;

        @ManyToMany(mappedBy="customers")
        private Set<Account> accounts;
....
}
```

The @JoinTable annotation is used to assign the table in the database that associates customers with accounts. The entity that specifies the JoinTable is the owner of the relationship, so in this sample the Account entity is the owner of the relationship with the Customer entity.

The joincolumn pointing to the owning side is described in the joinColumns element, while the join column pointing to the inverse side is specified by the inverseJoinColumns element.

JPA query language

The Java persistence query language (JPQL) is an extension of the Enterprise JavaBeans query language (EJB QL) and combines the syntax and simple query semantics of SQL. Figure 4.5-2 shows the main architectural components that support JPQL.

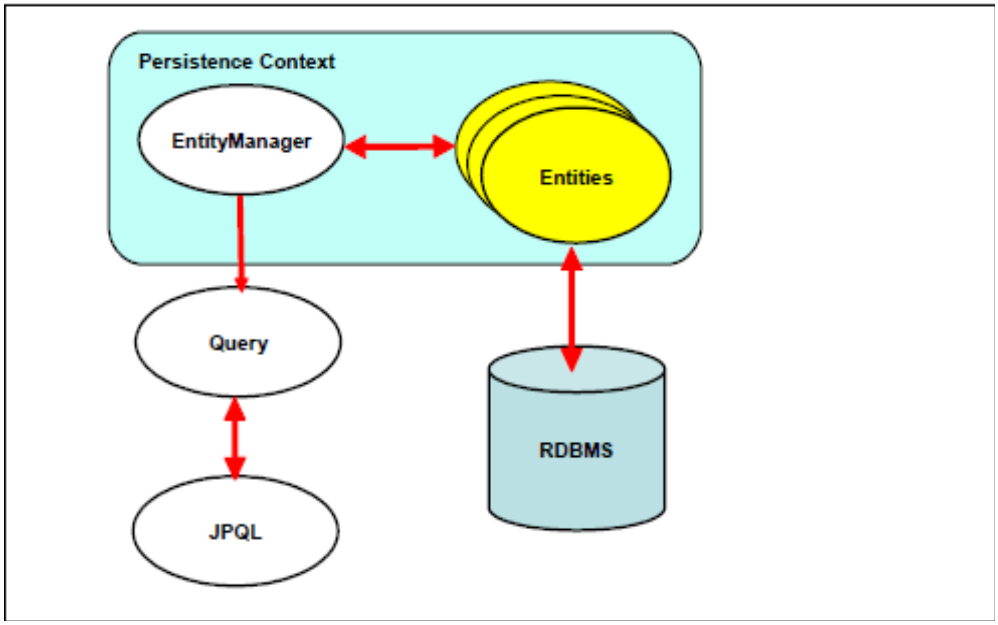


Figure 4.5-2 Main components included in JPQL[16]

The application creates an instance of the javax.persistence.EntityManager interface.

The EntityManager creates an instance of the javax.persistence.Query interface, through its public methods, for example createNamedQuery. The Query instance executes a query (to read or update entities).

There are 2 types of queries defined in JPQL:

Dynamic queries: They are created at run time.

Named queries: They are intended to be used in contexts where the same query is invoked several times. Their main benefits include the improved reusability of the code, a minor maintenance effort, and finally, better performance, because they are evaluated once.

From the technical aspect, a dynamic / named query can be seen as a JDBC Statement / PreparedStatement. However, named queries exist in a global scope, so that different EJB3 components can invoke them.

Query instances are created using the methods exposed by the EntityManager interface. There are 3 methods used to create a query instance. Table 4.5-1 shows the methods and their usage.

Method	Description
createQuery	Create an instance of Query for executing a Java Persistence query language statement.
createNamedQuery	Create an instance of Query for executing a named query
createNativeQuery	Create an instance of Query for executing a native SQL statement, for example, invoking a procedure.

Method	Code
createQuery	EntityManager em = ... Query q = em.createQuery("SELECT c FROM Customer c"); List<Customer> results = (List<Customer>)q.getResultList();

createNamedQuery	/***** a JPA entity ****/
	@Entity
	@Table (schema="ITSO", name="CUSTOMER")
	@NamedQuery(name="getCustomerByName",
	query="select c from Customer c where c.lastName = ?1")
	public class Customer implements Serializable {
	...
	}
	/*** a stateless session bean *****/
	public class EJB3BankBean implements EJB3BankService {
	String name;
	EntityManager em = ...
	……..
	query = em.createNamedQuery("getCustomerByName");
	query.setParameter(1, name);
	return (Customer)query.getSingleResult();
	…….

createnativeQuery	/** a procedure named ChangeTax defined in database **/ EntityManager em = ... private String QueryNoneReturnValueStoreProcedure() { **Query query = em.createNativeQuery("{call ChangeTax()}");** **query.executeUpdate();**

	}

Table 4.5-1 How to create a Query instance

The more information about JPQL can be found at[13][16][17].

5 Application Server

5.1 Web Container

A web container, also called servlet engine or servlet container, is a Java runtime environment which implements the web component contract of the J2EE architecture to response the request from client and manage the lifecycle of JSP pages and servlets [13].

A web container acts as an interface between client and JSP/Servlet, when a web container receives a request from client side; it invokes the instance of a servlet and manages its lifecycle. In other word, JSP pages and servlets are running inside of a web container. An application does never directly instantiate any servlet or call the init(), service() or destroy() methods, that are taken over by web container. Web container automatically instantiates and initializes the Servlets on application startup or when the Servlets are invoked for the first time.

J2EE specification [J2ee_spec] defines the contract between JSP/Servlet and container, and specifies the deployment model for them. The contract specifies how to develop and deploy JSP/Servlet, and how can JSP/Servlet access the services provided by container. In J2EE the contract is specified by various interfaces and classes, developer writes the classes that implement these interfaces or extend the classes and provide corresponding implementation of various methods.

A web container is usually built into a web server or installed as a plug-in component to a web server. According to the J2EE specification, all web containers must provide support for HTTP protocol, almost all of them support HTTPS protocol too, to provide a security connection between client and container.

5.2 EJB Container

An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans and acts as an interface between enterprise beans

and other application components that require service of enterprise beans. The EJB container generates the instance of enterprise beans and manages their Lifecycle. The instances are stored in an instance pool maintained in the EJB container at run time. When application components need the services of an enterprise bean, the EJB container provides them an instance from the instance pool. An EJB container can manage many enterprise beans that are maintained in different EJB modules.

The EJB container provides many services to the enterprise bean, including the following [19] :

- Beginning, committing, and rolling back transactions as necessary.
- Maintaining pools of enterprise bean instances ready for incoming requests and moving these instances between the inactive pools and an active state, ensuring that threading conditions within the bean are satisfied.
- Most importantly, automatically synchronizing data in an entity bean's instance variables with corresponding data items stored in persistent storage.

5.3 WebSphere Application Server

WebSphere Application Server (WAS) is the IBM runtime platform for java-based applications, it provides an environment to run Web-based On Demand Business applications.

The core of WebSphere Application Server consists of the servlet engine and EJB container that conforms to the J2EE 1.2, 1.3, and 1.4 specifications, at the same time, WebSphere Application Server provides management of various services, such as database connection management, authentication and authorization. Additionally, a WebSphere Application Server can be seen as a basic version of other IBM servers[20], e.g. IBM MQSeries [21] that provides Java Message Service [22] .

WebSphere Application Server version 6.1 is Java EE 1.4 compliant application server [20] and supports Java standard edition 1.5. With installation of EJB feature pack, WAS6.1 provides support for EJB 3.0.

WebSphere Application Server can be installed on a wide range of operating systems. Table 5.3-1 lists the systems that WebSphere Application Server version 6.1 supports.

Operating Systems	**Version**
Windows	Windows server 2000 Windows server 2003 Windows XP Professional with SP2
Linux	Red Hat Enterprise Linux AS V3 with Update 5 or 6 Red Hat Enterprise Linux AS V4 with Update 2 SUSE Linux Enterprise Server V9 with SP2 or 3
IBM i5/OS and OS/400	i5/OS and OS/400, V5R3 i5/OS V5R4
z/OS	z/OS 1.6 or later
IBM AIX® 5L™	AIX 5L Version 5.2 Maintenance Level 5200-07 AIX 5L Version 5.3 with Service Pack 5300-04-01
Sun™ Solaris™	Solaris 9 with the latest patch Cluster Solaris 10 with the latest patch Cluster
HP-UX	HP-UX 11iv2 (11.23) with the latest Quality Pack

Table 5.3-1 WAS supported operating system

WebSphere Application Server provides great flexibility to build scale various server system based on business requirement.

It can runs as a stand-alone server using default port 9060 to provide a simple service. For stand-alone, each server holds its own management and acts as a unique entity. More stand-alone servers can be installed on one same machine with different ports or on different machines with same port to build a full business resolution, each server provides its own

services. These servers work together with an HTTP server that provides the uniform address and port to client and delivers the requests to the corresponding server. Table 5.3-2 shows the list of HTTP servers that Websphere Application Server v6.1 supports. However, in this case, WebSphere Application Server does not provide centralized management or administration for multiple stand-alone application servers.

HTTP Server
Apache HTTP Server 2.0.54
IBM HTTP Server for WebSphere application Server 6.0.2
IBM HTTP Server for WebSphere application Server 6.1
Internet Information Services 5.0
Internet Information Services 6.0
IBM Lotus® Domino® Enterprise Server 6.5.4 or 7.0
Sun Java™ System Web Server 6.0 SP9
Sun Java System Web Server 6.1 SP3

Table 5.3-2 HTTP Server

Further more, WebSphere Application Server supports network deployment to build a distributed server. In this environment, there's a central administration to manage the workload and server failover[24].

More about WebSphere Application Server you can find at [4][16] [20][21].

5.4 Resource Configuration on WebSphere Application Server 6.1

WAS6.1 provides a GUI Tool using brower to manage and configure the server, so that the managers can manipulate the server from remote places.

5.4.1 Datasource Configuration

WAS can manage the database connection and provide a data source to other component. The developer must not take care of the database connection at runtime. The benefit is that the username and password for the database are reserved on application server, must not be exposed to

other components, such as JSP pages.

Each data source must have a JDBC Provider that declares the JDBC driver type and connection type. For example, we want to create a DB2 data source named jdbc/db2 which connects to a DB2 database running on binks.informatik.uni-leipzig.de that provides S1D931 as database name and 4919 port to remote client.

As first step, we must define a new JDBC provider that uses db2 driver. shown in figure 5.4.1-1

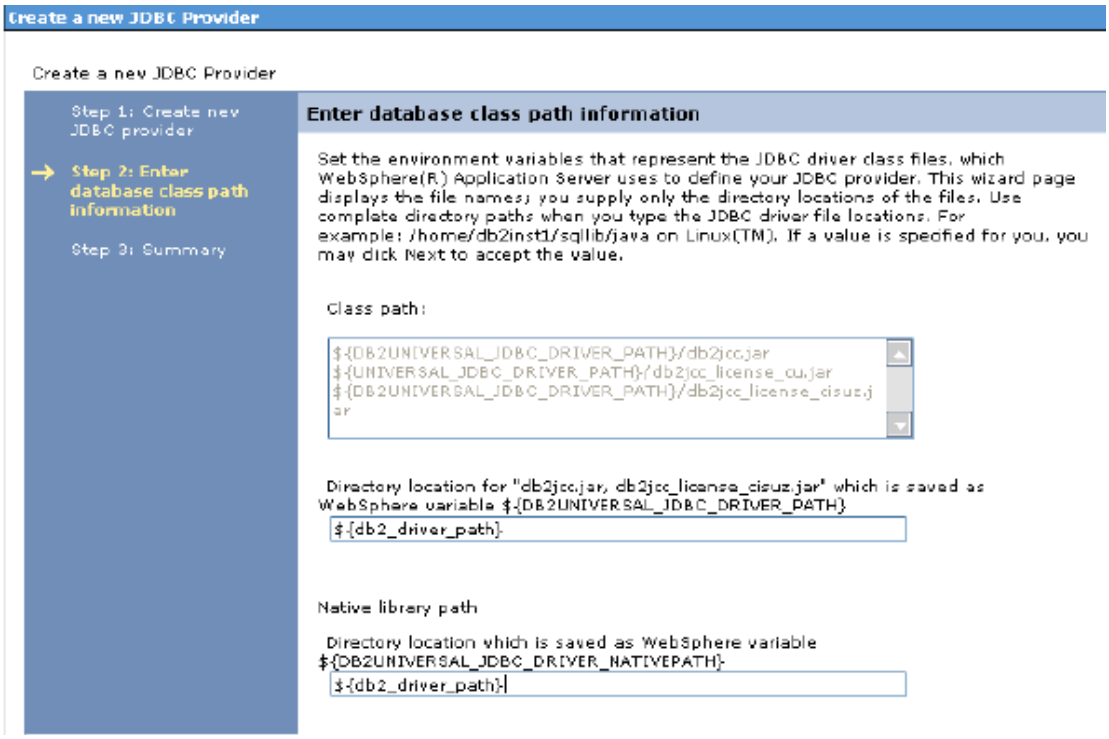


Figure 5.4.1-1 creating a new JDBC provider

Then we can create the datasource that uses db2 JDBC provider, the step is shown in figure 5.4.1-2.

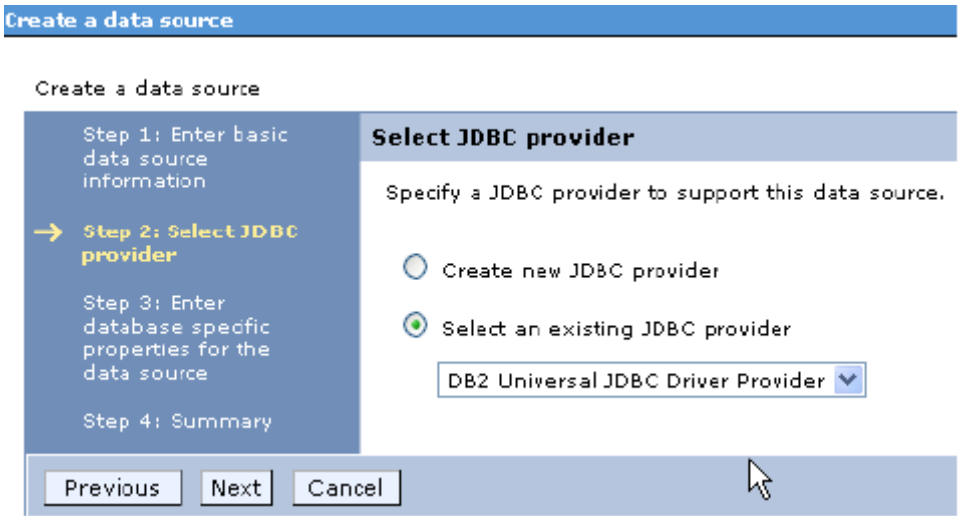


Figure 5.4.1-2 creating data source

After choosing the JDBC provider, we must enter the necessary information that is required for the connection.

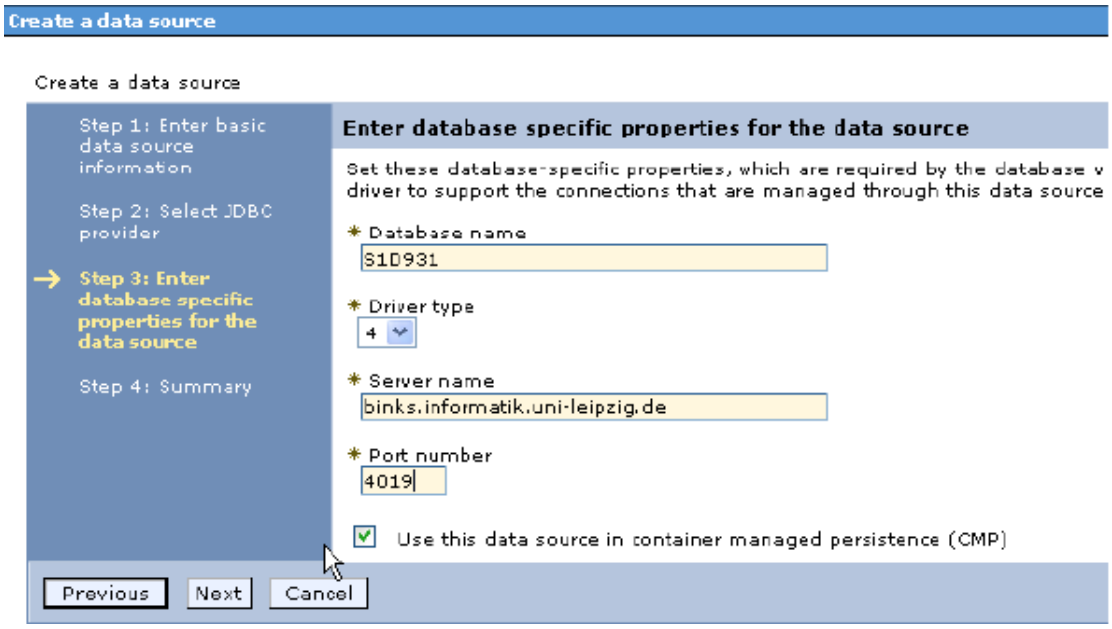


Figure 5.4.1-3 creating a data source

At last, we add 2 new customer properties *user* and *password* that contains the username and password respectively.

The specific steps can be found at[]

5.4.2 JMS Configuration

This Configuraton is divided into following steps.

At first we create a service integration bus (SIB) named **MDBSIBus** that is a runtime environment for JMS resources.(shown in figure 5.4.2-1)

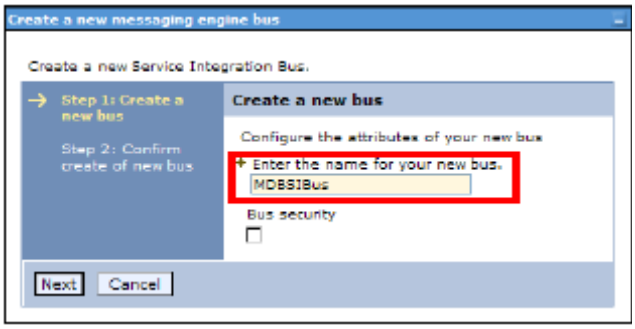


Figure 5.4.2-1 Creating Service Integration Bus(SIB)

Then we add member to the bus and specify which application server will host the messaging engine of the bus. The third step is the creation of a destination, for the project of this thesis, we create a Queue named **MDBQueue** as the destination. (see figure 5.4.2-2)

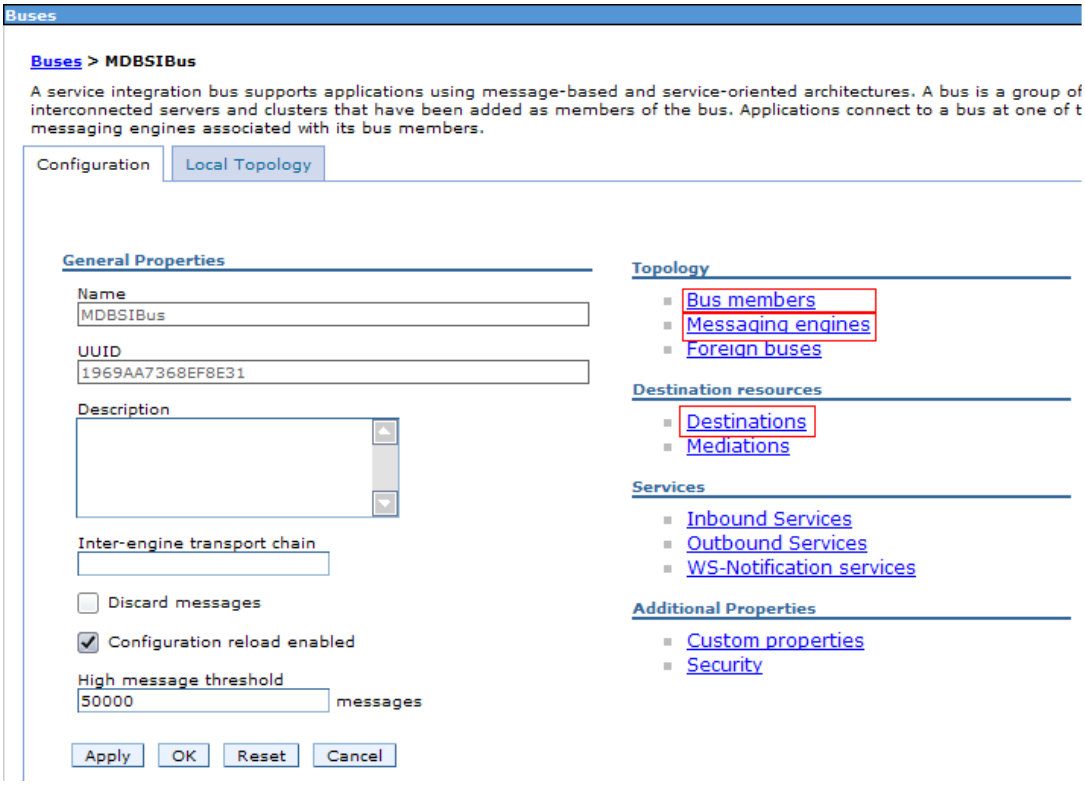


Figure 5.4.2-2 setting properties of SIB

In addition to the service integration bus, we have to configure the JMS provider and define a queue connection factory and a queue that match the JNDI names used in the servlet. At last, we have to define an activation specification that matches the MDB (shown in figure 5.4.2-3). A *Queue connection factory* defines which port and protocol are used to generate or receive a message. We create a new *Queue connection factory* with JNDI

name **jms/messageQueueCF**. In *Queues* section we create a new Queue with JNDI name **jms/messageQueue**, this queue points to the **MDBQueue** contained in **MDBSIBus**. In *Activation specification* section we create a new item with JNDI name **jms/mdbQueueActivationSpec** that holds a Queue with JNDI name **jms/messageQueue** as destination and **MDBSIBus** for bus name.

Figure 5.4.2-3 Configuration of Default messaging provider

The complete steps can be found at the additional tutorial <<JMS and MDBs>>.

6 J2EE Application Architecture

6.1 Enterprise Application

A J2EE application project contains the hierarchy of resources that are required to deploy a J2EE enterprise application, often referred to as an EAR file. A typical J2EE Application consists always of following J2EE modules and Java projects: Web modules, EJB modules, application client modules, connector modules, general utility Java JAR files, and EJB client JAR files. Figure 6.1-1 illustrates a sample of J2EE application architecture.

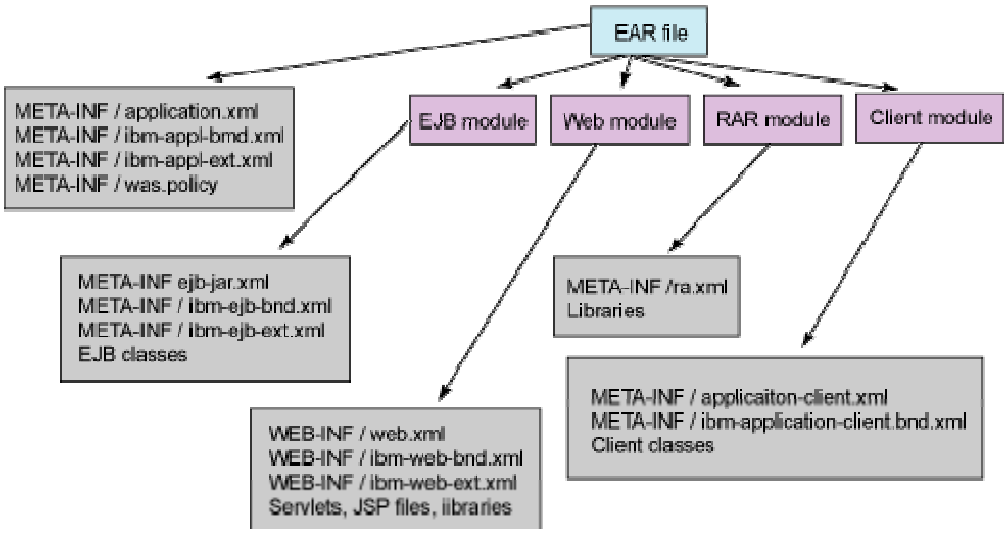


Figure 6.1-1 J2EE application archive structure[13]

The application.xml file is important and absolutely necessary for a J2EE application. This file is the deployment descriptor for the enterprise application, as defined in the J2EE specification, that is responsible for associating J2EE modules to a specific EAR file. Figure 6.1-2 illustrates a piece of deployment descriptor for the PlantShop project that contains 2 modules: PlantShopEJB.jar and PlantShopWebTest.war, where PlantShopEJB.jar is the archive file for EJB project bracketed by <ejb> attribute; PlantShopWebTest.war is the archive file for web project bracketed by <web> attribute. The PlantShop element in <context-root> attribute defines the web entrance for the web module, users can use a

web browser by typing http://host:port/PlantShop/ on address bar to visit the website.

```
<?xml version="1.0" encoding="UTF-8"?>
<application id="Application_ID" version="5"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/application_5.xsd">

 <display-name> PlantShopEAR</display-name>

<module>
  <ejb>PlantShopEJB.jar</ejb>
 </module>

 <module>
  <web>
        <web-uri>PlantShopWebTest.war</web-uri>
        <context-root>PlantShop</context-root>
  </web>
 </module>

</application>
```

Figure 6.1-2 application.xml

6.2 Web module

A Web module represents a web(dynamic or static) application. A Web module consists of assembling servlets, JSP files, and static content such as HTML pages, javascript files. Web modules are stored in Web archive (WAR) files, which are standard Java archive files.

There is a special file named web.xml stored in the /WEB-INF/ directory in the WAR file. It contains the web application's contents that declare the structure and resource dependencies of web components in the module and describe how the components are used at run time.

Figure 6.2 illustrates a sample web.xml file that demonstrates how servlets and resource reference are declared.

```
<?xml version="1.0" encoding="UTF-8"?>
……
```

```
        <display-name>PlantShopWebTest</display-name>

1       <servlet>
2               <description>
3               </description>
4               <display-name>
5               ShoppingServlet</display-name>
6               <servlet-name>ShoppingServlet</servlet-name>
7               <servlet-class>
8
        com.ibm.websphere.samples.pbwweb.ShoppingServlet</servlet-class>
9       </servlet>
10      <servlet-mapping>
11              <servlet-name>ShoppingServlet</servlet-name>
12              <url-pattern>/servlet/ShoppingServlet</url-pattern>
13      </servlet-mapping>

14      <welcome-file-list>
16              <welcome-file>index.html</welcome-file>
16              <welcome-file>index.htm</welcome-file>
17              <welcome-file>index.jsp</welcome-file>
18              <welcome-file>default.html</welcome-file>
19              <welcome-file>default.htm</welcome-file>
20              <welcome-file>default.jsp</welcome-file>
21      </welcome-file-list>

22      <ejb-local-ref id="EJBLocalRef_1271992031718">
23              <ejb-ref-name>Plant/Cart</ejb-ref-name>
24              <ejb-ref-type>Session</ejb-ref-type>
25              <local-home></local-home>
26              <local>com.ibm.websphere.samples.pbwejb.ShoppingCart</local>
27      </ejb-local-ref>

28      <resource-ref id="ResourceRef_1273113543749">
29              <description>
30              </description>
31              <res-ref-name>mailsender</res-ref-name>
32              <res-type>javax.mail.Session</res-type>
33              <res-auth>Container</res-auth>
34              <res-sharing-scope>Shareable</res-sharing-scope>
35      </resource-ref>
        </web-app>
```

Figure 6.2 web.xml

Lines 1 to 9 declare a serlvel named ShoppingSerlet which uses java class com.ibm.websphere.samples.pbwweb.ShoppingServlet specified in <servlet-class> attribute. Lines 10-13 specify the servlet's URL where the users can invoke the servlet.

Lines 14-21 declare the default entrance page of the website.

Lines 22-27 declare a session bean named Plant/Cart. This bean is a stateful session bean, because a stateful session bean must be looked up through JNDI. See the explanation in section 4.4-dependency injection.

Lines 28-35 define a resource reference named mailsender that is a mail session resource referring to a pre-defined resource on WebSphere Application Server.

6.3 EJB module

An EJB module contains enterprise java beans and other server-side components that are deployed on application server. The EJB module can also contain the ejb-client package and JPA project that can be stand alone modules if necessary.

The ejb-jar.xml file plays an important role in EJB2.x projects [1][4], it declares the enterprise java bean, relationships for entity bean and other resources. But in EJB3, the file is unnecessary because almost all resources can be accessed through meta annotations using default JNDI name. To provide developers more flexibility to define JNDI binding, EJB3 reserves the ejb-jar.xml file.

Another important deployment descriptor is the persistence.xml file that indicates which database is used for JPA entities at run time. Section 4.4 introduces how to define a JPA entity that uses universal setting to fit all databases which have the same database schemas and table structure. At run time, we must exactly indicate the relationships between JPA entities and databases to the application server.

The persistence.xml file is a standard configuration file in JPA. It has to be

included in the META-INF directory inside the JAR file that contains the entity beans. The persistence.xml file must define a persistence-unit with a unique name in the current scoped classloader.

```
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">

    <persistence-unit name="PBW" transaction-type="JTA">
        <jta-data-source>jdbc/db2</jta-data-source>
        <class>com.ibm.websphere.samples.pbwjpa.OrderItem</class>
        <class> com.ibm.websphere.samples.pbwjpa.Order</class>
        <class>com.ibm.websphere.samples.pbwjpa.Customer</class>
    </persistence-unit>

    <persistence-unit name="ITSO" transaction-type="JTA">
        <jta-data-source>jdbc/derby</jta-data-source>
        <class>itso.band.customer</class>
    </persistence-unit>
```

Table 6.3 persistence.xml

Table 6.3 gives a sample persistence.xml file where 2 persistence units named PBW and ITSO respectively are defined. The PBW unit has 3 JPA entities that use jdbc/db2 data source at runtime. The ITSO unit has only one JPA entity that connects to jdbc/derby data source at runtime. Both of the 2 persistence units use JTA data source.

7 Projects

For this thesis, 5 projects: IBMWeb Basic Servlets, Xtreme Travel, ITSO bank, PlantShop and MDB Demo, are deployed on WAS6.1. Figure 7-1 shows the home page of the projects.

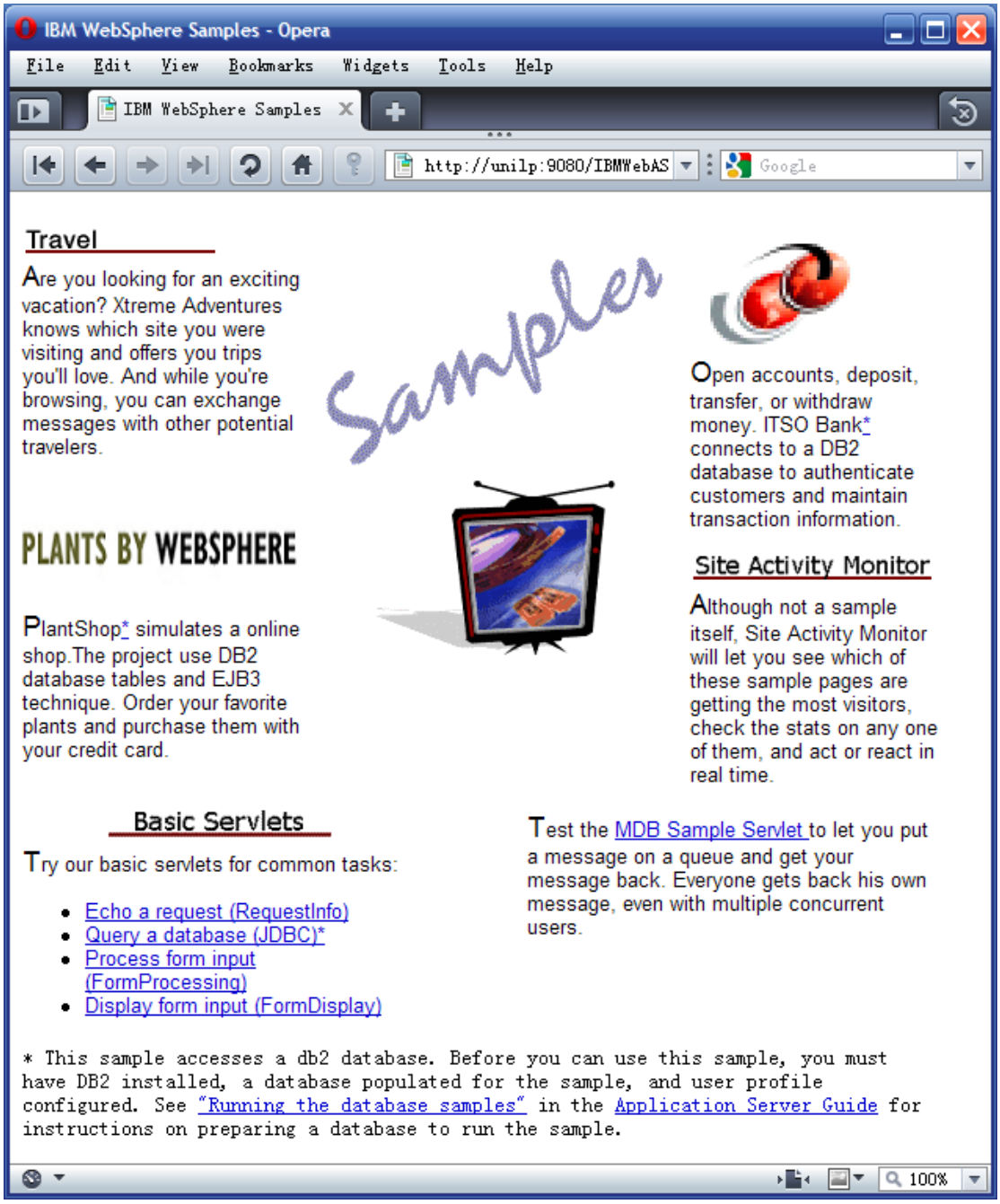


Figure 7-1 Projects

7.1 Basic Servlets

This project demonstrates the invoking of servlets that are samples in IBM Redbook [25].

7.1.1 ReqInfoServlet

As a simple servlet ReqInfoServlet shows the Http Headers and Request parameters, see Figure 7.1.1 This servlet invokes a java class named sample.class as Resourcebundle in the initiation phase. The sample.class contains the characters of various languages, so that a servlet can localize the output to adapt the user's language setting.

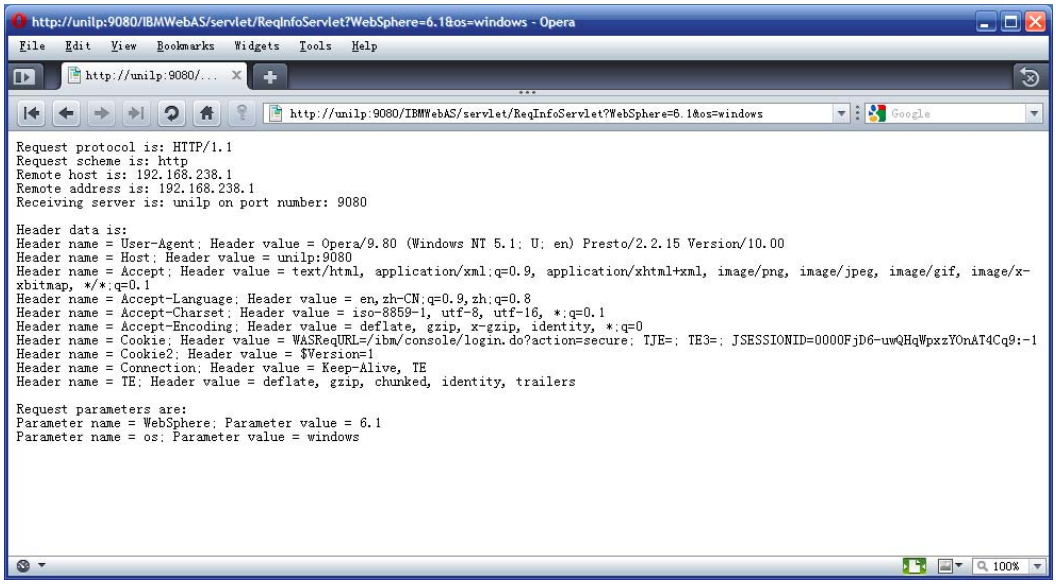


Figure 7.1.1 Output of ReqInfoServlet

Table 7.1.1 lists the methods that are used in the servlet to acquire the Http Headers.

Method	Description
getProtocol()	**Returns the name and version of the protocol the request uses in the form**
getScheme()	**Returns the name of the scheme used to make this request, for example, http, https, or ftp.**
getRemoteHost()	**Returns the fully qualified name of the client or the last proxy that sent the request**
getRemoteAddr()	**Returns the Internet Protocol (IP) address of the client or last proxy that sent the request**
getServerName()	**Returns the host name**
getServerPort()	**Returns the port number**
getHeaderNames()	**Returns an enumeration of all the header names this request contains**
getHeader(<Variable>)	**Returns the value of the specified request header**
getParameterNames()	**Returns an Enumeration of String objects containing**

	the names of the parameters contained in this request
getParameterValues(<Parameter>)	**Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist**

Table 7.1.1 Methods of HttpServletRequest

7.1.2 FormDisplayServlet

FormDisplayServlet is also a simple servlet. It outputs the request parameters and values contained in a HTML Form that uses post as method type, so the doPost() method in servlet is invoked to respond the request. This sample is shown in figure 7.1.2.

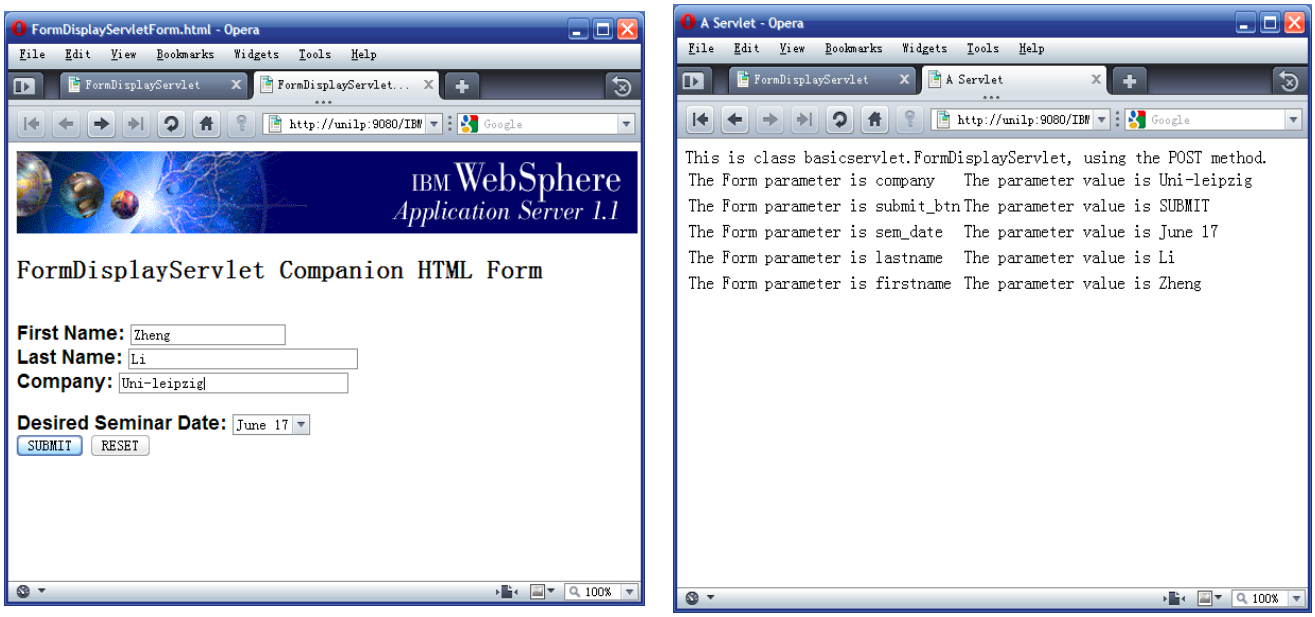


Figure 7.1.2 Input (left) and Output(right) page of FormDisplayServlet

7.1.3 FormProcessingServlet

The FormProcessingServlet is an extension of FormDisplayServlet. It verifies the completeness of the data, then saves the correct data to the file named seminar.txt. The seminar.txt must locate at the root path of the project.

7.1.4 JDBCServlet

JDBCServlet demonstrates a 3-tier web application to retrieve department and employee recorders that are stored in a DB2 database.

In initiation phase the servlet performs the init() method to create a datasource that connects to the database and to create 2 connections that are stored in a connection pool. Against the description in Redbook, this servlet does not read the login.property file to get the username and password for the database, it uses look-up through JNDI to get the datasource pre-defined in WAS6.1 (see section 5.4.1). The below code shows the new method.

```
try {
                    Hashtable parms = new Hashtable();
                    parms.put(Context.INITIAL_CONTEXT_FACTORY,

        "com.ibm.websphere.naming.WsnInitialContextFactory");
                    InitialContext ctx = new InitialContext(parms);
                    ds = (DataSource) ctx.lookup("jdbc/db2");
                    ……..
            } catch (Exception e) {
                    throw new BasicServletException("Can't connect to
        Database.");
            }
```

Table 7.1.4-1 Look up datasource through JNDI

The connections in this servlet are never closed in the service phase. servlet lifecycle. The connection pool manages the connections through putCon() and getCon() methods. When the getCon() method is invoked, the connection pool provides a connection from the pool, if the pool is empty, it creatse a new connection to furnish. In the putCon() method the pool checks whether the pool is full, if the pool is full, the connection will be

closed and released, elsewise the connection is placed in the pool.

In the destroy phase, all connections will be closed.

Beside the pre-prepared statement, user can also give his own query. Figure 7.1.4 illustrates an example.

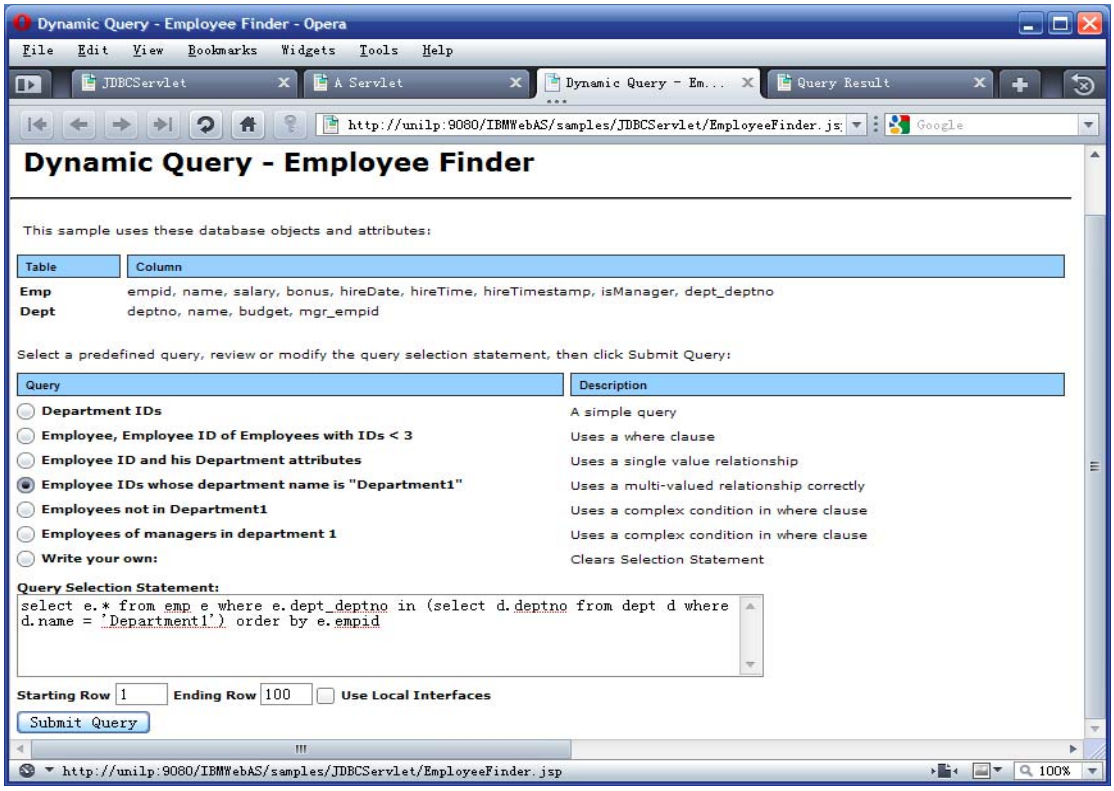
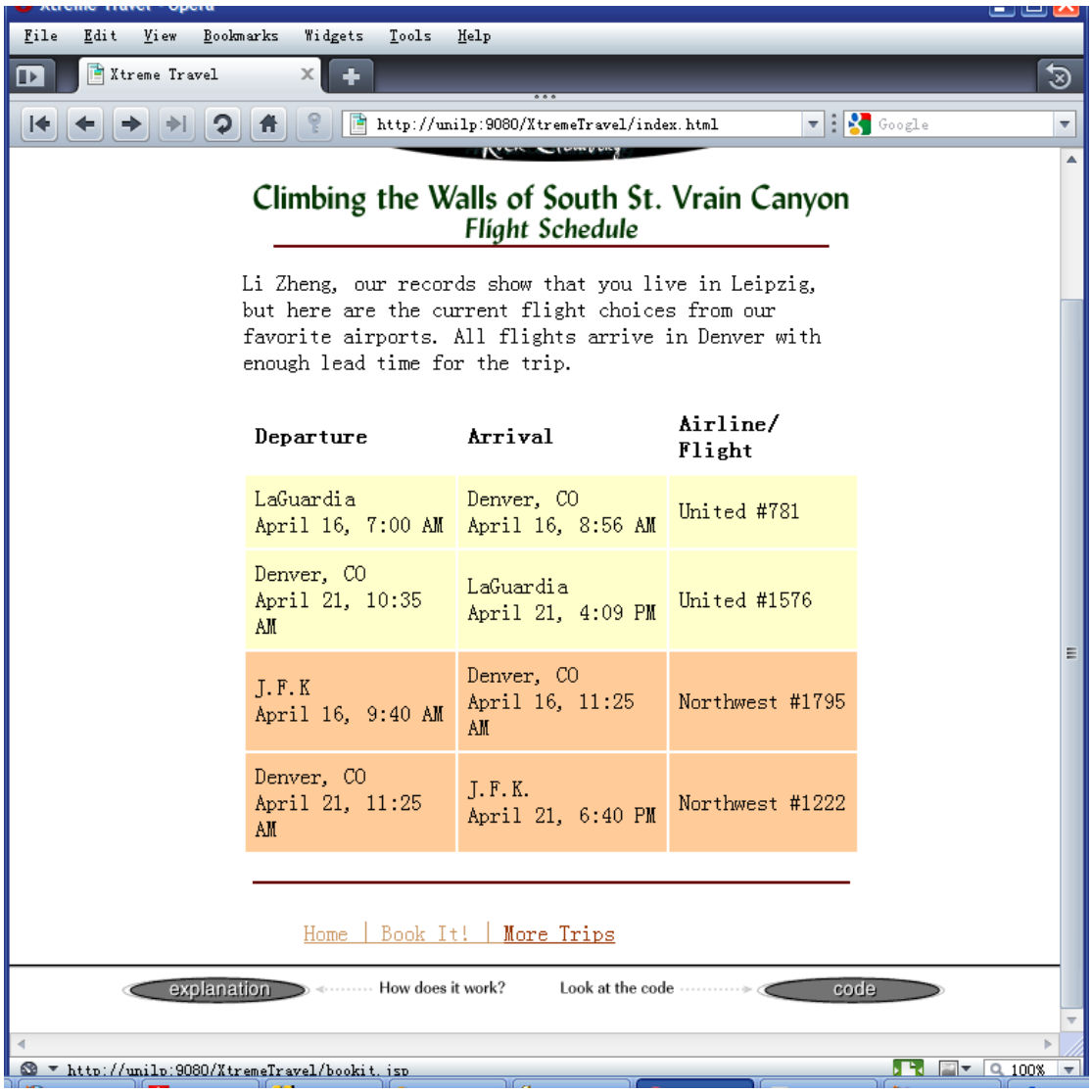


Figure 7.1.4 Example for JDBCServlet

7.2 Xtremel Travel

The Xtreme Travel application is a JSP application to demonstrate that a company offers extreme travel for different interest groups. The application guides the visitors step by step to find their interest travel. The visitors can

also book a trip on the website. After booking a trip, the visitor receives a message from the web site, that message shows the contact information of other users that have booked the same trip. This application uses a java bean named XTbean to store the user data in the session. Figure 7.2 shows the booking page, the user's name and residence are shown in the first line.



This project is also a sample in the IBM Red Book <<OS/390 e-business Infrastructure: IBM WebSphere Application Server 1.2>>. In the original project the application invokes the servlets in the com.ibm.servlet.servlets.personalization.util package to check and send messages, but I can't find the package in IBM Download Center. So the application can only complete the Booking operation. I guess, the

package does not exist today, because it is used for WebSphere 1.2 that is published years ago.

7.3 ITSO Bank

The ITSO Bank project is an extension of WOMBank project that simulates an on-line banking system where customers can register, open new account, check and execute transactions.

The WOMBank project is already implemented by Mr. Ronneburger[] and Mr. Kumke[] using servlet and EJB2.x respectively. This time, I use EJB3 to rewrite the project. The old WOMBank project uses the 4-tier model, all layout-setting are written in the servlets. But changing layout in servlets is a boring error-prone work, so I abandon the layout in WOMBank and rebuild the ITSO Bank to a 5-tier model. If it is necessary, the layout can be easily changed to fit the user's requirement. Figure 7.3-1 illustrates the application.

This project is just a simple demo for an on-line banking system, because the project is not secure. It uses http protocol, so that all messages are transported as plain text. The project has other shortages too. For example, a customer can create numberless accounts, this is unallowed in an on-line banking system.

The project consists of 2 modules: EJB3BankEJB.jar and EJB3BankBasicWeb.war.

EJB3BankEJB contains 3 JPA entities and a stateless session bean shown in table 7.3-1. The relationship between Account and Customer is many-to-many, the relationship between Account and Transactions is one-to-many. As a stateless session bean in EJB, the EJB3BankBean implements only one interface - EJB3BankService that defines the business methods, such as addCustomer, closeAccount ,deleteCustomer and so on.

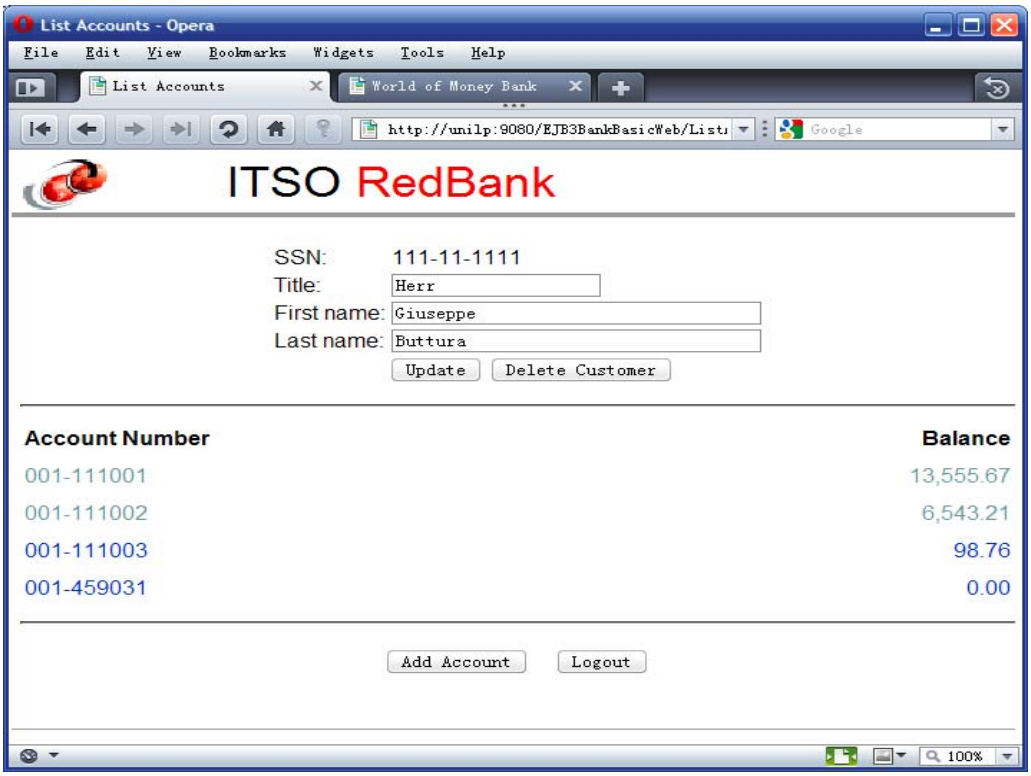


Figure 7.3-1 ITSO Bank

Bean	Type
Account	JPA Entity
Customer	JPA Entity
Transations	JPA Entity
EJB3bankBean	Stateless session bean

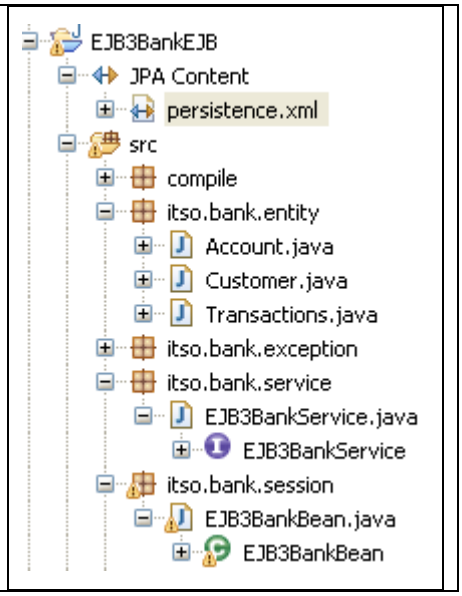


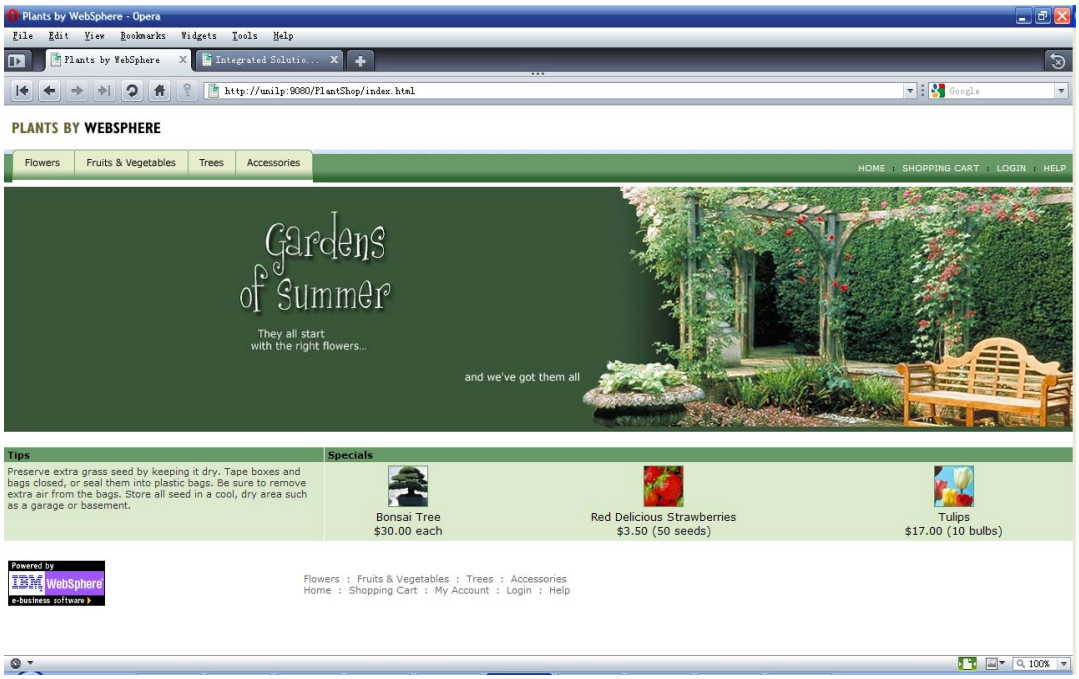
Table 7.3-1 Beans in ITSO Bank

EJB3BankBasicWeb is the web module that contains JSP pages and servlets.

7.4 PlantShop

The PlantShop project is a 5-tier web application written with EJB3 to demonstrate an on-line shop that sells plants and gardening tools as the project name suggests. On the website, customers can open accounts,

browse for items to purchase, view product details, and place orders. The project uses JPA entities, stateless session beans, a stateful session bean, JSP pages, and servlets to build a 5-tier model. The project is one of the WAS6.1 samples, but written with EJB2.x technology. Figure 7.4 shows the homepage of the project.



The project consists of 3 modules: PlantShopEJB.jar, PlantShopWebTest.war and PlantUAR.jar.

The EJB module contains a stateful session bean – ShoppingCartBean. As its name suggested, the bean keeps up the items in the cart before checkout.

PlantShopWebTest.war is the web module. Because a stateful session bean is used in the project, we must configure the web.xml manually to add the JNDI name for the stateful session bean (see figure 6.2) .

PlantUAR.jar is the assistant module containing the exceptions and debug methods.

In the original version, after checkout, an Email containing the purchase information and the total cost should be sent to the customer via an in WAS6.1 predefined mail session. I don't implement the function, because

WAS6.1 supports neither gmail nor hotmail.

7.5 MDB Demo

This MDB Demo application is a simple order management application using message-driven bean, the original version locates at [26]. Figure 7.5 shows a page.

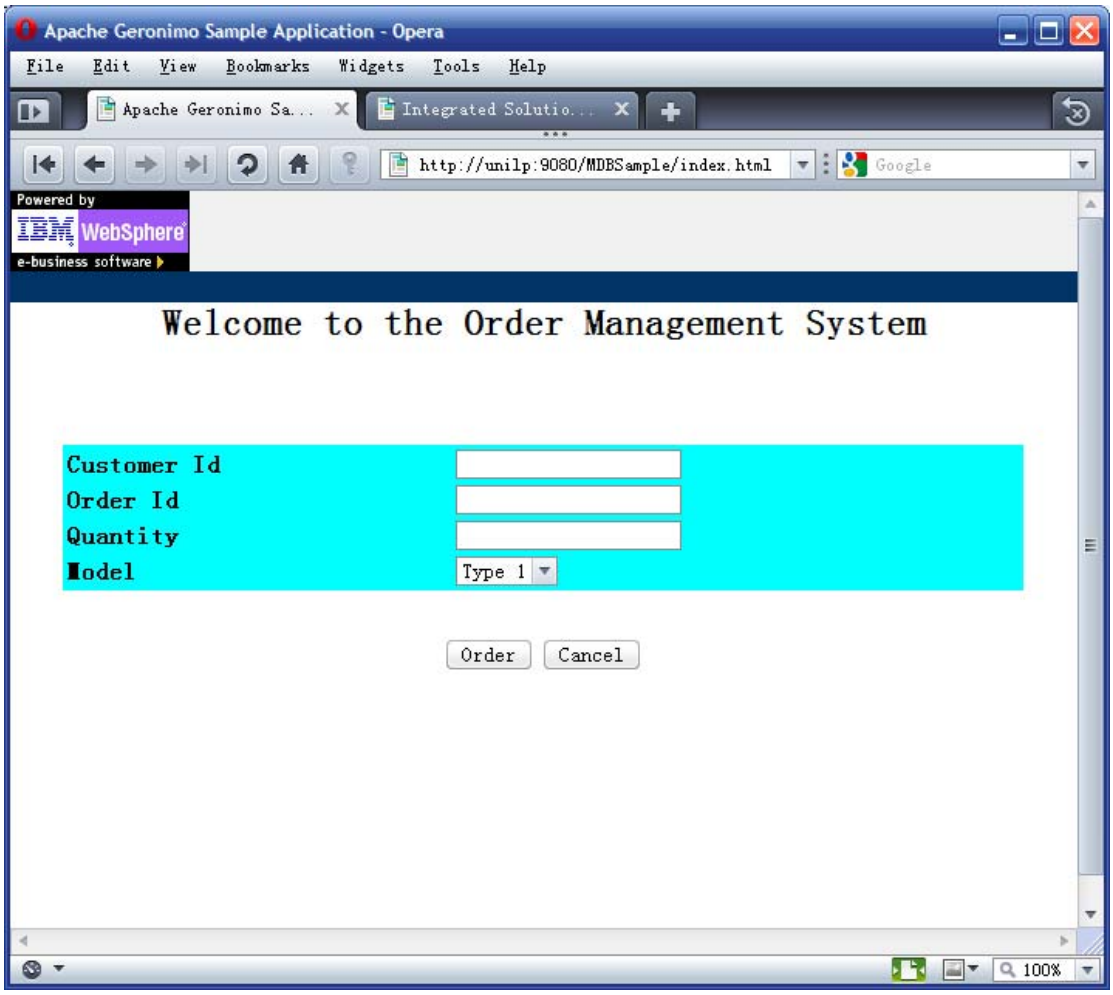


Figure 7.5 MDB Demo Application

This simple application contains 2 modules: MDBSampleEJB.jar and MDBSampleWar.war

As the EJB module, the MDBSampleEJB.jar contains a message-driven bean that listens a queue. So we must populate the ibm-ejb-jar-bnd.xml file to create a binding item for the message-driven bean. The below code shows the bind file. The message-driven bean is named **AsyncMessageConsumerBean** using **jms/mdbQueueActivationSpec** as activation specification and **jms/messageQueue** as destination. The

activation specification and destination are predefined resource in WAS6.1, see chapter 5.4.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar-bnd
        xmlns="http://websphere.ibm.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://websphere.ibm.com/xml/ns/javaee
http://websphere.ibm.com/xml/ns/javaee/ibm-ejb-jar-bnd_1_0.xsd"
        version="1.0">
    <message-driven name="AsyncMessageConsumerBean">
     <jca-adapter
            activation-spec-binding-name="jms/mdbQueueActivationSpec"
                destination-binding-name="jms/messageQueue"/>
     </message-driven>
    </ejb-jar-bnd>
```

Figure 7.5-2 ibm-ejb-jar-bnd.xml

The OrderSenderServlet in Web module uses a queue connection factory and a queue to send the order message to system. Both of the 2 resources are predefined in WAS6.1, we must configure the web.xml to make a reference to the resources, so that the servlet can use dependency injection instead of JNDI lookup to use the resources. Figure 7.5-3 illustrates the piece of the resource reference, Figure 7.5-4 shows the code piece for the dependency injection

```
<resource-ref id="ResourceRef_1265947627437">
        <description>
        </description>
        <res-ref-name>messageQueueCF</res-ref-name>
        <res-type>javax.jms.QueueConnectionFactory</res-type>
```

```
 <res-auth>Container</res-auth>
 <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>

<message-destination-ref id="MessageDestinationRef_1205878292921">
<message-destination-ref-name>MDBMessageQueue
</message-destination-ref-name>
<message-destination-type>javax.jms.Queue</message-destination-type>
<message-destination-usage>Produces</message-destination-usage>
</message-destination-ref>
```

Figure 7.5-3 web.xml

```
public class OrderSenderServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    @Resource(name = "messageQueueCF")
    private ConnectionFactory factory;

    @Resource(name = "MDBMessageQueue")
    private Queue receivingQueue;
```

Figure 7.5-4 code piece of OrderSenderServlet

8 Conclusion

EJB3 and WAS are both large topics, this document can only introduce the basis concept and features of EJB3 and the basic usage of WAS6.1 that are used for the master thesis. Other important topics, such as Web Services, authentication and authorization, can be found at [27][28] and implemented by other students as future work.

Beside the document, 3 tutorials, introducing how to step by step to develop EJB3 project, are on the CD.

Reference

1 IBM Red Book: WebSphere Application Server V6.1: Planning and Design URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247305.pdf

2 Ralf Ronneburger Diplomarbeit: Internet-basierte Anwendungen mit Java und DB2 unter OS/390

3 Thomas Kumke Diplomarbeit: Untersuchungen von Webanwendungen auf der Basis der J2EE-Umgebung unter z/OS

4 IBM Red Book: Rational Application Developer V7.5 Programming Guide URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247305.pdf

5 Patrick killelea: Web Perfomance Tuing, O'Reilly Media, October 1998

6 Tomcat Homepage: URL http://tomcat.apache.org

7 WebLogic Homepage: URL http://www.oracle.com/weblogic/

8 SunGlassFish Homepage: URL http://java.sun.com/javaee/community/glassfish/index.jsp

9 WebSphere Application Server URL http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.ejbfep.multiplatform.doc/info/welcome_nd.html

10 Gary B. Shelly, Harry J. Rosenblattt: Systems Analysis and Design, Course Technology, Jan 2001

11 Qusay H. Mahmoud: Portability Verification of Applications for the J2EE Platform, URL http://java.sun.com/developer/technicalArticles/J2EE/portability/

12 J2EE Platform Enterprise Edition Specification, v1.4 URL http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

13 J2EE5 Tutorial URL http://download.oracle.com/docs/cd/E17477_01/javaee/5/tutorial/doc/javaeetutorial5.pdf

14 Mark Wutka, Alan Moffet, Kunal Mittal: Sams Teach Yourself JavaServer Pages 2.0 with Apache Tomcat in 24 Hours, Complete Starter Kit, Sams, Dec 2003

15 Java Enterprise Community Homepage URL http://community.java.net/java-enterprise

16 IBM Red Book: Experience JEE! Using Rational Application Developer V7.5
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247827.pdf

17 Richard Monson-Haefel, Bill Burke: Enterprise JavaBeans 3.0 5th Edition, O'Reilly, May 2006

18 IBM Red Book: Experience J2EE! Using WebSphere Application Server V6.1 URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247297.pdf

19 IBM online Book: URL http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/cejb_ecnt.html

20 IBM Red Book: WebSphere Solution Bundles: Implementation and Integration Guide
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246550.pdf

21 IBM Red Book: WebSphere Application Server and WebSphere MQ Family Integration
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246878.pdf

22 Kareem Yusuf: Enterprise Messaging Using JMS and IBM WebSphere, Prentice Hall PTR, Feb 2004

23 IBM Red Book: Rational Application Developer V7.5 Programming Guide
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247672.pdf

24 IBM Red Book: WebSphere Application Servers: Standard and Advanced Editions
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg245460.pdf

25 IBM Red Book: OS/390 e-business Infrastructure: IBM WebSphere Application Server 1.2
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg245604.pdf

26 IBM Red Book: Application Server Version 6.1 Feature Pack for EJB 3.0
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247611.pdf

27 IBM Red Book: Web Services Feature Pack for WebSphere Application Server V6.1
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg247618.pdf

28 IBM Red Book: IBM WebSphere Application Server V6.1 Security Handbook
URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246316.pdf

Abbreviations and acronyms

API application programming interface
BMP bean managed persistence
CMP container managed persistence
EJB Enterprise JavaBeans
HTML Hypertext Markup Language environment
J2EE Java 2, Enterprise Edition
JDBC Java Database Connectivity
JMS Java Message Service
JNDI Java Naming and Directory Interface
JPQL JPA query language
JSP JavaServer Pages
JSR Java Specification Request
JVM Java Virtual Machine
MDB message-driven bean
JPA Java Persistence Architecture
POJI plain old Java interface
POJO plain old Java object
RAR resource archive
RMI Remote Method Invocation
SIB service integration bus
URL uniform resource locator
WAR Web archive
XML eXtended Markup Language