SIEBEL INTERACTIVE SELLING TRANSACT SERVER INTERFACE REFERENCE

VERSION 7.0, REV. I

12-BD4FKR

JANUARY 2003

Contents

Introduction

This guide provides information, instructions, and guidelines for installing Siebel Transact™ and its related components on a Microsoft Windows or Solaris host. The guide is useful primarily to individuals whose title or job description matches one of the following:

Web Server Administrators	Individuals responsible for converting Transact XML order format to Order Management System order format.
Database Administrators	Individuals who administer the database system, including data loading, system monitoring, backup, and recovery, space allocation and sizing, and user account management.
Integration Engineers	Individuals responsible for converting Transact XML order format to Order Management System order format.
Siebel Application Administrators	Individuals responsible for planning, setting up, and maintaining Siebel applications.
Siebel Application Developers	Individuals who plan, implement, and configure Siebel applications, possibly adding new functionality.
Siebel System Administrators	Individuals responsible for the whole system, including installing, maintaining, and upgrading Siebel applications.
Application Server Administrators	Individuals responsible for setting up and maintaining Transact, and enabling integration between Transact and the Order Management System.

How This Guide Is Organized

The opening chapter of the guide describes the Transact architecture and summarizes the five steps required to install and implement Transact. The second chapter describes how to install Transact and the subsequent chapters show how to implement Transact.

The guide contains six chapters and several appendices:

- Chapter 1, "Overview," describes how the Transact Server provides the integration points that enable the exchange of data between a Siebel application and a third-party Order Management System. This chapter also describes the major components in the operating environment. Finally, this chapter describes how to implement Siebel Transact in five steps.
- Chapter 2, "Installing Transact Server," describes the steps required to install and set up the Transact Server, and then how to set up the supporting databases and application servers.
- Chapter 3, "Working with Configurations," describes how to manage configurations and how to create the Add to Cart, View Order, Save Configuration, and View Configuration List buttons for a Siebel application.
- Chapter 4, "Integrating the Order Management System," describes the steps required to integrate a Siebel application with an Order Management System so that when you add a configuration to your shopping cart, Transact connects with the Order Management System.
- Chapter 5, "Authentication and Login Support," describes how to set up your authentication and login system using an LDAP directory server.
- Chapter 6, "The Shopping Cart," describes how to set the Shopping Cart properties and then insert Transact API calls into the browser-based application so items can be added to the shopping cart and the cart UI can be brought up on demand.
- The Appendices describe various specifics of the Transact Server and its operating environment.

Revision History

Siebel Interactive Selling Transact Server Interface Reference, Version 7.0, Rev. I

Overview 1

Siebel Transact provides integration points that enable the exchange of data between a Siebel application and a third-party Order Management System (OMS).

These integration points allow you to:

- Use a third-party system’s shopping cart.
- Simplify order submission to an OMS.
- Save and restore configurations.
- Share configurations.

The integration points are implemented with client-side and server-side components as well as tables in a relational database management system.

Architecture

Siebel Transact's client-side components reside in a Siebel application and are referred to as the "Siebel Transact Module," and Transact's server-side components reside on an application server and together are referred to as the "Transact Java Engine." The server-side components store information, such as configurations and configuration lists, in the Transact database (RDBMS).

As an example, the Siebel Transact Module could reside within Siebel eAdvisor, the Transact Java Engine could reside on a WebSphere application server hosted by a Solaris UNIX platform, and the RDBMS could be an Oracle instance.

The following sections describe the Transact architecture:

- The Data Flow
- The Client
- The Application Server
- The Remote Systems

The basic unit of data transferred is a configuration.

The Data Flow

The data flow to Transact begins when someone has determined a specific configuration of products using a Siebel application and clicks a button or link that calls the Siebel Transact Module. The Siebel Transact Module then passes the configuration data to the Transact Java Engine. The Java Engine stores the configuration in the Transact database, and then generates an XML or HTTP stream which it can then pass on to an Order Management System.

The Client

The Siebel Transact Module is embedded within the Siebel client and communicates with the Transact Java Engine using Transact API function calls. These calls are included in the onClick event of buttons or links which are specific to the Siebel Transact shopping cart. Part of integrating the Siebel application with the Order Management System is adding these buttons or links to the Siebel application UI.

You can add the following functions as buttons or links to the Siebel application UI:

- **Save Configuration** saves a record of the items selected from a page set, a “configuration.” Configurations are saved in a configuration list that resides in the Transact database and they can be viewed at a later time.
- **Add to Cart** adds the configuration to your shopping cart.
- **View Configuration List** displays a saved configuration. If you select a configuration from a configuration list, the Siebel application displays the page set and selections as they appeared at the time you saved the configuration.
- **View Cart** displays the contents of the shopping cart.
- **Submit Cart** transfers an order to the Order Management System.

When a user selects one of these buttons or links, the onClick event handler transmits the configuration data to the Transact Java Engine. Data is transmitted to the Transact Java Engine using HTTP and returns data to the application as an HTML/Javascript response.

The saved configurations reside in the Transact database.

The Application Server

The Transact Java Engine runs on an application server, and this server manages communications to the Transact database (RDBMS) and the Order Management System. The Transact Java Engine uses Enterprise Java Beans (EJBs) to store, retrieve, and pass configurations from the Siebel application to the Order Management System. EJBs are session and entity beans, distributed objects that encapsulate logic and data within the application server.

Depending on your system configuration, more than one Web server can be positioned between the application server and the Siebel application. If the application server supports clustering, the application server may span more than one machine.

The Transact Java Engine

The Transact Java engine consists of three layers: the Communication, Business Logic, and Data Object layers.

Communication Layer

This layer is a lightweight Java servlet that communicates with Enterprise Java Beans (EJBs). This servlet accepts requests from the browser, directs them to the correct EJB for processing, and then returns the resulting output to the browser or to a targeted Order Management System.

Business Logic Layer

This layer is a set of session EJBs that use the Data Object layer to create, modify, remove, and manipulate data objects. Most processing, including XML creation, occurs on this level because EJBs establish a distributed environment that improves processing. The XML produced on this level is sent to the Order Management System by the Communication layer as HTTP. At this point, the Order Management System controls the request and the subsequent response to the browser.

Data Object Layer

This layer is a set of entity EJBs based on a relational table design. The entity beans include data access and modification methods.

The Remote Systems

Siebel Transact accesses two remote systems: the Transact database (a RDBMS) and the Order Management System.

The RDBMS

The RDBMS contains relational tables that store the data mapped to the entity beans by the Data Object layer.

The Order Management System

The Order Management System manages the shopping cart UI—it receives XML from the Transact Java Engine and responds to browser requests from the Siebel application with HTML.

You must customize the Order Management System so that it is fully integrated with Transact. The Order Management System must accept and process the XML generated by the Transact Java Engine, and then respond by returning the appropriate HTML.

More than one Order Management System may be accepting submissions from a single Siebel application.

Implementation

Siebel Transact operates on several layers. The layers include the browser and application layer, the Web server layer, the database layer, and the Order Management System layer. Transact should thus be implemented by individuals who are responsible for and understand each of these levels. Because of the requisite job skills and responsibilities, more than one person may be required to complete the installation and implementation of Transact. This section summarizes the implementation tasks and organizes these tasks according to the individuals who, according to their job role, would be expected to complete each task.

This section also describes the major steps required to implement and take full advantage of Siebel Transact.

Implementation Tasks and Team

Most likely, several individuals with specific responsibilities and skills will be required to implement Siebel Transact. These tasks are summarized below and assigned to individuals according to their respective job roles:

- Application Server Administrators
- Database Administrators
- Integration Engineers
- Siebel Browser-Based Application Developers
- Siebel System Administrators

The discussion below summarizes each of these roles in terms of their responsibilities and technical skills. Each summary includes a list of specific tasks the individual performing the role would need to complete as well as a list of those sections in this guide most important to completing these tasks.

Application Server Administrator

The Application Server administrator is responsible for setting up and maintaining Siebel Transact. This individual should be familiar with their database and application server setup, and with the directory structure and configurable options of Siebel Transact. This individual works with the integration engineer to enable integration of the Siebel application with the Order Management System. The integration tasks include developing the XML style sheet and the JSP page.

The Application Server Administrator completes the following tasks:

Task	Topics to Read
Install and set up Siebel Transact	Chapter 2, "Installing Transact Server"
Set server-side options using the property editor.	"Setting the Transact Properties" on page 53
Work with the integration engineer to enable integration with the Order Management System.	"Integration Engineer" on page 23

Database Administrator

The Database Administrator is responsible for setting up and maintaining the Transact database. This database can be an Oracle, MSSQL, or DB2 database.

The Database Administrator completes the following tasks:

Task	Topics to Read
Set up the Oracle database.	Refer to the Oracle documentation.
Set up the MS SQL database.	Refer to the MS SQL documentation.
Set up the IBM DB2 database.	Refer to the IBM DB2 documentation.

Integration Engineer

The Integration engineer must be familiar with XML, XSLT, and the Order Management System requirements. This individual develops and writes customized code to reconcile the Siebel Transact and Order Management System XML formats and is constrained by the data elements available through the Siebel Transact XML format.

If the Order Management System does not accept XML, this individual must develop a JSP page for posting a form. Or, as another option, the engineer can develop a custom program that converts the XML output to text using one of the publicly available XML parsers to convert the Siebel XML into the desired text format.

The Integration Engineer completes the following tasks.

Task	Topics to Read
Define the XML/XSLT format accepted by the Order Management System.	"About XSLT" on page 91
If required, post a form from an Add to Cart link.	"Posting a Form from Add to Cart" on page 80
Set up linkback from your Order Management System to the Siebel browser-based application.	"Linking Back to Configurations" on page 102

Siebel Browser-Based Application Developer

The Siebel Application Developer is responsible for setting the Siebel Transact Server variables in app_config.js in order to connect the Siebel application to the Order Management System. This individual also adds buttons and links to the Siebel application UI.

The Siebel Application Developer completes the following tasks.

Task	Topics to Read
Set the Siebel Transact Server variables within the Siebel application.	"Setting the Transact Properties" on page 53
Add the Transact UI buttons to the Siebel application.	"Connecting Your Siebel Application UI to Transact" on page 76
Customize the Configuration List template.	"Working with the Configuration List" on page 62

Siebel System Administrator

The Siebel system administrator is responsible for setting up and maintaining the Siebel Application Server. This individual completes the following task:

Task	Topics to Read
Set up the Siebel System.	See your Siebel System Administration Guide documentation.

Implementation Steps

You can implement Siebel Transact in five major steps. Each chapter in this guide describes how to complete these steps.

To implement Siebel Transact

1. Install and set up Transact.

   See Chapter 2, "Installing Transact Server."

2. Set up the authentication and login system (optional).

   See Chapter 5, "Authentication and Login Support."

3. Create the Add to Cart, View Order, Save Configuration, and View Configuration List buttons in the Siebel application.

   See "Working with the Configuration List" on page 62.

4. Edit the ConfigList JSP page to call the appropriate Config List functions (modifies the Configuration list UI).

   See "Modifying the Configuration List UI" on page 63.

5. Create a style sheet for the DTD that determines how the configuration information is be passed to the Order Management System shopping cart.

   For more information, see "Using Style Sheets" on page 91.

Installing Transact Server 2

This chapter describes how to install and set up the application server environment for Transact server. The chapter describes how to prepare for the installation of Transact Server and how to install the server on an application server. The chapter continues with a discussion of how to configure the application server environment, the Transact database, and the Transact server so that it can connect with the Siebel application. The chapter closes with a discussion of how to troubleshoot problems that may occur during the installation process.

Prepare for the Installation

The installation process must conform to a few dependencies and be conducted by individuals who are responsible for and understand the various operational levels over which Siebel Transact operates. This section describes the dependencies and defines a set of terminological conventions that simplify the discussion of the implementation process. You should familiarize yourself with these conventions.

Dependencies

Before you can install the Transact Server, you must install:

- Application server
- Database
- Java Development Kit (JDK)
- LDAP Directory Server (*optional*)

  You can install the Transact Server with or without an LDAP directory server. If you do not plan to use LDAP login and authentication features, leave the LDAP properties blank when you install the Transact Server.
- Web server

For additional information on supported components, refer to *System Requirements and Supported Platforms*.

NOTE: If you do not plan to use LDAP login and authentication features, leave the LDAP properties blank when you install the Transact Server.

Installation Steps

You can install, set up, and implement the Siebel Transact Server in four steps.

To install Siebel Transact

1. Install and set up Transact on an application server.
2. Configure the environment on the application server.
3. Prepare the Application to Connect to Transact.
4. Troubleshoot Installation Problems.

Terminology

This guide uses a few conventions when referencing host platforms and installation directories. Familiarize yourself with these conventions before you install and set up Transact Server.

Generic Directories

This guide uses the following variables to generically name directories referenced during the installation and configuration process. You should familiarize yourself with this notation before you install and set up Transact Server.

Directory	Description
< Transact root >	Where you install Transact Server. Examples: ■ If you install Transact Server on BEA WebLogic, then the Transact root directory is `c:/sea700/transact_wl`. ■ If you install Transact Server on IBM WebSphere, then the Transact root directory is `c:/sea700/transact_ws`.
< WebSphere root >	Where WebSphere server is installed.
< IBM HTTP Server root >	Where the IBM HTTP server is installed.
< SQLLIB root >	Where the IBM DB2 client is installed.
< Windows root >	Where Windows NT/2000 is installed.

Install Transact on An Application Server

You can install Siebel Transact on a Windows or a Solaris platform if an application server, WebSphere or WebLogic, has already been installed.

NOTE: This guide references as "Windows" all Microsoft Windows operating systems listed as supported for this release in *System Requirements and Supported Platforms*. Similarly, "MS SQL Server" refers to the version of that database referenced in *System Requirements and Supported Platforms*.

Install Transact Server on Windows

You can install Transact Server on Windows in a few steps.

To install Transact Server on Windows

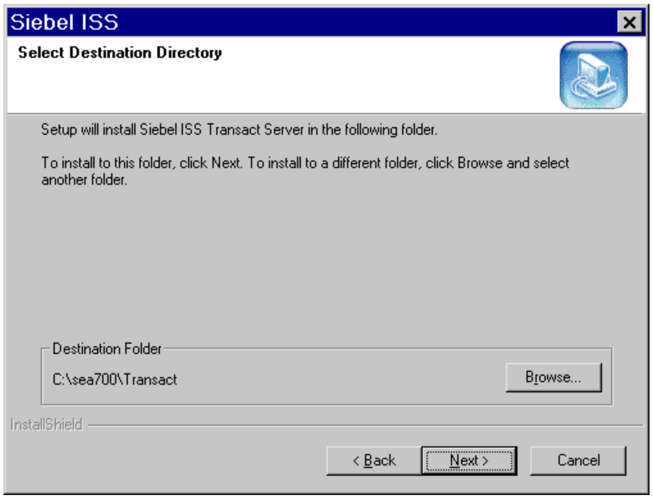
1. Execute install.exe.
   - Open install.exe from the isstransact_wl directory if you are using a WebLogic Application Server.
   - Open install.exe from the isstransact_ws directory if you are using a WebSphere Application Server.
2. Select which languages to install, and then click OK.

   The Welcome window appears.

   NOTE: Transact supports only English, so if you select another language the install program still copies the English (American) files.

   ---

3 Click Next and the Select Directory Destination window appears.



Siebel ISS

Select Destination Directory

Setup will install Siebel ISS Transact Server in the following folder.

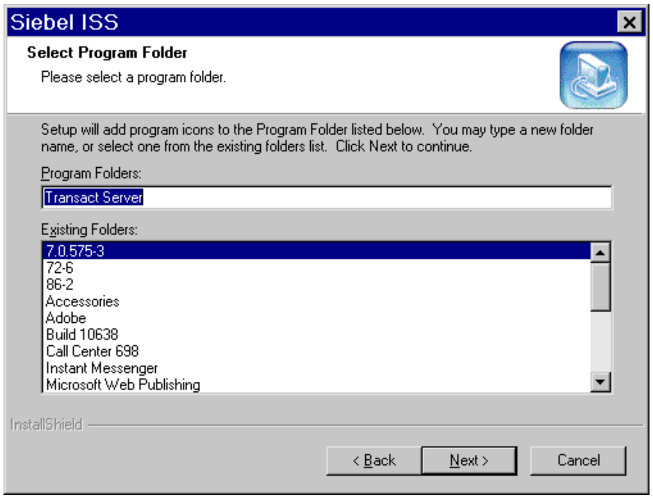
To install to this folder, click Next. To install to a different folder, click Browse and select another folder.

Destination Folder

C:\sea700\Transact

4 Rename the default folder if you want to change it, and then click Next.

After the Siebel Installer creates directories and copies files to the specified folder, it returns the Select Program Folder window.



Siebel ISS

Select Program Folder

Please select a program folder.

Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing folders list. Click Next to continue.

Program Folders:

Transact Server

Existing Folders:

7.0.575-3
72-6
86-2
Accessories
Adobe
Build 10638
Call Center 698
Instant Messenger
Microsoft Web Publishing

5. Select or enter a program folder, and then click Next.

   The Siebel Installer installs Transact Server.

6. After Transact Server is installed, enter environment variables in the Siebel Software Configuration Utility.

   The utility does not validate your input but only places it in the corresponding files.

7. If you enter an incorrect value for a variable or you want to edit the environment settings later, then execute:

   "<Transact root>/bin/ssincfgw.exe <Transact root>/admin/transact_wl.scm" (BEA WebLogic Application Server)

   "<Transact root>/bin/ssincfgw.exe <Transact root>/admin/transact_ws.scm" (IBM WebSphere Application Server)

Install Transact Server on Solaris

When you install Siebel Transact on Solaris, the installation directors vary according to whether WebLogic or WebSphere is the application server.

To install Transact Server on Solaris

1. Run install_transactWL (WebLogic Application Server) or install_transactWS (WebSphere Application Server) from the install directory.

   NOTE: Make sure you have not set SIEBEL_ROOT environment.

2. Enter the location where you want to install Transact Server.
3. Type Y to create a new installation directory (if it does not already exist).
4. Type Y to accept the setting for Transact Server.
5. Start the configuration setup.

To start the configuration setup

1. Enter the WebLogic or WebSphere Root directory.

   If the installer cannot locate the WebLogic directory, the setup will exit. Once you exit, you have to restart from step 1.

2. Type in the WebLogic or WebSphere Host name.
3. Type in the LDAP host name.
4. Type in the LDAP domain name.
5. Type in the eAdvisor URL.
6. Type in the SMTP host name.
7. Type in the Java Path.

   You can at a later time modify the items in Steps 3, 4, 5, and 6 using the property editor.

Configure the Application Server Environment

Before you can deploy Transact Server EJB, you must configure the environment, register Java beans, and create servlets. Follow the guidelines for your application server to complete these tasks.

For additional information on the respective application servers, refer to:

- BEA WebLogic Information:

  http://www.WebLogic.com/docs51

- IBM WebSphere Information:

  http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/index.html

NOTE: For a WebSphere installation that uses Oracle rather than DB2, refer also to http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/wasstd/022421.html.

The WebLogic Environment

To set up the WebLogic environment for Transact Server, you must complete these major tasks:

1. Set up the class and directory paths.
2. Modify WebLogic.Properties file.
3. Set up the Transact Database.
4. Set up the LDAP directory server (*optional*).

The discussion below shows you how to complete each of these tasks.

Setting Up the Class Path and Directory Path

The class and directory paths for the WebLogic environment can be completed in four steps.

To set up the class path and directory path

1 Copy three folders to <WebLogic root>/myserver/public_html:

- login
- propertieseditor
- transactserver under <transact root>/jsp

2 Set up the environment class path.

You can use the setenv script file provided by WebLogic. For additional information, refer to the WebLogic5.1 documentation:

http://www.WebLogic.com/docs51/install/startserver.html
http://www.WebLogic.com/docs51/admindocs/classpath.html

Use the following file names in the environment classpath:

- Append <transact root>/Javalib/rincon41.jar into classpath for setting up Rincon.

  Example:

  ```
  CLASSPATH=%JDK_CLASSES%……;<Transact root>/JavaLib/
  rincon41.jar
  ```

- Append ODBC driver.

  If you use SQL Server, add <SQL server root>/classes.
  If you use Oracle 8.0.5, add <WebLogic root>/bin/oci805_8.

3 Add Transact Server jar files to the WebLogic classpath.

You can do this using startWebLogic.cmd. For more information, refer to the WebLogic5.1 documentation at the following links:

http://www.WebLogic.com/docs51/install/startserver.html
http://www.WebLogic.com/docs51/admindocs/classpath.html

4 Append the following files to the WEBLOGIC_CLASSPATH parameter:

< Transact root > /Javalib/rincon41.jar

< Transact root > /Javalib/transact41.jar

< Transact root > /Javalib/xalan.jar

< Transact root > /Javalib/xerces.jar.

< Transact root > /Javalib/xml.jar.

< WebLogic root > /persistence/WebLogic_RDBMS.jar.

Modifying <WebLogic root>/WebLogic.Properties

According to the WebLogic administration guidelines, you must modify the following variables located in your weblog.properties file.

NOTE: If you have a clean WebLogic file, in which variables are commented out by the prefix “#,” remove the “#” sign before modifying the values.

Variable	Action	Value
WebLogic.ejb.deploy	Append	Add < Transact root > /Javalib/rincon41.jar < Transact root > /Javalib/transact41.jar
WebLogic.jdbc.connectionPool .OraclePool	Modify	Replicate all values related to WebLogic.jdbc.connectionPool.OraclePool. Change OraclePool to SiebelPool. The new variable name is WebLogic.jdbc.connectionPool.SiebelPool.
WebLogic.httpd.register.*.jsp		Make sure the “#” sign is removed.
WebLogic.httpd.initArgs.*.jsp		Make sure the “#” sign is removed.

Add these variables to the bottom of the WebLogic.properties file.

To set up the WebLogic the application server

1. Create servlets, SiebelTransact, LoginProcess, and Logout.
    - WebLogic.httpd.register.SiebelTransact = com.siebel.isscda.wl.servlet.mars.Re questHandler
    - WebLogic.httpd.register.LoginProcess = com.siebel.isscda.wl.rincon.LoginProc ess
    - WebLogic.httpd.register.Logout = com.siebel.isscda.wl.rincon.Logout
2. Set up a security reference for your WebLogic Application Server.
    - WebLogic.allow.reserve.WebLogic.jdbc.connectionPool.SiebelPool = everyone
    - WebLogic.allow.lookup.WebLogic.jndi.WebLogic = system
    - WebLogic.allow.lookup.WebLogic.jndi.WebLogic.ejb = system
    - WebLogic.allow.modify.WebLogic.jndi.WebLogic = system
    - WebLogic.allow.modify.WebLogic.jndi.WebLogic.ejb = system
    - WebLogic.allow.list.WebLogic.jndi.WebLogic = system
    - WebLogic.allow.list.WebLogic.jndi.WebLogic.ejb = system
3. Set up a data source for list quote and list configuration.

    Example:

    ```
    WebLogic.jdbc.DataSource.SiebelDataSource=SiebelPool where
    SiebelPool is the database pool you setup for Transact Server.
    ```

Setting Up the Database

Transact Server operates with Oracle and SQL 2000 JDBC drivers. You must set up the JDBC driver on your system with a user name, password, and service name.

Oracle

To create tables, execute < Transact root > /db/oracle/auth_orcl.sql and < Transact root > /db/oracle/transact_orcl.sql.

Microsoft SQL Server (Windows only)

Assume you already installed jdriver for MSSQL and modify the path in setenv.cmd and startWebLogic.cmd. Then you can set up Microsoft SQL Server.

1. To create tables, replace $(username) in <transact root>/db/msql/auth_msql.sql and <transact root>/db/msql/transact_msql.sql with your user name. Execute these two files by typing:

   ```
   <MSSQL root>/bin/isql -U<username> -P<password> -S<servername>
   
     -i<transact root>/../../<transact_msql and auth_msql>.sql
   ```

   Where

   -U is username

   -P is password

   -S is servername

   -i is the sql input file

2. Add the MSSQL connection pool.

   You can add the connection pool through the WebLogic property file or other WebLogic recommended methods. For more information, refer to the WebLogic5.1 documentation at the following link:

   http://www.WebLogic.com/docs51/classdocs/conn_pools.html#635447

   An example of the Connection Pooling Property for MSSQL:

   ```
   url=jdbc:WebLogic:mssqlserver4:<table name>@<db host>:1433,\
   driver=WebLogic.jdbc.mssqlserver4.Driver,\
   props=user=<user>;password=<password>;server=<db host>
   ```

Setting Up the LDAP Server

You can set up an LDAP server to authenticate users, although this is optional.

To set up the LDAP Server (optional)

1. From your <WebLogic root> command prompt, type setenv.
2. Type startWebLogic.

   The WebLogic Server starts.
3. Open the property editor from the Web browser using the following URL:

   http://host:7001/propertieseditor/index.htm.
4. Click the Authentication tab to get to the authentication.jsp page.
5. Enter a value for the LDAP credential variable.

   The value will be encrypted and not displayed to the user.
6. Click Submit, and then shut down the WebLogic server.
7. Copy siebel.prp and siebel_default.prp to <transact root>\scripts directory.
8. From your <transact root>\scripts, type setRinconPath.
9. Run rincon_setup.
10. Run transact_setup.

    This sets up the LDAP server.
11. If you want to uninstall your setup, run transact_uninstall first, and then rincon_uninstall.

    An error appears if you try to run rincon_uninstall before you run transact_uninstall.
12. Each time you change your LDAP credential using the property editor, you must copy siebel.prp from <WebLogic root> to <transact root>/script.

    This step allows the LDAP setup script files to pick up the new variable for running the setup.

13 Open the WebLogic property file and append this line to the end of the file.

WebLogic.security.realmClass = com.siebel.isscda.wl.rincon.WLRealm.

14 Restart WebLogic server to make security take effect.

The WebSphere Environment

Complete these tasks to set up the WebSphere environment for Transact Server:

1. Create the Siebel directory.
2. Copy the Siebel property file.
3. Set up the WebSphere class path.
4. Create a Web application and Servlet.
5. Set up the LDAP password (*optional*).
6. Register Enterprise Java Beans (EJBs).
7. Create an Enterprise Application.
8. Enable Security Permission.
9. Set up the Transact database.
10. Set up the LDAP user group.
11. Set up eMail service in WebSphere.
12. Set up the Transact Server login.

The following discussion shows you how to complete each of these tasks.

Creating the Siebel Directory

With WebSphere 3.5, you must create a directory for the new application server when you create the application server.

To create a directory for the new application server

1. Create a new folder name called Siebel under <WebSphere root>/appserver/hosts/default_host/.

   The Siebel folder name is the same as your application server name.

2. Use the same case across the entire installation because the .jsp page reference is case-sensitive.

3. Copy the following folders to <WebSphere root>/appserver/hosts/default_host/Siebel:

   - Login
   - Propertieseditor
   - transactserver (under the <transact root>/jsp directory)

Administration for Siebel Property Files

Copy Siebel.prp and Siebel_default.prp from <transact-root>/properties to <Windows root>/system32. For the UNIX platform, copy Siebel.prp and Siebel_default.prp from <transact-root>/properties to <WebSphere Root>/AppServer/bin directory.

NOTE: Siebel does not recommend that you start WebSphere by running adminserver.bat on the Windows platform. However, if you choose to do that, you must first copy Siebel.prp and Siebel_default.prp to the directory where adminserver.bat resides.

Setting Up the WebSphere Class Path

You can set up the class path for WebSphere in two steps.

To set up the WebSphere class path

1. Open admin.config from <WebSphere root>/appserver/bin.
2. For com.ibm.ejs.sm.adminiserver.class path, add all Transact jar files as follows: com.ibm.ejs.sm.adminserver.classpath=...

   <Transact root>/javalib/xalan.jar;

   <Transact root>/javalib/tsIBM.jar;

   <Transact root>/javalib/xerces.jar;

   <Transact root>/javalib/xml.jar;

   <Transact root>/javalib

   Alternatively, you can add the classpath through the WAS admin console. Please refer to the IBM info center link.

The following steps are based on using the default application server. To set up a different application server, change the parameters or settings accordingly. Refer to the IBM InfoCenter for more information.

Restart the WAS Admin Server and WAS Console.

Creating a Web Application

Use the IBM WAS AdminServer to create a Web application.

To create a Web application using IBM WAS AdminServer

1. From the Tasks menu, select Create Web Application.
2. Enter Siebel in the Web Application Name field.
3. Click Serve Servlets by classname and then click Next.
4. Click Next.

5. Select your server node name and select Default Server/Default Servlet Engine, and then click Next.
6. Replace Web Application Web Path field from /webapp/Siebel to /Siebel, and then click Next.
7. Remove "\web" at the end of Document Root field.
8. Click Finish.

Creating a Servlet

Use the IBM WAS AdminServer to create a Servlet.

To create a servlet using IBM WAS AdminServer

1. In the left view panel, click Default Servlet Engine under Default Server, right-click on Siebel (the Web Server created in the step above), and then select Create and Servlet.
   - **a** Enter SiebelTransact in the Servlet Name.
   - **b** Enter com.siebel.isscda.ws.servlet.mars.RequestHandler in the Servlet name.
   - **c** Click Add for Servlet Web Path List.
   - **d** Append SiebelTransact after "Siebel/" in the Servlet Web Path.
   - **e** Click OK.
2. In the left view panel, click Default Servlet Engine under Default Server, right-click on Siebel (the Web Server created in the step above), select Create and Servlet.
   - **a** For LoginProcess, enter Servlet Name.
   - **b** Enter com.siebel.isscda.ws.servlet.rincon.LoginProcess in the Servlet Class Name.
   - **c** Click Add for Servlet Web Path List.
   - **d** Append "LoginProcess" after "/Siebel/" in the Servlet Web Path.
   - **e** Click OK.

3. In the left view panel, click Default Servlet Engine under Default Server, right-click on Siebel (the Web Server created in the step above), select Create and Servlet.

   a. Enter Logout in the Servlet Name.

   b. Enter com.siebel.isscda.ws.servlet.rincon.Logout in the Servlet Class Name.

   c. Click Add for Servlet Web Path List.

   d. Append “Logout” after “/Siebel/” in the Servlet Web Path.

   e. Click OK.

Setting Up the LDAP Password

At this point, if you are going to use an LDAP server, you need to set up an LDAP password.

To set up the LDAP password

1. Copy the propertieseditor folder from <transact root>/jsp to <Web Server>/<docs root>.
2. Open index.htm file. Change from “src=/common.jsp” to “src =

   /Siebel/propertieseditor/common.jsp.”
3. Make sure the HTTP server is up and running.
4. Start the default application server from the console.
5. Open a browser and enter http://<hostname>/propertieseditor/index.htm to modify the LDAP Credential from the Authentication page.
6. Enter your LDAP password and click Submit.

Register EJBS

Before users can continue to create an application server for Transact, they must check whether or not the JDBC driver and data source are installed and set up correctly. For information on installing and setting up JDBC and data source, refer to Appendix I, "Additional Tasks."

To create EJBs

1. Link DataSource to the application server.
   - In the left view panel, click Default Server.
   - Select Default Container.
   - In the right view panel, select Data Source.
   - Click Change.
   - Select the correct DataSource.
   - Click Apply.
2. Create EJBs.
   - In the left view panel, select Default Server, right-click on Default Container, and then select Create and EnterpriseBean.
3. Enter the Java bean name (*Siebel prefix is recommended*).
4. Click Browse and select tsIBM.jar from <transact root>/javalib.
5. Click Deploy and Enable WLM.

   A significant amount of time is required to create all the Java beans.
6. After the bean has been deployed, you can set up the database to enable table creation for the CMP bean from the console; or, you can perform this setup at a later time. Refer to the IBM information center document for more information.

7. Two types of Java beans are created: CMP beans and Session beans.

   CMP beans are OLKeySequencer, OLSession, OLAppData, OLItem, OLQuote, OLPackage, and OLGenericElem. Usually, “create table” is checked from the default setup for all the CMP beans. However, Siebel Systems recommends that you check if “Create Table” is marked for all the CMP beans. To see the database setup, click EJB and in the right panel click “DataSource.”

8. Select OLKeySequencerHome and click the General tab from the right panel.
9. Change the Database Access from “Shared” to “Exclusive,” and then apply this change.

Creating an Enterprise Application

Use the IBM WAS AdminServer to create an Enterprise application.

To create an enterprise application using IBM WAS AdminServer

1. Select Create Enterprise Application from the Tasks menu.
2. Enter ISSCDA in the Enterprise Application.
3. Select the EJBs created in the previous section.
4. Highlight the beans, and then click Add.
5. Select Siebel, which is the Web Server created in the previous procedure.
6. Click Finish.

Enabling Security Permission

Use the IBM WAS AdminServer to enable security permission.

To enable security permission using IBM WAS AdminServer

1. Enable Global Security.
   a. Use the IBM WS AdminServer to Enable Global Security.
   b. Select Configure Global Security from the Tasks menu.
   c. Select Enable Security, and then click Next.

- **d** Select Basic (User ID and Password), and then click Next.
- **e** Select Lightweight Third Party Authentication, and then click Next.
- **f** Under User Register, provide the required information, and then click Finish.

  Examples of input values:

  ```
  Security ServerID: <valid user in LDAP>/Security Server

  Password: <user password>/Directory

  Type: Netscape/Host: <ldap host name>/Port:389/Base
  Distinguished

  Name: ou=People, o=<domain name>.
  ```
- **g** Click Finish.
- **h** Enter an LTPA password.
- **i** Click Finish.

2 Enable Resource Security.

Select Configure Resource Security from the Tasks menu.

- **a** Enterprise Beans
  - ❑ Click Enterprise Beans.
  - ❑ Select a Transact EJB, and then click Next.
  - ❑ Use the Default method.
  - ❑ Highlight all the methods from the EJB, and then click Finish.
  - ❑ Repeat to configure all the EJBs.
- **b** Virtual Hosts
  - ❑ Click Virtual Hosts.
  - ❑ Select default_host, and then click Next.
  - ❑ Select a Transact servlet.

- Use the Default method.
- Select all methods, and then click Finish.
- Repeat to configure for all the Transact servlets.

3 Enable Security Permission.

Select Configure Security Permission from the Tasks menu.

a Select ISSCDA, the Enterprise Application created in the previous procedure, and then click Next.

b Highlight all the methods, and then click Next.

c Select EveryOne to enable the security for everyone.

d Click Finish.

In order for these security changes to take effect, you must restart the IBM WAS AdminServer.

Set Up the Transact Database

The Transact database tables can reside in a DB2 or Oracle database instance.

DB2 Database Instance

You must alter the DB2 tables to set up the Transact database.

1. Once the application server starts, you must run the two scripts provided that alter the Transact tables in the DB2 database.
2. Run tsDB2.sql to update the Transact tables.

Oracle Database Instance

You must alter the Oracle tables to set up the Transact database.

1. Once the application server starts, you must run the two scripts provided that alter the Transact tables in the Oracle database.
2. Run tsORACLE. sql to update the Transact tables.

Setting Up the LDAP User Group

If you choose to use an LDAP server, then you must set up the LDAP user group.

To set up the LDAP user group (optional)

1. Run setRinconPath.bat from <Transact root>/scripts for Windows platform.
2. Copy Siebel.prp and Siebel_default.prp from <Windows root>/system32 to <Transact root>/Scripts for Windows platform.
3. Copy Siebel.prp and Siebel_default.prp from <WebSphere Root>/AppServer/bin to <Transact root>/Scripts for UNIX platform.
4. Default Rincon script only provides system/guest as default users. You must use Rincon_add_user.sci script to add more users.
5. Run Rincon_setup.bat for the first time setup for the Windows platform, or run Rincon_setup.sh for UNIX platform.
6. Before you can add LDAP users, you have to modify rincon_add_user.sci. (If you do not want to add new users, you can skip this step.)
   - Run rincon_run.bat to add new users for Windows platform, or run Rincon_run.sh for UNIX platform.
7. Run transact_setup.bat to set address and group information for users from the Windows platform, or run transact_setup.sh for UNIX platform.

Setting Up eMail Service

To set up an eMail service, you must install a package that supports Java mail.

To set up eMail Service

1. Install any package that supports Java Mail 1.1 API.
2. Set up the class path to reference the Java Mail Package.
3. Open the property editor to change the email_enable_flag to “on.”
4. Restart the WebSphere server.

Setting Up the Transact Server Login

You can set up the Transact Server login in two steps.

To set up the Transact Server login

1. Open the LoginPage.jsp from

   < WebSphere root > /appserver/hosts/default_host/Siebel/login. Change "action = /LoginProcess" to "action = /Siebel/LoginProcess."

2. Modify "ShoppingCart.jsp" to update the line " < script src = "http:// < web host name > / < your application folder > /jd/header.js" > < /script > " with the correct URL.

Prepare the Application to Connect with Transact Server

To prepare the Siebel application so that it can connect with the Transact server, you must complete the following tasks:

1. Initialize Transact configuration values and the Order_Subvar variable.
2. Initialize properties in the Transact property file.
3. Redirect pagesets to the correct Order Management System.
4. Configure the HTTP server.

These tasks complete the installation and implementation process.

Setting the app_config.js Properties

After you install Transact, you must set the several properties in the app_config.js file. This file is located in the Custom directory where you installed your Siebel browser-based application:

- Transact Active
- Transact URL

Table 1 describes all the Transact properties which you can modify in the app_config.js file.

Table 1. Transact Properties to Set in app_config.js

Property	What It Does	Example
Transact Active	Turns on/off Transact	var TRANSACT_ACTIVE = true;
Transact Not Active Msg	Sets the message that will appear if a user tries to access Transact functionality when TRANSACT_ACTIVE is set to false.	var TRANSACT_NOT_ACTIVE_MSSG = “Sorry, the action you have requested is currently unavailable. \n Please contact your system administrator.\n”;

Table 1. Transact Properties to Set in app_config.js

Property	What It Does	Example
Transact URL	Sets the URL of Siebel Transact.	var TRANSACT_URL = "http://application server/siebel/SiebelTransact";
Transact Third Party Cart	Determines whether or not a third party cart will be used.	var TRANSACT_THIRD_PARTY_CART = true;
Transact Show Cart URL	Sets the URL that will be used when the third party cart is opened.	var TRANSACT_SHOW_CART_URL = "http://www.company.com/shoppingcart.jsp";
Transact Cart Winargs	Sets the window properties for the shopping cart. This is also used to open a window for the third party cart.	var TRANSACT_CART_WINARGS = "height=500,width=500,scrollbars=1, resizable=1,menubar=0";
Transact Cart Target	Determines whether the Transact cart will appear in a window ("_new") or in a frame ("frameName").	var TRANSACT_CART_TARGET = "_new";

Setting the Order_Subvar Configuration Variable

You must initialize the ORDER_SUBVAR variable that defines subitems in app_config.js file.

Siebel browser-based applications use the ORDER_SUBVAR variable to identify "items" in your feature table. Feature tables can store various kinds of data that applies to a product.

For example, a car may be represented by the features SKU, MODEL, COLOR, and CAR_ALARM. Now, if you want to use subitems on a bill of materials, you must first identify the column that represents the product as an "item." Before you make this distinction, every feature (SKU, MODEL, COLOR, and CAR_ALARM) is the same to Siebel eAdvisor. Once you define the ORDER_SUBVAR variable, you can identify which features are descriptive and should not be considered line items (such as COLOR), and which are actual items and should appear on a bill of materials (such as CAR_ALARM).

ORDER_SUBVAR should be set to the column name of a feature table which contains a part number or other information which identifies that item distinctly from other items in the feature table. If you set the ORDER_SUBVAR variable to “SKU,” when a feature table contains a column named SKU, each row that has an entry in the SKU column becomes an item. In this scenario, a bill of materials would include parts for every row that includes SKU.

Note that subitems are needed only when a main level product is made up of other products.

Example:

```
var ORDER_SUBVAR = "SKU";
```

See *Siebel Interactive Designer Administration Guide* for more information on feature tables.

Setting the Transact Properties

After you install Transact, use the property editor to set the following:

- Transact URL
- eAdvisor URL

Use the property editor to set any additional Transact Server properties you want to change.

To open the property editor

1. Start WebLogic/WebSphere.
2. Open a browser and enter the URL:

   http:// < localhost:7001 > /PropertiesEditor

You can modify the URL as needed by your installation.

Table 2 provides a description of all Transact properties you can set in the property editor.

Table 2. Transact Properties to Set in the Property Editor

Property	What It Does	Example
Transact Log Debug	Sets the Transact log to debug on or off. When off, errors are logged; when on, all activities are logged.	on
Transact Log Filename	The filename of the Transact log file. This property requires an absolute path.	e:\\WebLogic\\ol_transact.log
Transact URL	The URL of Transact: the URL of the WebLogic server + “/SiebelTransact”	http://www.company.com/SiebelTransact
eAdvisor URL	The URL of the Siebel eAdvisor application.	http://www.company.com/eAdvisor
Use Third Party Cart	Determines whether or not Transact is using a third-party shopping cart.	true
OMS URL	Sets the URL of the Order Management System that will accept XML from Transact. Transact posts the XML when the AddToCart function is called if the Transact Third Party Cart variable in app_config.js is set to “true.” See “Redirecting a Page Set to a New OMS” on page 57 for information on using multiple Order Management Systems.	http://www.companyName.com/servlet/showXML
Transact Cart XSLT	The URL or absolute filepath of the XSLT stylesheet Transact will use to convert data to the desired XML format.	http://www.company.com/olcq2cXML.xsl
Form Post JSP Page	If you are using a third-party cart and want to use a form post, enter the name of the JSP page you are using to dynamically create the form.	TransactServer/form_post.jsp
SMTP Server	Sets the name of your email server from which configuration emails will be sent.	Post

Table 2. Transact Properties to Set in the Property Editor

Property	What It Does	Example
Configuration Time-Out Period	Sets the number of days after which configurations will be removed from the Configuration list.	14
Use Multiple OMS	Determines whether or not you are using multiple Order Management Systems. When you set this property to true, you will need to implement the InitAltOMSUrl function. See "Redirecting a Page Set to a New OMS" on page 57 for additional information.	false
Multi-OMS Redirect	If you are using multiple Order Management Systems, use this property to specify the URL to be displayed after the user clicks Add to Cart.	http://www.company.com/ index.htm
Email enable flag	You can turn the email flag off if you do not wish to use the email function.	on
Display shopping cart flag	You can turn the display shopping cart flag off to avoid shopping cart being displayed after you have executed AddToCart function.	true
Shopping cart opener	If you customized your shopping cart and it needs to refer to previous frame context, modify this parameter to refer to your previous frame context.	Top.opener
Login required action	You can specify login required actions from mar_guest_ok parameter.	AddToCart, SubmitCart, ShoppingCart.jsp, OpenPackage, ViewQuote.jsp, EmailQuote.jsp, EmailConfig.jsp, SavePackage, SaveQuote, ConfigList.jsp, QuoteList.jsp, ShoppingCart_orig.jsp, ShoppingCart_Header.jsp, ShoppingCart_Bottom.jsp, ShoppingCart_Middle.jsp
List quote database table name	The database table name for list quote action.	Ejb.olquotebeantbl

Table 2. Transact Properties to Set in the Property Editor

Property	What It Does	Example
List package database table name	The database table name for list package action.	Ejb.olpackagebeantbl
List quote/package database user id	The database user ID for list quote/package action.	Dbuser
List quote/package database user password	The database user password for list quote/ package action.	Dbpassword
List package/quote database source name	The data source name for list package/quote action.	DbSource. For WebLogic, you will need to add a line in WebLogic.property file to indicate the data source name (WebLogic.jdbc.DataSource.SiebelDataSource = onlinkPool). For WebSphere, the data source name will be the data source name you used when you set up for Transact Server EJBs. You need to append jdbc/ in front of the data source name—for example, jdbc/SiebelDB.

NOTE: The property editor does not check for invalid values, so be precise when you enter a property value.

Redirecting a Page Set to a New OMS

When a user clicks Add to Cart, the selected products are sent to the Order Management System. When you want to use different Order Management Systems for different pagesets, you must implement the InitAltOMSUrl function to specify which OMS URL each page set will access. You must complete this action in addition to setting the Use Multiple OMS property in the property editor to true.

Call the InitAltOMSUrl function from the <pageset>_x.js file in Siebel eAdvisor. This function takes one parameter (a URL) and it is at the top level of the application.

Example:

```
OL.InitAltOMSUrl("http://other.oms.com/dest/");
```

All products ordered from the pageset that use this function will be sent to the URL you supplied as a parameter.

See *Siebel Interactive Designer Administration Guide* for more information.

Configuring the HTTP Server

To set up a link between one of the Transact Server's supported Web servers and WebLogic/WebSphere, you must configure WebLogic/WebSphere and your Web server to redirect requests to WebLogic/WebSphere.

To set up a link between Transact Server and a Web server

1. Read the WebLogic/WebSphere manual to understand what plug-ins or sharable libraries are needed for the Web server you are using.
2. Read the Web server manual to understand how to redirect HTTP traffic.
3. Install WebLogic/WebSphere plug-ins or sharable libraries on the Web server.
4. Set up redirection inside the Web server to the WebLogic/WebSphere URL. Specifically, understand the need to redirect .jsp pages and specific servlet requests.
5. For IIS Web Servers, you may need to move the transactserver folders to your Web server root directory.

Troubleshoot Transact Server

After installing Transact, you may encounter problems. To identify, characterize, and solve these problems, you can turn to the log file. The discussion below summarizes how to obtain comprehensive information from the Transact log file and also how to manage concurrency.

Identify the Problem

If you run into problems after the installation, use the log file to identify the source of the problem. Once identified, if you are unable to correct the problem, then you can contact Siebel Technical Support.

To identify a problem

1. Set the Transact Log Debug property to on in the property editor.

   Transact then records in the log details about all activities.

2. Perform the same actions you did when the problem occurred.

3. Open the log file.

   The location of the log files is specified in the property editor.

The error message contains a time stamp, an error number, a message that states the attempted action and consequences, the actual Java exception message, and a stack trace of where it occurred in the code. In the following example, the Configuration List could not be generated because the socket connection was closed (the user navigated elsewhere before it had a chance to return). This error is given by the Java exception.

Example Error:

```
8/16/00 2:55 PM[OLM_MAR_E_05000] ERROR: Couldn't complete action!

(action=ConfigList.jsp)

java.net.SocketException: Connection reset by peer: socket write
error
```

From looking at the log file, you may be able to determine the problem yourself. If not, record the log activity that occurred while you were reproducing the problem and contact technical support. Providing your technical support representative with this information is the easiest way to resolve the problem.

Concurrency Control

There may be cases where more than one user attempts to update a configuration. To manage concurrent access, configurations have a date last modified (date_modified) timestamp field which is updated whenever the object is written to. When a configuration is accessed for reading or writing, the date_modified field currently associated with the object is queried. When you attempt to update the object, the date_modified held locally is compared with the date_ modified of the stored object. If the date_modified of the object is more recent than the local copy, you will not be able to modify the object and you will receive a message asking you to open it and save it again, or save a new copy. By saving the configuration as a new configuration, you can still save your work.

Only the user who initially created the configuration has permission to delete that configuration. If you delete the configuration while someone else has it open for reading, the other user will not be allowed to save changes and will be notified that the user who created this configuration has deleted it and it is no longer valid. The user can then save a new copy to save his or her work.

Working with Configurations 3

This chapter describes configurations, the configuration list, quotes, and how to manipulate configuration lists.

When you click the Save Configuration button in your Siebel application, all of the selections for the current page set are saved as a configuration in the configuration list. You can then open the configuration from the configuration list and continue to work with your saved settings.

A quote is a collection of configurations and includes quantities and pricing information. A quote also includes account information. An order is a quote that has been specifically submitted as an order to your Order Management System. See the documentation for your shopping cart and Order Management System for information about working with quotes and orders.

Working with the Configuration List

In the Siebel application, the application developer provides a link to the configuration list. Users can view all of the configurations they have saved by opening the configuration list. With the proper permissions, a user can also view and edit configurations saved by other users. For additional information, see "Permissions" on page 109.

Maintaining the Configuration List

To make sure that your configuration list displays configurations that are in use and do not grow too long, use the property editor to set a time-out period after which unused configurations are deleted from the Configuration list. After configurations are removed from the Configuration list, they remain in the database and can be restored by making the time-out period longer or by changing the date_created field of a particular configuration in the database. Your database administrator will occasionally need to clean old configurations out of the database.

To set the time-out period for saved configurations, set the Configuration Time-Out property in the property editor to the desired number of days.

Modifying the Configuration List UI

To modify the configuration list UI, edit the ConfigList JSP (Java Server Page) to call the appropriate ConfigList functions and set the sort order. The ConfigList functions determine what information will be displayed in the configuration list. Appendix C, "ConfigList API" provides a list of all the functions you can use on the JSP page.

By default, the configuration list displays a configuration ID number, the name of the configuration, the date the configuration was created, a Delete button, and a link to the restored configuration in the Siebel application.

The default JSP page calls these functions:

- getUniqueId()
- getName()
- getDateCreated()
- getDeleteURL()
- getRestoreURL()

During the call, the sort order is by name in descending order:

```
<% ConfigList.setSortField("Name");
ConfigList.setSortOrder("Desc"); %>
```

These settings produce the following list of configurations, shown in Table 3.

Table 3. Configuration Lists

Open Config	Name	Date Created	Delete	Email
Open	Golcen	2000-08-31 15:54:47.000	Delete	Email Config
Open	Chihuahua	2000-08-31 15:54:20.000	Delete	Email Config
Open	Shelty	2000-08-31 15:53:48.000	Delete	Email Config

About the JSP Page

The configuration list is created in a JSP page called ConfigList.jsp which is located under < WebLogic home > /myServer/public_html/TransactServer.

JSP is a technology for controlling the content and appearance of Web pages through the use of servlets. Servlets are small programs that are specified in the Web page and run on the Web server. These programs modify the Web page before it is returned to the requestor.

To modify the format of the Configuration List, use the ConfigList API functions. From the JSP page, you must call the ConfigListBean. The ConfigListBean defines the functions you can call from your JSP page.

The UseBean Tag

Implement the UseBean tag to import the ConfigListBean into your JSP page. Include the following code near the top of your JSP page.

```
<!-- BEA WebLogic -->
<jsp:useBean id=”ConfigList”
scope=”page”
class=”com.siebel.isscda.wl.transact.ConfigListBean”>
<!-- BEA WebLogic -->
<!-- IBM WebSphere -->
<jsp:useBean id=”ConfigList”
scope=”page”
class=”com.siebel.isscda.ws.transact.ConfigListBean”>   ">
<!-- IBM WebSphere -->

<% ConfigList.setSortField(“DateCreated”);
ConfigList.setSortOrder(“Desc”);
ConfigList.createList(session, request); %>
</jsp:useBean>
```

This code imports the ConfigListBean, sets up the sort order, and then creates the configuration list.

The While Loop

The JSP page must also include a while loop. The while loop traverses the list of saved configurations. Each iteration of the loop is a single saved configuration (or one row in the table). Get methods are used to return information about the saved configuration and display this information in the Configuration list.

The following HTML defines the table structure for display of the Configuration list.

```
<table border=1>
<tr><th>Config ID</th><th>Name</th><th>Date Created</
th><th>Delete</th><th>See XML</th></tr>
<% while (ConfigList.nextConfig()) { %>
<tr>

<td><a href=”<%= ConfigList.getRestoreURL() %>” target=”_new”><%=
ConfigList.getUniqueId() %></a></ td>
<td><%= ConfigList.getName() %></td>
<td><%=ConfigList.getDateCreated() %></td>
<td><a href=”<%= ConfigList.getDeleteURL() %>”>Delete</ a></
td><td><a href=”<%= ConfigList.getExportURL() %>”
target=”_new”>xml</a></td>
</tr>

<% } %>
```

Between the fourth line of code:

```
<% while (ConfigList.nextConfig()) { %>
```

and the last line of code:

```
<% } %>
```

you can add any of the get() functions from the ConfigList API.

For a complete description of the available get() functions, see Appendix C, “ConfigList API.”

Default ConfigList JSP Page

Below is the complete code for the default ConfigList JSP page.

```
<!doctype html public “-//w3c/dtd HTML 4.0//en”>
<html>
<head><title>Config List</title></head>

<%@ page
info=”JSP test”
contentType=”text/html”
%>

<!-- BEA WebLogic -->
<jsp:useBean id=”ConfigList”
scope=”page”
class=”com.siebel.isscda.wl.transact.ConfigListBean”>
<!-- BEA WebLogic -->

<!-- IBM WebSphere -->
<jsp:useBean id=”ConfigList”
scope=”page”
class=”com.siebel.isscda.ws.transact.ConfigListBean”>
<!-- BEA WebLogic -->

<% ConfigList.setSortField(“DateCreated”);
ConfigList.setSortOrder(“Desc”);
ConfigList.createList(session, request); %>

</jsp:useBean>
Saved configurations for <%= ConfigList.getUserId() %><br>

<table border=1>
<tr><th>Config ID</th><th>Name</th><th>Date Created</
th><th>Delete</th><th>See XML</th></tr>
<% while (ConfigList.nextConfig()) { %>
<tr>

<td><a href=”<%= ConfigList.getRestoreURL() %>” target=”_new”><%=
ConfigList.getUniqueId() %></a></ td>
<td><%= ConfigList.getName() %></td>
<td><%=ConfigList.getDateCreated() %></td>
<td><a href=”<%= ConfigList.getDeleteURL() %>”>Delete</ a></
td><td><a href=”<%= ConfigList.getExportURL() %>”
target=”_new”>xml</a></td>
</tr>
<% } %>
</table>
</body>
</html>
```

Emailing a Configuration

If you want to share a saved configuration with another user, you can select and email the configuration for the configuration list.

To add the Email Configuration functionality

1. Add a button or link to the Configuration List page.
2. Call the getEmailURL function.

   See "getEmailURL" on page 178 for information on implementing this function.
3. Implement a function to target a new window.
4. From the Email JSP page, call Email Bean functions to implement email functionality.

   For more information about the Email Bean functions, see Appendix D, "Email Bean API."
5. Specify the name of your SMTP server in the property editor.
6. Restart the WebLogic server.

Sample Email Bean JSP Page

Use the following JSP page as a template. You can modify this template to implement the functionality that allows a user to email a configuration from the Configuration List page.

Each section of code includes an explanation. Remove the explanations to view the complete code for the JSP page. You can also open the Email JSP page (EmailConfig.jsp) from the TransactServer directory located under WebLogic/myServer/public_html.

Email JSP Page Code

```
<%@ page
        info="JSP test"
        contentType="text/html"
%>
```

This page statement states that you are in a JSP page.

```
<html>
<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<!-- BEA WebLogic -->
<jsp:useBean id=”Email”
scope=”page”
class=”com.siebel.isscda.wl.transact.ConfigListBean”>
<!-- BEA WebLogic -->

   <!-- IBM WebSphere -->
   <jsp:useBean id=”Email”
   scope=”page”
   class=”com.siebel.isscda.ws.transact.ConfigListBean”>
   <!-- BEA WebLogic -->
```

This statement imports the bean that contains the Email Bean functions that are used on this page.

```
<%
   Email.setSession(session);
   Email.setRequest(request);
%>
```

These calls are placed inside the useBean tag to make sure that they are executed first. They are initialization statements designed to provide the bean with the session and request variables available to the JSP page. “Session” and “request” are passed in, and JSP is equipped to recognize them.

```
</jsp:useBean>
```

This is the end tag for useBean.

```
<%
   if (request.getParameter("To") != null) {
```

This if statement searches for a parameter named “To” in the request.

There is a text field named “To” on this form. The form submits to the page that contains the form. If the “To” parameter contains information, it means the user has entered their email information and submitted the form. At this point, the email is ready to be sent.

```
if (!Email.sendMail(request.getParameter("To"),
request.getParameter("Subject"),
request.getParameter("Message"))) {
%>
```

This call to sendMail sends the email. Concise syntax is used by calling sendMail from within an if statement. The parameters for sendMail are obtained from the request, using the field names, which are identical to the field names in the form below (because the user filled out the form and submitted it).

```
<script language=JavaScript>
alert("<%=Email.getErrorMessage() %>");
</script>
```

If sendMail returned false, this section of code is called, displaying a JavaScript alert whose message text is supplied by a call to getErrorMessage. getErrorMessage returns an error message that attempts to explain to the user why the mail did not get sent.

```
<%
   }
   else { %>
```

In this section, JSP tags close the if statement and start the else statement. If we get to the else statement, we know the email was sent successfully and the next section of code is called.

```
<script language=JavaScript>
self.close();
</script>
```

Since the mail was sent successfully, this section of code closes the email window.

```
<%      }
   }
%>
```

This section closes the else and the if (request.getParameter(“To”) != null) statements.

```
</head>
<body bgcolor="#FFFFFF">
<form name="Email" action="<%=Email.getAction("EmailConfig.jsp")
%>" method="post">
```

This section fills in the form action using the JSP call to getAction. The name of the template is passed in to the getAction method.

```
<div align="center"><center>
<table border="0">
    <tr>
        <td valign="top">To:</td>
        <td valign="top"><input type="text" size="80" name="To"
value="<%= (request.getParameter(“To”)!=null ?
request.getParameter("To") : "") %>" ></td>
```

This statement says that if the user has entered a value for the “To” field, prefill the text box named “To” with that value. Otherwise, leave the text box blank.

```
    </tr>
    <tr>
        <td valign="top">Subject:</td>
        <td valign="top"><input type="text" size="80"
name="Subject" value="<%= (request.getParameter(“Subject”) !=
null ? request.getParameter("Subject") : "") %>"></td>
```

This statement says that if the user has entered a value for the “Subject” field, prefill the text box named “Subject” with that value. Otherwise, leave the text box blank.

```
    </tr>
    <tr>
        <td valign="top">Message: </td>
        <td valign="top">
            <p><textarea name="Message" rows="4" cols="80">
             <%= (!Email.getMailSent()?
request.getParameter("Message") : Email.getRestoreConfigURL())
%>
```

In this section, if the mail did not get sent, the “Message” field is prefilled with the text the user entered when he or she submitted the message. Otherwise, a URL is entered to reopen the configuration. This is the URL that the email recipient will click to launch your Siebel application and restore the saved configuration in the browser, so it is very important to include it in the message.

```
            </textarea></p>
        </td>
    </tr>
    <tr>
        <td align="center" colspan="2"><input type="submit"
        name="OK" value="OK"> <input type="button" name="Cancel"
        value="Cancel" onClick="self.close()"></td>
```

This section creates the HTML buttons for submitting or canceling out of the email dialog box.

```
    </tr>
</table>
</center></div>
</form>
</body>
</html>
```

Modifying the Save Configuration UI

When you click Save Configuration, a dialog box appears that solicits a name for the configuration. If you linked back to the configuration and then clicked Save, a dialog box would appear soliciting a unique response:

- Overwrite the configuration
- Save a new configuration
- Cancel

You may want to write your own JavaScript dialog box or implement Save Configuration as a button on your Results page. You can override the default functionality of the Save Configuration dialog box by using the OR_GetSaveConfigName function. For more information, see "OR_GetSaveConfigName" on page 168.

Integrating the Order Management System 4

Transact connects to your Order Management System when you add a configuration to your shopping cart.

To enable this functionality, you must perform the following steps to integrate the Siebel application with the Order Management System:

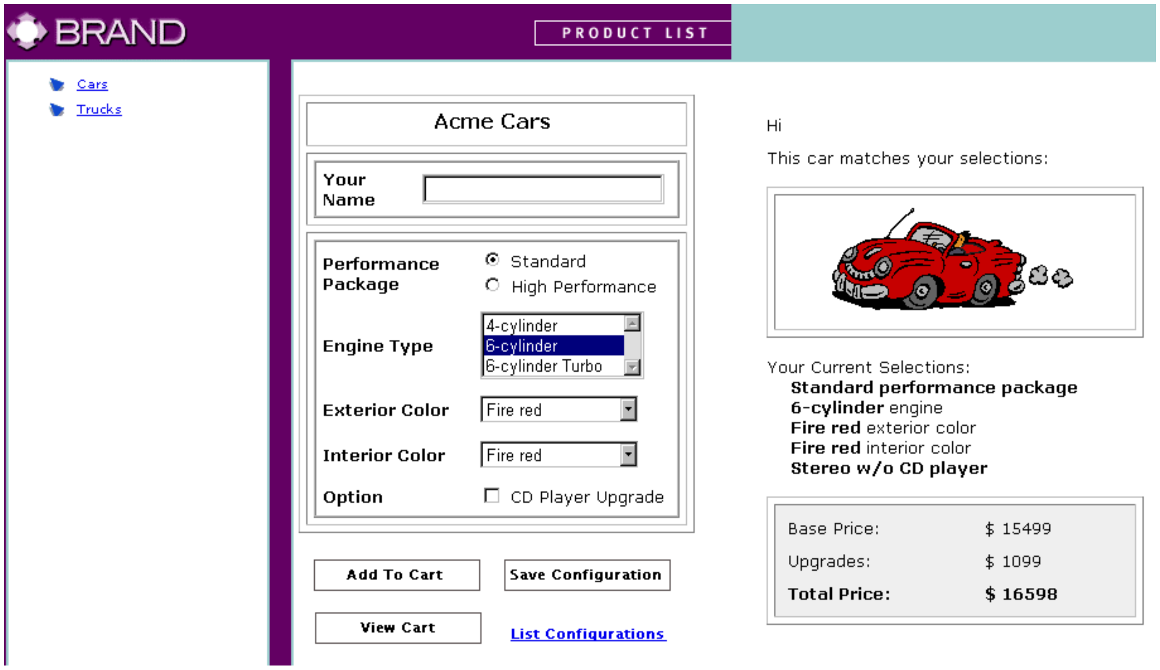
- Connect buttons in the Siebel application to those of the shopping cart using the eAdvisor Javascript API.
- Write a style sheet to the DTD (Document Type Definition) if you want to customize the format of the configuration information passed between the Siebel application and the Order Management System.
- Optionally, you can add a URL in the Order Management System that links back to the Siebel application configurations.

This chapter describes how to complete these tasks.

Connecting Your Siebel Application UI to Transact

To connect the Siebel application UI to Transact, you must add buttons or links to the Siebel application and then use OnClick events to call the Transact functions.

After you add the buttons and links to the Siebel application UI, the Transact UI occurs within the Siebel UI as illustrated by Figure 1. The Transact buttons and link appear below the Acme Cars panel.



Your Name

Performance Package: (● Standard) ○ High Performance

Engine Type: 4-cylinder, [✓ 6-cylinder], 6-cylinder Turbo

Exterior Color: Fire red

Interior Color: Fire red

Option: ☐ CD Player Upgrade

Hi

This car matches your selections:

Your Current Selections:
Standard performance package
6-cylinder engine
Fire red exterior color
Fire red interior color
Stereo w/o CD player

Base Price:	$ 15499
Upgrades:	$ 1099
Total Price:	**$ 16598**

Figure 1. The Transact UI in Your Siebel Application

For information on adding buttons and links in your Siebel application, see *Siebel Interactive Designer Administration Guide*.

You can use whatever kinds of links and names you choose. For example, you could use Show Cart instead of View Cart, and use a link rather than a button in your UI. For the purposes of discussing the UI in this guide, the working assumption is that the following buttons and link have been added to the application UI:

- Add To Cart button
- View Cart button
- Save Configuration button
- Configuration List link

Add To Cart Button

The Add to Cart button saves the current configuration to the shopping cart. Afterwards, the shopping cart opens and displays the configuration in the shopping cart list.

For information on posting a form when Add to Cart is clicked, see "Posting a Form from Add to Cart" on page 80. When you add a configuration to your cart, it is saved in Transact so it can be referenced by the cart, but is not added to the configuration list.

For information on connecting to your shopping cart, see "Connecting Your Siebel Application UI to Transact" on page 76.

To connect the Add to Cart button to Transact Server

1. Set the Transact Third Part Cart property to true in app_config.js.
2. Provide the URL for the cart for the Transact Show Cart URL property in app_config.js.
3. Set the Use third-party Cart property to true in the property editor.
4. Provide the URL of the OMS for the OMS URL property in the property editor.

5 Add a button named Add To Cart to the eAdvisor application.

6 Call the AddToCart function from the button.

Example:

```
<INPUT type="button" value="Add To Cart" name=add
onClick="OL.AddToCart('_new');"><br>
```

For more information, see "AddToCart" on page 156.

View Cart Button

The View Cart button opens the shopping cart.

For information on connecting to your shopping cart, see "Connecting Your Siebel Application UI to Transact" on page 76.

To connect the View Cart button to Transact

1. Add a button named View Cart to the eAdvisor application.
2. Call the ShowCart function from the button.

Example:

```
<a href="" onClick="OL.ShowCart('_new'); return false;">Show
Cart</a><br>
```

For more information, see "ShowCart" on page 158.

Save Configuration Button

The Save Configuration button saves the current configuration to the configuration list. This list is stored in the Transact database.

To connect the Save Configuration button to Transact

1. Add a button named Save Configuration to the eAdvisor application.
2. Call the SaveConfig function from the button.

Example:

```
<INPUT type="button" value="Save Config"  name=save
onClick="OL.SaveConfig();"><br>
```

For more information, see "SaveConfig" on page 158.

Configuration List Link

The Configuration List link opens the configuration list which displays your saved configurations.

To connect the Configuration List link to Transact

1. Add a link named List Configurations to the eAdvisor application.
2. Call the ConfigList function from the link.

Example:

```
<a href="" onClick="OL.ConfigList('ol_ui.mainArea');return
false;" >List Configs</a><br>
```

For more information, see "ConfigList" on page 157.

Posting a Form from Add to Cart

Instead of connecting to your Order Management System shopping cart, you can display a shopping cart by posting a form. This section provides an example JSP page together with the resulting HTML that posts a form.

For more information, see Appendix C, “ConfigList API.”

Sample Form Post JSP Page

The following example JSP page posts a form.

```
<html><head></head><body onLoad=document.oms_form.submit()>
<!-- BEA WebLogic Application Server-->
<%@ page
contentType=”text/html” import="com.siebel.isscda.wl.transact.*"
%>  ">

<jsp:useBean id="configBean"
class="com.siebel.isscda.wl.transact.ConfigAccessBean"> ">
<!-- BEA WebLogic Application Server-->

<!—IBM WebSphere Application Server-->
<%@ page
contentType=”text/html” import="com.siebel.isscda.ws.transact.*"
%>  ">
<jsp:useBean id="configBean"
class="com.siebel.isscda.ws.transact.ConfigAccessBean"> ">

<!—IBM WebSphere Application Server-->
<%
  configBean.setRequest(request);
  configBean.setSession(session);
  try {
     configBean.getConfig();
  }
  catch(Exception e) {
%>
   <script language="JavaScript">
   alert("An error occurred!  Your configuration was not added to
the cart!!");
<%
  }
%>
```

```
</jsp:useBean>
<form name=oms_form action="http://www.myCompany.com"
method=POST target="new">
<input type="hidden" name="shipto_name" value="<%=
configBean.getShippingName() %>">
<input type="hidden" name="shipto_street" value="<%=
configBean.getShippingStreet() %>">
<input type="hidden" name="shipto_city" value="<%=
configBean.getShippingCity() %>">
<input type="hidden" name="shipto_state" value="<%=
configBean.getShippingState()  %>">
<input type="hidden" name="shipto_zip" value="<%=
configBean.getShippingZip() %>">
<input type="hidden" name="shipto_country" value="<%=
configBean.getShippingCountry() %>">
<%

    while ( configBean.hasMoreParts() )
        {
            configBean.nextPart();
%>

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_description" value="<%= configBean.getPartDescr() %>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_siebel_id" value="<%= configBean.getPartLineItemID() %>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_part_num" value="<%= configBean.getPartNum()%>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_qty" value="<%= configBean.getPartQty()%>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_myfield" value="<%= configBean.getPartField("myfield")%>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_mydesc" value="<%=  configBean.getPartField("DESC") %>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_price" value="<%= configBean.getPartPrice()%>">

<input type="hidden" name="item_<%= configBean.getPartIndex()
%>_extprice" value="<%= configBean.getPartExtPrice() %>">
<%
        }
   configBean.finishUp();
%>
</form>
</body>
</html>
```

Form Post Result

The following code is the HTML that results from the sample Form Post JSP page.

```
<html><head></head><body onLoad=document.oms_form.submit()>

<form name=oms_form action="http://www.myCompany.com"
method=POST target="new">

<input type="hidden" name="shipto_name" value="">

<input type="hidden" name="shipto_street" value="">

<input type="hidden" name="shipto_city" value="">

<input type="hidden" name="shipto_state" value="">

<input type="hidden" name="shipto_zip" value="">

<input type="hidden" name="shipto_country" value="">

<input type="hidden" name="item_0_description" value="cats page">

<input type="hidden" name="item_0_siebel_id" value="74">

<input type="hidden" name="item_0_part_num" value="">

<input type="hidden" name="item_0_qty" value="1">

<input type="hidden" name="item_0_myfield" value="">

<input type="hidden" name="item_0_mydesc" value="">

<input type="hidden" name="item_0_price" value="60.00">

<input type="hidden" name="item_0_extprice" value="60.00">

<input type="hidden" name="item_1_description" value="Tabby">

<input type="hidden" name="item_1_siebel_id" value="74">
```

```
<input type="hidden" name="item_1_part_num" value="TA-MH-GR-GR-
1">

<input type="hidden" name="item_1_qty" value="1">

<input type="hidden" name="item_1_myfield" value="">

<input type="hidden" name="item_1_mydesc" value="Tabby">

<input type="hidden" name="item_1_price" value="60.00">

<input type="hidden" name="item_1_extprice" value="60.00">

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

Working With the DTD

Transact translates a configuration into an XML string which it can pass to the Order Management System. The DTD (Document Type Definition) defines the format of the XML string. If your Order Management System cannot read and interpret the XML string, then you must write a style sheet to the DTD so that the XML can be processed by the Order Management System. See "Using Style Sheets" on page 91 for more information.

The Line Item DTD organizes the data into a hierarchy of items, each identified as parent or child. For example, if you select a computer, modem card, mouse, and keyboard in the eAdvisor application, the modem card, mouse, and keyboard are children of the parent computer. Items can contain an unlimited number of subitems, and subitems of subitems.

You determine parent and children items by setting the ORDER_SUBVAR and ORDER_ITEMVARS variables in the appconfig.js file.

There are two level entities that describe a configuration:

- The top level is LineItem, a placeholder that contains user information and all items that have been selected from the page set.
- The second level is Items, a placeholder that contains configuration information, and whether the Item is a child of another Item in the LineItem.

The LineItem DTD Elements

Elements identify the nature of the content. The LineItem DTD contains the following elements:

- SiebelLineItem
- User
- Price
- ConfigData
- Item

Attributes are name-value pairs that occur inside tags after the element name. Attributes for each element are listed in the following section.

SiebelLineItem

The SiebelLineItem element is required. It contains information about a line item using the following attributes:

- siebel_id
  This required attribute provides a unique Siebel identifier that the Order Management System will use in its linkback URL.

- created_on
  This required attribute provides the date and time when the line item was created.

- id
  This implied element provides a third-party unique identifier.

The DTD contains the following code for this element.

```
<!ELEMENT SiebelLineItem (User?, Item+)>
<!—siebel_id: Siebel unique identifier —>
<!-- created_on: timestamp -->
<!-- id: 3rd party unique identifier -->
<!ATTLIST SiebelLineItem
   siebel_id ID #REQUIRED
   created_on #REQUIRED
   id ID #IMPLIED
>
```

User

The User element is optional. It contains information that identifies the buyer using the following attributes:

- user_id
  This required attribute provides a unique user identification. This attribute matches the user id property stored in your LDAP system.

- account_id
  This implied attribute provides a unique account identification. This attribute matches the account id property stored in your LDAP system.

- session_id
  When a configuration is passed into Transact on linkback from the Order Management System, this implied attribute provides a unique session identification as a persistent parameter for the OMS.

- order_id
  When a configuration is passed into Transact on linkback, this implied attribute provides a unique order identification as a persistent parameter for the OMS.

The DTD contains the following code for this element.

```
<!-- user element is optional, contains buyer identification
information -->
<!ELEMENT User>
<!ATTLIST User
   user_id CDATA #REQUIRED
   account_id CDATA #IMPLIED
   session_id CDATA #IMPLIED
   order_id CDATA #IMPLIED
>
```

Price

The price element is used for every item where price is defined in the data model. This element contains price information for the line item using these attributes:

- type
  This required attribute defines the type of price.
  Valid values: "quoted" and "discount"

- currency
  This implied attribute defines the type of currency.

The DTD contains the following code for this element.

```
<!--Price type examples: quoted, list, discount -->
<!ELEMENT Price (#PCDATA)>
<!ATTLIST Price
          type     CDATA #REQUIRED
          currency CDATA #IMPLIED

>
```

ConfigData

The ConfigData element is required. This element contains information that defines the configuration or feature data using these attributes:

- name
  This required attribute is the name of the data element in the eAdvisor data model (table name.column name or column name).
  Example: `TABLENAME.DESC`

- character data
  The value of the ConfigData element is the contents of the selected row for the column, indicated by the name attribute information from the data model. In the following example, the character data is the string *Cat*.
  Example: `<ConfigData name = PETTYPE> Cat </ConfigData>`

  The DTD contains the following code for this element.

```
<!ELEMENT ConfigData (#PCDATA)>
<!ATTLIST ConfigData
   name CDATA #REQUIRED

>
```

```
<!— An item must have config data (name/value pairs) associated
with it
    An item can have zero or more items as children
  An item can have zero or more Price elements —>
```

Item

The Item element is required. It contains the information for the item using the attributes:

- part_number
  This implied attribute is the part number for the item. For information on setting the part number variable (called order_subvar), see "Setting the Order_Subvar Configuration Variable" on page 52.

- quantity
  This required attribute is the number of items ordered.

- description
  This implied attribute provides a description of the item.

The DTD contains the following code for this element.
Note that Item can have Price and Item (for nested items) children and must have at least one ConfigData child.

```
<!ELEMENT Item (ConfigData+, Price*, Item*)>
<!ATTLIST Item
  part_number CDATA #IMPLIED
  quantity CDATA #REQUIRED
  description CDATA #IMPLIED

>
```

The Generic LineItem DTD

This section includes the complete code for the generic LineItem DTD.

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE document >
<!ELEMENT SiebelLineItem (User?, Item+)>
<!—siebel_id: Siebel unique identifier —>
<!-- created_on: timestamp -->
<!-- id: 3rd party unique identifier -->
<!ATTLIST SiebelLineItem
   siebel_id ID #REQUIRED
   created_on #REQUIRED
   id ID #IMPLIED
>

<!-- user element is optional, contains buyer identification
information -->
<!ELEMENT User>
<!ATTLIST User
   user_id CDATA #REQUIRED
   account_id CDATA #IMPLIED
   session_id CDATA #IMPLIED
   order_id CDATA #IMPLIED
>

<!--Price type examples: quoted, list, discount -->
<!ELEMENT Price (#PCDATA)>
<!ATTLIST Price
          type     CDATA #REQUIRED
          currency CDATA #IMPLIED
>

<!ELEMENT ConfigData (#PCDATA)>
<!ATTLIST ConfigData
   name CDATA #REQUIRED
>

<!— An item must have config data (name/value pairs) associated
with it
    An item can have zero or more items as children
   An item can have zero or more Price elements —>
<!ELEMENT Item (ConfigData+, Price*, Item*)>
<!ATTLIST Item
   part_number CDATA #IMPLIED
   quantity CDATA #REQUIRED
   description CDATA #IMPLIED
>

]>
```

Using Style Sheets

While an HTML form-based method of integration is available, XML is the standard protocol for interacting with third-party Order Management Systems and Siebel Systems encourages its use.

Transact supports any XML format that contains sufficient information for the integration task. Transact accepts and subsequently returns any data element that it cannot identify—for example, a unique identifier used by the third-party Order Management System.

Transact uses XSLT stylesheets to convert XML generated to the generic DTD into a desired XML format. In the XSLT, elements from the XML input are identified and mapped to their desired XML element output. Static and calculated content can also be included.

About XSLT

XSLT (Extensible Stylesheet Language Transformation) is a language primarily designed for transforming one XML document into another. This is necessary if the application needs to communicate with other systems that accept data in XML but conform to a different DTD. For example, if you need to push your orders to a third-party Order Management System, you must convert your order documents into a format that the Order Management System can process. XSLT is used to develop conversion or mapping rules to modify the generated XML, and these rules are applied against the input XML documents to produce XML documents in the required format. The sample XSLT file provided translates an XML document that conforms to Siebel's DTD to an XML document that conforms to a cXML DTD.

Siebel provides a sample XSLT file (olcp2cXML.xsl) to convert configurations into cXML.

Figure 2 illustrates the flow of an XML document from Siebel's Transact application to a third-party Order Management System. The Order Management System can process XML only if it is in the cXML format.

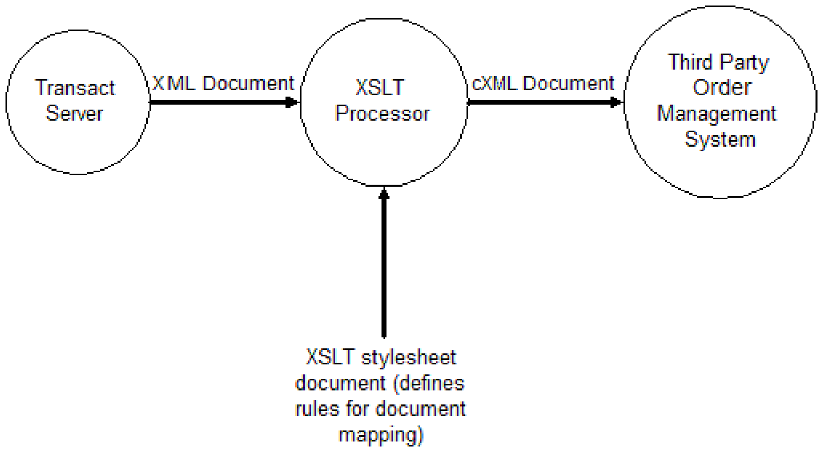

Transact Server
XML Document
XSLT Processor
cXML Document
Third Party Order Management System
XSLT stylesheet document (defines rules for document mapping)

Figure 2. XML to cXML Flow

The XSLT document contains the rules defined by the style sheet developer. The XSLT processor is an engine that reads the XSLT document, applies the rules to incoming input XML documents and, as a result of applying those rules, produces an output XML document that is in cXML format. It can then be understood and processed by the third-party Order Management System. For more detailed information about XSLT, tutorials, and guides, access http://www.xslt.com.

About cXML

cXML stands for Commerce Extensible Markup Language and allows buyers, suppliers, aggregators, and intermediaries to communicate using a single, standard, open language.

cXML transactions consist of documents, which are simple text files with well-defined format and contents. Most types of cXML documents are analogous to hardcopy documents traditionally used in business.

Refer to www.cXML.org for detailed information about the cXML DTD.

About the Siebel DTD

The Siebel DTD defines the format of the XML order document. The order document contains information about the following:

- **Siebel Line Item**
  siebel_id: a unique number that identifies the order
  created_on: specifies the date the order was generated
  id: the OMS unique id, if available

- **User**
  user_id: identifies the user who created the order
  account_id: identifies the account associated with the user
  session_id: identifies the OMS session (if available)
  order_id: identifies the OMS order (if available)

- **Shipping address**
  Name: name of the user (will be the userid in most cases)
  Street: street address including the building number, street name, and apartment number (if any) all as one string
  City: name of the city
  State: a string describing the two-character state code or the full state name.
  PostalCode: a string that specifies the ZIP Code.
  Country: string that contains the country’s name.

- **Billing address**
  Name: name of the user (will be the userid in most cases)
  Street: street address including the building number, street name, and apartment number (if any) all as one string
  City: name of the city
  State: a string describing the two-character state code or the full state name.
  PostalCode: a string that specifies the ZIP Code.
  Country: string that contains the country’s name.

- **Item**
  Part Number: identifies the item
  Quantity: specifies the amount being ordered
  Description: any textual description of the item
  Config Data: describes the configuration
  Price: describes the currency and the price of the item
  Item: any subitems nested within it as children items (subitems contain the same information as the parent)

- **Price**
  Type: specifies the type of the price (discount, retail, preferred, ...)
  Currency: specifies the currency (USDollars, Roubles, ...)
  Value of the entity: denotes the amount in the specified currency

The Sample XSLT Style Sheet

This section describes each section of code from the sample XSLT style sheet (olcp2cXML.xsl).

The following lines of code instantiate the document as an XSLT style sheet.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/
Transform" version="1.0">
```

If the XML output needs to be presented for human reading, the following line indents the XML content with new lines.

```
<xsl:output indent="yes"/>
```

cXML defines selected parameters that must be passed as part of the document. These parameters are constructed outside of the XML style sheet and then passed to the XSLT processor. The following lines extract the values that are passed for those parameters from the calling client and inserts them at appropriate places in the document as defined by the cXML DTD. For more detailed information about cXML and its DTD, refer to the documentation at http://www.cXML.org.

```
<xsl:param name="timestamp"/>
  <xsl:param name="version" select="'1.0'"/>
  <xsl:param name="locale" select="'en-US'"/>
<xsl:param name="from_domain"/>
  <xsl:param name="to_domain"/>
```

```
    <xsl:param name="sender_domain"/>
    <xsl:param name="from_identity"/>
    <xsl:param name="to_identity"/>
    <xsl:param name="sender_identity"/>
    <xsl:param name="sender_secret"/>
    <xsl:param name="user_agent"/>
```

The following line searches for the node named "SiebelLineItem" and applies the rules defined by this style sheet to this node.

```
<xsl:template match="SiebelLineItem">
```

Optional Code

At this point, you can include the following optional line of code.

```
<xsl:attribute name="version"><xsl:value-of select="$version"/
></xsl:attribute>
```

This line specifies the version of the cXML protocol. A validating XML parser could also determine the version attribute from the referenced DTD. However, all cXML documents should include the version explicitly to assist applications using nonvalidating parsers.

```
<xsl:attribute name="xml:lang"><xsl:value-of select="$locale"/
></xsl:attribute>
```

This line specifies the locale used for all free text sent within this document. The receiver should reply or display information in the same or a similar locale. For example, a client specifying xml:lang = "en-UK" in a request might receive "en" data in return.

```
<xsl:attribute name="payloadID"><xsl:value-of
select="$payloadid"/></xsl:attribute>
```

This line specifies a unique number with respect to space and time, used for logging purposes to identify documents that might have been lost or had problems. This value should not change for retry attempts. The recommended implementation is datetime.process id.random number@hostname.

```
<xsl:attribute name="timestamp"><xsl:value-of
select="$timestamp"/></xsl:attribute>
```

This line specifies the date and time the message was sent, in ISO 8601 format. This value should not change for retry attempts. The format is YYYY-MM-DDThh:mm:ss-hh:mm (for example, 1997-07-16T19:20:30+01:00). <xsl:param name="payloadid"/>

The following line instructs the XSLT processor to look for the node "User" in the input document and retrieve the value of its attribute named "order_id," and bind the value to a variable named "order_id." The variable "order_id" can be used many times later in the style sheet to insert the order id into the document.

```
<xsl:variable name="order_id" select="./User/@order_id"/ >
```

At this point, the values needed to construct the header defined by cXML have been obtained from the calling client and the input document. The following lines construct the cXML header portion with the obtained values.

```
<Header>
       <From>
         <Credential>
            <xsl:attribute name="domain"><xsl:value-of
select="$from_domain"/></xsl:attribute>
            <Identity>
              <xsl:value-of select="$from_identity"/>
            </Identity>
         </Credential>
       </From>
       <To>
         <Credential>
            <xsl:attribute name="domain"><xsl:value-of
select="$to_domain"/></xsl:attribute>
            <Identity>
              <xsl:value-of select="$to_identity"/>
            </Identity>
         </Credential>
       </To>
       <Sender>
         <Credential>
            <xsl:attribute name="domain"><xsl:value-of
select="$sender_domain"/></xsl:attribute>
            <Identity>
              <xsl:value-of select="$sender_identity"/>
            </Identity>
            <SharedSecret>
              <xsl:value-of select="$sender_secret"/>
            </SharedSecret>
```

```
          </Credential>
          <UserAgent>
            <xsl:value-of select="$user_agent"/>
          </UserAgent>
        </Sender>
      </Header>
```

The cXML header is followed by the OrderRequest node. The OrderRequest contains OrderRequestHeader and one or more ItemOut nodes.

```
<OrderRequest>
<OrderRequestHeader>
   <xsl:attribute name="type">new</xsl:attribute>
   <xsl:attribute name="orderID"><xsl:value-of
select="$order_id"/></xsl:attribute>
```

cXML defines the total node to contain the total price of the items being ordered as well as the currency. The following lines look for any node named Total in the input document. If more than one are returned, it selects the first one and extracts the price and the currency from it. If no nodes named Total are found in the document, it inserts a default Total node in the output document with the price being 0 and the currency being “USDollars.”

```
<Total>
          <Money>
            <xsl:choose>
              <xsl:when test="./Total">
                  <xsl:attribute name="currency"><xsl:value-of
select="./ Total[position()=1]/@currency"/></xsl:attribute>
                  <xsl:value-of select="./
Total[position()=1]"/>
              </xsl:when>
              <xsl:otherwise>
                  <xsl:attribute name="currency"><xsl:value-of
select="'USDollars'"/ ></xsl:attribute>
                  <xsl:value-of select="0.0"/>
              </xsl:otherwise>
            </xsl:choose>
          </Money>
        </Total>
```

The following lines insert the shipping and billing addresses into the output cXML document.

```
<xsl:apply-templates select="./ShippingAddress"/>
<xsl:apply-templates select="./BillingAddress"/>
```

The following lines insert some Siebel-specific data as extrinsic values that is needed in order to translate the document from cXML back into Siebel XML format.

```
          <Extrinsic>
             <xsl:attribute name="name">siebel_id</ xsl:attribute>
             <xsl:value-of select="./@siebel_id"/>
          </Extrinsic>
          <Extrinsic>
           <xsl:attribute name="name">date_created</ xsl:attribute>
             <xsl:value-of select="./@created_on"/>
          </Extrinsic>
          <Extrinsic>
             <xsl:attribute name="name">id</xsl:attribute>
             <xsl:value-of select="./@id"/>
          </Extrinsic>
     </OrderRequestHeader>
```

Templates

At this point, use the templates below to perform the following actions.

This template searches for nodes named Item in the input document, and starts constructing ItemOut nodes as defined by the cXML DTD to be inserted as part of the output document.

```
          <xsl:apply-templates select=".//Item">
             <xsl:with-param name="itemUser" select="./ User"/>
          </xsl:apply-templates>


       </OrderRequest>

     </cXML>

    </xsl:template>
```

This template defines rules for the XSLT processor to use to search for any node named “ShippingAddress” in the input document and inserts its values into the output document in the format required by cXML.

```
<xsl:template match="ShippingAddress">
    <ShipTo>
     <Address>
      <Name>
       <xsl:attribute name="xml:lang"><xsl:value-of 
select="$locale"/></xsl:attribute>
       <xsl:value-of select="Name"/>
      </Name>
      <Street><xsl:value-of select="Street"/></Street>
      <City><xsl:value-of select="City"/></City>
      <State><xsl:value-of select="State"/></State>
      <Country><xsl:value-of select="Country"/></Country>
      <PostalCode><xsl:value-of select="PostalCode"/></
PostalCode>
     </Address>
    </ShipTo>
  </xsl:template>

  <xsl:template match="BillingAddress">
    <BillTo>
     <Address>
      <Name>
       <xsl:attribute name="xml:lang"><xsl:value-of 
select="$locale"/></xsl:attribute>
       <xsl:value-of select="Name"/>
      </Name>
      <Street><xsl:value-of select="Street"/></Street>
      <City><xsl:value-of select="City"/></City>
      <State><xsl:value-of select="State"/></State>
      <Country><xsl:value-of select="Country"/></Country>
      <PostalCode><xsl:value-of select="PostalCode"/></
PostalCode>
     </Address>
    </BillTo>
  </xsl:template>
```

This template defines rules for constructing and inserting item information from the input document into the output document. The ItemOut as defined by cXML consists of nodes that contain information about the price of the item (including the currency), the quantity being ordered, the item line number, the part number of the item, and so on. However, Siebel's XML input document contains more information that may not be needed by cXML-compliant applications, but is very necessary in order to reconstruct the Siebel XML document given to the cXML document. For this reason, any Siebel-specific information is saved as Extrinsic values so that it can be used later on (for example, to reconstruct the original Siebel XML document). One example would be Configuration data that may not be used by cXML but is necessary for use in Siebel Transact applications.

```
<xsl:template match="Item">
     <ItemOut>
       <xsl:attribute name="quantity"><xsl:value-of
select="@quantity"/></xsl:attribute>
      <xsl:attribute name="lineNumber"><xsl:value-of select="../
@id"/></xsl:attribute>
       <ItemID>
         <SupplierPartID>
          <xsl:value-of select="@part_number"/>
         </SupplierPartID>
       </ItemID>
       <ItemDetail>
          <UnitPrice>
            <Money>
              <xsl:choose>
                  <xsl:when test="./Price">
                   <xsl:attribute name="currency"><xsl:value-of
select="./ Price[position()=1]/@currency"/></xsl:attribute>
                 <xsl:value-of select="./ Price[position()=1]"/>
                  </xsl:when>
                  <xsl:otherwise>
                   <xsl:attribute name="currency"><xsl:value-of
select="'USDollars'"/ ></xsl:attribute>
                    <xsl:value-of select="0.0"/>
                  </xsl:otherwise>
              </xsl:choose>
            </Money>
          </UnitPrice>
          <UnitOfMeasure></UnitOfMeasure>
          <Description>
            <xsl:value-of select="@description"/>
          </Description>
          <xsl:if test="$itemUser">
```

```
                  <Extrinsic>
                     <xsl:attribute name="name">userID</ xsl:attribute>
                     <xsl:value-of select="$itemUser/@user_id"/>
                  </Extrinsic>
                  <Extrinsic>
                   <xsl:attribute name="name">accountID</ xsl:attribute>
                     <xsl:value-of select="$itemUser/ @account_id"/>
                  </Extrinsic>

                  <Extrinsic>
                   <xsl:attribute name="name">sessionID</ xsl:attribute>
                     <xsl:value-of select="$itemUser/ @session_id"/>
                  </Extrinsic>
                 </xsl:if>
                 <xsl:apply-templates select="./ConfigData"/>
              </ItemDetail>
            </ItemOut>
         </xsl:template>

         <xsl:template match="ConfigData">
           <Extrinsic>
             <xsl:attribute name="name"><xsl:value-of
       select="concat('cfg_', @name)"/></xsl:attribute>
             <xsl:value-of select="."/>
           </Extrinsic>
         </xsl:template>
```

This line is the closing tag for the style sheet.

```
</xsl:stylesheet>
```

Linking Back to Configurations

When users work in the Siebel eAdvisor application, they select items and then click the Add to Cart button to save them to the shopping cart. When they have selected all the items they want from a page set, they can click the Save Configuration button to save all the items in their cart as a single configuration. When the configuration is saved, the Siebel application passes it to Transact to save in its database. An order can contain multiple configurations.

If desired, you can provide a link back to configurations in eAdvisor from the Order Management System shopping cart. The Configuration list also provides linkback capabilities by default. A third-party transactional system will be able to send a request to Transact to restore the configuration state in the Siebel application, which was previously submitted to the third-party system. You can do this in two ways:

- From a URL you provide in your shopping cart
- From the Configuration List URL

From the Shopping Cart

To create a link back from the shopping cart to a configuration, in the OMS shopping cart UI, you must add a link to the Siebel application URL with the ID of the configuration.

Example:

```
http://eadvisor//home.htm?config_id=<siebel_id attribute of
Siebel Line Item from XML>
```

For better performance, if you know the Siebel application will be running whenever the Order Management System is running, you can directly call into the Siebel application instead, using an OnClick event:

```
<top.ol>.OpenFromURL (‘config_id=<ID>’);
```

See the documentation for your Order Management System for instructions on adding UI elements.

From the Configuration List

When you click on a configuration in the Configuration list, Transact links back to the configuration by calling getRestoreURL, which provides the URL of the configuration.

To create a link back to a configuration from the Configuration list

1. Add a URL to the Configuration List page.
2. Use getRestoreURL for the href property.

   See "getRestoreURL" on page 181 for information on implementing this function.
3. Call RestoreConfig on the onClick event and send "this."

   See "RestoreConfig" on page 157 for information on implementing this function.

Version Checking

When you open a configuration from the configuration list, the unique key for the configuration is passed to the server, which it uses to get the configuration data. The data is then moved into the Siebel application, and your inputs and results pages are displayed exactly the way they appeared when you saved the configuration.

You can implement the following two callout points that cause an event to occur when a configuration links back to an old version:

- COP_AppDataVersionCheck
- COP_PagesetVersionCheck

Define these callout points to return false if the application or page set is out of date. This prevents them from opening. You can implement a dialog box that returns a message such as "This item may be out of date. Do you want to continue?" If the user clicks Yes, the function returns true and the application or page set opens. If the user clicks Cancel, the function returns false and the application or page set does not open.

See "COP_AppDataVersionCheck" on page 162 and "COP_PagesetVersionCheck" on page 166 for more information.

Authentication and Login Support 5

Using an LDAP system, your Siebel application provides support for user authentication and login. When a user attempts to access restricted data, a login page opens that prompts the user to enter a name and password. You can customize the look of the login page by editing LoginPage.jsp. This file is located in the WebLogic/myserver/public_html/TransactServer directory.

To set up users, permissions, and resources, you execute calls from the command line. See "Command-Line API" on page 114 for more information.

The following sections describe the LDAP architecture, the properties to set, and the calls to make in order to set up your LDAP system.

NOTE: The LDAP login and authentication features are optional.

About LDAP

Lightweight Directory Access Protocol (LDAP) is a software protocol that allows users to locate organizations, individuals, and other resources such as files and devices in a network, whether on the Internet or on a corporate intranet. LDAP is a "lightweight" (smaller amount of code) version of DAP (Directory Access Protocol), which is part of X.500, a standard for directory services in a network. LDAP is lighter because in its initial version it did not include security features.

LDAP is both a naming server and a directory server. The naming server provides the capability to name something or associate a set of data with a name. For example, in a file system we create files and give them names. The directory server provides the capability to arrange and store these objects in some structured way to avoid name collision and for easier retrieval.

An LDAP directory is organized in a simple "tree" hierarchy consisting of the following levels listed in order of parent to child:

- The root directory
- Countries
- Organizations
- Organizational units (divisions, departments, and so forth)
- Individuals (including people, files, and shared resources such as printers)

An LDAP directory can be distributed among many servers. Each server can have a replicated version of the total directory that is synchronized periodically. An LDAP server is called a Directory System Agent (DSA). An LDAP server that receives a request from a user takes responsibility for the request, passing it to other DSAs as necessary, but providing a single coordinated response for the user.

When you create your LDAP directory structure, you may have multiple organizations.

LDAP Models

The LDAP information model defines the types of data and basic units of information you can store in your directory. The basic unit of information in the directory is the entry, a collection of information about an object. Often, the information in an entry describes some real-world object, such as a person. In a typical directory, you find thousands of entries that correspond to people, departments, servers, printers, and other real-world objects in the organization. Figure 3 shows part of a typical directory, with objects corresponding to some of the real-world objects in the organization.

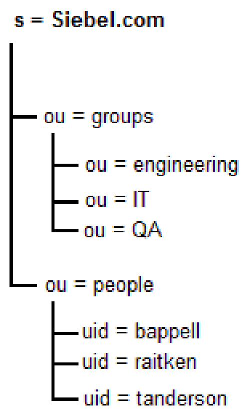


Figure 3. Sample LDAP Directory

Where:

O	Organization
OU	Organizational unit
UID	User ID

An entry is comprised of a set of attributes, each of which describes one particular trait of the object. Each attribute has a type and one or more values. The type describes the kind of information contained in the attribute, and the value contains the actual data.

For example, an entry for a person may have the following attribute types and values:

Attribute	**Description**	**Example**
cn	Common Name	Rick Aitken
sn	Surname	Aitken
password	User ID	lucky456
mail	eMail Address	raitken@ Siebel.com

You can create your own objects and attributes that are meaningful to your organization.

Default LDAP Directory

During the installation process, a default directory structure, eAdvisor, is created as an organization unit under the Organization. Under eAdvisor, there are:

- Permissions
- ACL (resources)
- Quote users (ol_quoteusers)
- Business accounts (ol_bizaccounts)

Permissions

This directory contains all the permissions that used by WebLogic and the Siebel application.

ACL (Resources)

This directory contains the resources that might need restricted access. The resources could be servlets, JSPs, connection pools, and so on. Each ACL contains one or more ACL entries under it. A single ACL entry contains a user or a group and a list of permissions for that user or group to the given ACL.

For information on adding resources, see "New ACL" on page 115.

Quote Users

This directory contains one or more entries. Each entry contains a user, the quotes group that the user is associated with, and a permission of either transactSuper or transactGroup.

Business Accounts

This directory contains the list of customer accounts. Each account contains information about the account, such as users associated with the account and billing/shipping addresses.

Login Pages

Various pages fall under the category of login pages.

Login Page

When a user attempts to access restricted data, a login page opens, prompting the user to enter a name and password. You can customize the look of the login page by editing LoginPage.jsp. This file is located in the WebLogic/myserver/public_html/Login directory. You can also use your own login page by setting the Login Page property in the Application Server tab of the property editor.

After Login

Login Success Page

When a user logs in successfully, by default the system performs the requested action. When logging into the application for the first time, or in cases where you want to display information on login, you can redirect the user to your own login success Web page. To use your own page, edit the Login Success Page property on the Transact tab of the property editor.

Login Error Page

When a user fails to log in successfully, the system redirects the user to a login error page. By default, Siebel displays the message “Sorry, we didn't recognize your login name and/or password. Please try again.” You can modify the LoginError.jsp page, or set the Login Error Page property on the Application Server tab of the property editor to redirect to your own error page.

Setting LDAP Properties

If you change properties on your Application server, you must modify the Siebel Application Server properties in the property editor so that they match the new settings in the Application server property files.

LDAP Passwords

You can change the WebLogic user system's LDAP password or the LDAP user directory manager's password.

WebLogic User System's Password

If you change this password, you must make sure that your user system's password in the LDAP system matches the one in the WebLogic.properties file. You cannot set this property in the property editor. In the WebLogic.properties file, edit the following line (all WebLogic servers require this line of code) that specifies the password for the user system:

```
WebLogic.password.system=lucky68
```

If you change this password in the WebLogic.properties file, you must change it in the LDAP system as well. Use the directory console (the GUI tool) provided by the LDAP server to change the password for the user system in the LDAP server.

NOTE: All WebLogic servers sharing the same LDAP system must have the same password for the user system.

LDAP User Directory Manager's Password

During installation, the Netscape LDAP directory server requires that you specify a password for the user directory manager. Update the property LDAP Credential in the property editor with the same password. The script file uses this user (Directory Manager and its password) to log in to LDAP and make changes.

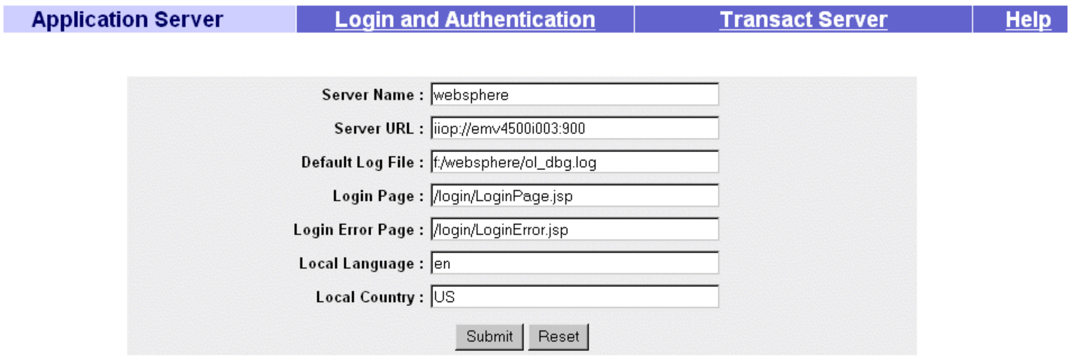
Setting Properties

If you change properties on your Application server (WebLogic), you must modify the Siebel Application server properties in the property editor so that they match the new settings in WebLogic.

To set properties in the property editor

1. Start WebLogic.
2. Open a browser and enter the following URL (modify the URL as needed for your system settings):

   http:// < localhost:7001 > /PropertiesEditor/index.htm
3. Click Application Server.



Server Name :	websphere
Server URL :	iiop://emv4500i003:900
Default Log File :	f:/websphere/ol_dbg.log
Login Page :	/login/LoginPage.jsp
Login Error Page :	/login/LoginError.jsp
Local Language :	en
Local Country :	US

4. Click Help for a description of the Application Server properties.

   After setting the Application Server properties, you will need to set the Login and Authentication properties.

5 Click Login and Authentication.

Since the LDAP Credential is encrypted, you have to remember to reset this field to the correct value if you modify any field from authentication property.

Click Help for a description of all properties.

NOTE: The property editor does not check for invalid values, so be precise when you enter a property value.

Command-Line API

During installation, the initial_login_setup script creates the necessary LDAP directory structure. This file uses one or more of the command-line functions. The initial_login_setup script file inserts security information that is required by WebLogic. Transact setup adds sample users and accounts. This gives WebLogic the basic information it needs to start up correctly.

The information in LDAP needs to be updated and maintained. You must write script files using the command-line functions to add, delete, and modify users, groups, ACL, and permissions. In addition, you can enroll users as members of selected groups, which automatically gives them the permissions for that group. See the following section, "Sample Script Templates" on page 117, for sample script templates you can use to base your script on.

You can implement the following functions in your script.

Add Group Member	cmd = addgroupmember;tgtgroup = everyone;user = john This command adds an existing user to the specified group. In the example above, this command adds user john to the target group everyone. cmd = initload must be run before running this command. See "Init Load" on page 115 for more information.
Add Owner	cmd = addowner;owneruser = system;theuser = john;acl = WebLogic.properties.LoginProcess Each ACL must have an owner. The owner is specified when the ACL is created. The owner has the rights to add permissions to a user for the ACL, add other users as owners, and delete the ACL. This command adds the user john as one of the owners to the ACL WebLogic.properties.LoginProcess after making sure the owner user is system.
Create Context	cmd = createcontext This command creates a directory structure in the LDAP system. If the structure already exists, the command throws an exception. This command is used by the initial_login_setup script file that is run during installation to create the structure of the LDAP system.
Delete ACL	cmd = delacl;user = john;acl = WebLogic.servlet.LoginProcess This command deletes the specified ACL. The specified user must have the owner rights to delete an ACL. In the example above, the ACL WebLogic.servlet.LoginProcess is deleted by the user john.

Delete Context	cmd = deletecontext This command deletes the directory structure that was created by the createcontext in the LDAP system.
Delete Group	cmd = delgroup;group = everyone This command deletes groups from the LDAP system. In the example above, the group everyone is deleted from the LDAP system.
Delete Group Member	cmd = delgroupmember;tgtgroup = everyone;user = john This command deletes group members from the LDAP system. In the example above, the user john is removed from the group everyone.
Delete Owner	cmd = delowner;owneruser = system;theuser = john;acl = WebLogic. properties.LoginProcess This command deletes ACL owners from the LDAP system. In the example above, the user john is removed from being the owner of the ACL WebLogic.properties.LoginProcess. The owneruser must be an owner of the ACL in order for this command to execute successfully.
Delete Permission	cmd = delpermission;permission = write This command deletes the specified permission from the LDAP system. In the example above, the permission write is deleted from the LDAP system.
Delete User	cmd = deluser;user = john This command deletes users from the LDAP system. In the example above, the user john is deleted from the LDAP system.
Init Load	cmd = initload This command instructs the program to load the data in the LDAP system into the cache. This command needs to be run before the following command can be run.
New ACL	cmd = newacl;acl = LogOut;user = john This command creates a new ACL with john as its owner. Only the owner/s will be able to add users/groups as users/owners with permissions to the ACL. The ACL is a resource that needs restricted access, and it contains data as to who (what users) can access the resource and how the permissions can use it. You must log on as directory manager (cn = Directory Manager) to use this command. The directory manager is set in the LDAP Principal and LDAP Credential properties in the property editor.

New Group

cmd = newgroup;group = everyone

This command creates a new group in the LDAP system. In the example above, this command creates a new group called everyone in the structure ou = Groups, o = your_org.com. In the course of maintenance, Siebel application users will need to add and remove groups. This API will help in creating groups relevant to your organization.

New Permission

cmd = newpermission;permission = read
cmd = newpermission;permission = write

This command creates a new permission in the LDAP system which can then be used to limit access to resources. In the example above, these commands create new permissions in the LDAP structure cn = permissions, cn = eAdvisor,o = your_org.com.

New User

cmd = newuser;user = john;credential = john

You can add users using this command or by using the GUI tool provided by the LDAP server.

This command adds a new user and associated password to the LDAP system. In the example above, this command creates a new user called john with john as the password in the structure ou = People, o = your_org.com.

You must log on as directory manager to use the NewUser API. The directory manager is set in the LDAP Principal and LDAP Credential properties under Login and Authentication in the property editor.

The user “guest” created by the Transact LDAP setup is reserved for internal use by Transact. You cannot log into Transact using the “guest” user.

Set Permission

cmd = setpermission;acl = LogOut;type = allow;permission = *;for
user = john;owneruser = system
cmd = setpermission;acl = LogOut;type = allow;permission = *;
forgroup = everyone;owneruser = system

The above commands create a record under the given ACL for the given user or group that specifies the specific permissions that the given user or the group has to the specified ACL.

The type contains one of two values: allow or disallow. If allow is specified, the specified permissions are positive, which means the user is allowed those permissions. If disallow is specified, the permissions are negative, which means the user is denied those permissions.

You must log on as directory manager to use the SetPermission API. The directory manager is set in the LDAP Principal and LDAP Credential properties in the property editor.

Sample Script Templates

Several templates are available that simplify LDAP installation and maintenance.

To save and run the following script templates

1. Save the code in a file.

   For example, addservlet.sci under the WebLogic home directory.

2. In the property editor, set the LDAP URL to your LDAP Server and the LDAP Credential to "Directory Manager."

3. Run the script from the command line by typing the following line (replacing addservlet.sci with the name of your script file) at the prompt:

   Rincon_run addservlet.sci

Supporting Adding Servlets to WebLogic

This script provides the support for WebLogic to accept new servlets. The procedure used in this script could be used to add any kind of resource to WebLogic that requires granting user permissions to it. This is a specific example of adding a servlet which is a resource. This script adds a servlet called Logout that handles logging the user out gracefully. The Logout servlet receives the request from the client browser when the user decides to log out by clicking a button or a URL link, and then deallocates any resources assigned to the user.

The first step is to add the servlet to the WebLogic.properties file to instruct WebLogic to load the servlet. To do this, add the following line in the WebLogic.properties file. (The package name of the servlet is com.siebel.isscda.wl.servlet.)

WebLogic.httpd.register.Logout = \com.siebel.isscda.wl.servlet.rinco n.Logout

Because the servlet is a resource, WebLogic needs to know which users have what permissions to this resource. Since WebLogic queries LDAP for all security information relating to users, groups, and ACLs (resources), this new servlet needs to be added as an ACL into the LDAP system as well and users granted permissions to this resource. The following template uses the command-line API to do this.

The Script

```
cmd=newacl;acl=WebLogic.servlet.Logout;user=system
cmd=initload
cmd=setpermission;acl=WebLogic.servlet.Logout;type=allow
;permission=*;forgroup=everyone;owneruser=system
end
```

About the Script

The first command creates an ACL named WebLogic.servlet.Logout with the user system as the owner of the ACL. Because we want to modify the data in the LDAP system, the second command (cmd = initload) loads the data from the LDAP system into memory. The third command gives all permissions that are positive to the group *everyone*. Now anyone belonging to the group *everyone* automatically has the permissions assigned to the group. The owneruser must be the owner of the ACL in order for the setpermission command to execute successfully. The last command (end) instructs the program that executes the script file to terminate itself.

When WebLogic loads this servlet, it gets the security access information to this servlet from the LDAP system, and implements it accordingly.

Adding Users and Groups

This script template adds a list of users and groups. It then adds the users and groups as members of other groups.

The Script

```
cmd=newuser;user=john
cmd=newuser;user=mary
cmd=newuser;user=kate
cmd=newgroup;group=Engineering
cmd=newgroup;group=Marketing
cmd=initload
cmd=addgroupmember;tgtgroup=Engineering;user=john
cmd=addgroupmember;tgtgroup=Engineering;user=kate
cmd=addgroupmember;tgtgroup=Marketing;user=mary
end
```

About the Script

The users *john* and *kate* automatically get the same kind of access as the *Engineering* group. Mary gets the same privileges as the *Marketing* group.

Deleting Users and Groups

This script template deletes a list of users and groups, and removes users from groups.

The Script

```
cmd=deluser;user=john
cmd=deluser;user=mary
cmd=delgroupmember;tgtgroup=Engineering;user=kate
cmd=delgroup;group=Engineering
end
```

About the Script

The users *john* and *mary* are deleted from the LDAP system while the user *kate* is removed from the *Engineering* group (although she will remain in the LDAP system). Finally, the group *Engineering* is deleted from the LDAP system.

The Shopping Cart 6

Once the shopping cart is installed, the Transact API calls must be inserted into the browser-based application. These calls add items to the cart and bring up the cart UI on demand. These calls should be put into the onClick event of a button or link, which appears on the browser-based UI. See "Prepare the Application to Connect with Transact Server" on page 51.

Setting Transact Shopping Cart Properties

You should have set most of these properties during either the shopping cart or the Transact Server installation, but verify that the following settings contain the correct values to use with the shopping cart shelf integration.

App_config.js variables

TRANSACT_ACTIVE	This Boolean variable should be set to the value 'true'.
TRANSACT_URL	This string variable should be set to the URL of your Transact Server installation. Example: "http://app_hostname/siebel/SiebelTransact";
TRANSACT_THIRD_PARTY_CART	This Boolean variable should be set to the value 'false'.
TRANSACT_CART_WINARGS	This string variable should contain the window properties to be used for the shopping cart. The format of the string follows the JavaScript convention for setting window properties. The string should not contain any space characters. Example: 'height = 500,width = 500,scrollbars = 1,resizable = 1,menubar = 0'
TRANSACT_CART_TARGET	This variable should contain a string specifying the frame name where the cart UI should be loaded. Use '_new' to indicate that a new window should be opened. If you elect to show the cart in a new window, it will exhibit window properties as defined by the TRANSACT_CART_WINARGS variable above. Example: '_new'

Properties Editor Properties

Use Third-Party Cart	This Boolean property should be set to the value 'false'.
OMS URL	This string property should be set to contain the URL of the order management system that is prepared to receive shopping cart submissions from the server-side shopping cart. Example: http://oms.mycompany.com/oms_acceptor
Transact URL	This property should contain the same value as the TRANSACT_URL variable in app_config.js.
Sales URL	This property should contain the URL of your CDA application, without any HTML page (like home.htm) specified. Example: http://www.mycompany.com/sales.

Data Modeling For Transact Server

For Transact Server to be able to display line items on the shopping cart, it must first determine what data in the data model pertains to products. This process is very simple, and the key to it is a variable in the user-modifiable application configuration file app_config.js. The variable is called ORDER_SUBVAR. Enter as a value for this variable the column name that you use to store product part numbers in the data model. Make a column with this name in the main configuration table of the data model. Each row of the MAIN configuration table corresponds to the part number specified in the ORDER_SUBVAR column and will be interpreted by Transact as a potential attribute of that part.

To specify subparts (children of the product in the MAIN table), use the same column name specified in the ORDER_SUBVAR variable in feature tables that hold information related to the subparts. When a row in the feature table is selected as part of the configuration results, the part number will be passed to Transact as a selected product. The other columns in the selected row of the feature table will be passed to Transact as fields relating to that selected product. The selected row in the MAIN configuration table will be the parent of the selected product.

If you specify an asterisk character (*) as the value for the ORDER_SUBVAR variable, all of the selected rows in all tables will be passed to Transact with the MAIN table being the root product and the feature tables comprising the set of children products. No part numbers for these products will be identified by Transact Server, however.

Parts of the Cart

The shopping cart can be thought of as having three distinct sections:

- *Header*

  This section displays general information pertaining to the entire quote. The header may contain address information, account information, or other fields created for use by the customer.

- *Quote line items*

  This section contains information about the products in the cart. The shopping cart template defines the display format for a single line item. A while loop is used to interpret this display format for each line item. A line item has a pageset name and description associated with it, and has the ability to linkback to its pageset, restoring the state of the configuration to what it was when the user clicked Add To Cart. Each line item may contain multiple parts. A part generally has a part number, description, price, and various feature table data associated with it. Parts can have parent-child relationships, which in previous versions of browser-based applications were described as item-subitem relationships. In the 4.0 version of browser-based applications, unlimited nesting of parts is supported.

- *Footer*

  This section displays subtotal and total information, and contains buttons that operate on the cart. These actions are available:

  - Clear Cart starts a new cart
  - Submit Cart submits the cart to the specified order management system
  - Save Cart assigns a name so that the cart can be accessed from the quote list
  - Print Cart displays a formatted, view-only cart page
  - Update Cart applies user changes

How It Works

When the user clicks Add To Cart, the configuration is saved as it normally would be in the Transact database, and then the configuration is associated with the cart for the user’s current session. The Shopping Cart.jsp template then displays the cart. All the information about the cart is retrieved from the Transact database. Then, the specific information referenced in the ShoppingCart.jsp template is easily accessed and incorporated into the static HTML. Once the ShoppingCart.jsp template has been executed, only HTML (and JavaScript) is contained in the page displayed to the user.

To allow the user to update the contents of the cart, the ShoppingCart.jsp template must contain an HTML form. Any updatable fields on the cart need to be represented using form elements such as text boxes and drop-down picklists.

Writing the JSP Template

The file ShoppingCart.jsp is located in the directory {WEBLOGIC_HOME}/myserver/public_html/transactserver. This file specifies the UI for the server-side shopping cart. The ShoppingCart.jsp template should first and foremost be a valid HTML document. Dynamic content and shopping cart functionality are provided using JSP tags that are embedded in the HTML.

The first JSP statement appearing in the file is:

```
<%@ page
        info="Shopping Cart"
        contentType="text/html"
%>
```

JSP tags always begin with "< %" and end with "% >". Once the JSP compiler has encountered the tag above, it looks for and processes all subsequent JSP tags. The info and contentType attributes should be self-explanatory. The page tag should appear somewhere before the <body> tag of the ShoppingCart.jsp page.

There are two important kinds of JSP tags used throughout the shopping cart template. These tags come in pairs and demarcate code statements or blocks.

The Code Block Pair

One is a plain-looking pair that begins with "< %" and ends with "% >". These tags demarcate a code statement or code block that needs to be executed without the result appearing on the page. You can think of these tags as almost the equivalent of "<script>" and "</script>" tags in JavaScript. Standard Java statements, like conditional "if" statements and loops, are wrapped in these tags. To insert HTML into a code block and have it appear on the page, simply end the code block with the "% >" end tag, put in the desired HTML, and then start the code again with another "< %" open tag. This makes the JSP more powerful than the JavaScript equivalent. So, you can have HTML appear as the result of a conditional statement as follows:

```
<p> The weather is <% if (sky.equals(“blue”)) { %>  <b>clear!</b>
<% } else { %> <i>cloudy.</i> <% } %> <br>
```

If the String variable sky contains the value "blue" in the example above, the following HTML will be produced:

```
<p> The weather is <b>clear!</b> <br>
```

The Single Java Statement Pair

The other JSP tag used throughout the ShoppingCart.jsp template begins with " < % = " and ends with " % > ". This tag may contain only a single Java statement, which does not need a semicolon at its close. The " < % = " tag tells the compiler to evaluate the single statement and print the result on the page, in the same location where the tag appears. A usage example of this tag is as follows:

```
<p><b>Welcome, <%= user_id %>!</b>
```

If the variable user_id contained the String value "Frank," the resulting HTML would appear as shown below:

```
<p><b>Welcome, Frank!</b>
```

This is nearly the extent of the JSP language you need to know to use and understand the ShoppingCart.jsp template. A little more JSP syntax is introduced in the next section. For further information on JSP syntax, see Sun's JavaSoft Web site, which has helpful tutorials.

Using the Shopping Cart Bean

The ShoppingCartBean class provides the API that is used to access all the information in the cart.

Directly following the `“<%@ page`” statement, there occurs the JSP statement:

```
<jsp:useBean id="bean"
class="com.siebel.isscda.wl.transact.ShoppingCartBean">
   <%
     bean.setRequest(request);
     bean.setSession(session);
     bean.getQuote();
   %>
</jsp:useBean>
```

The “jsp:useBean” tag just includes the ShoppingCartBean class, which implements the API that is used to access all the data which must be displayed in the shopping cart. The “id = bean” indicates that the ShoppingCartBean class will be referred to using the name “bean.” If you change this to a different name, replace every instance of “bean” in the template with your new name. The class = ”com.siebel.isscda.wl.transact.ShoppingCartBean” part of the useBean tag gives the fully qualified name of the ShoppingCartBean class to the JSP compiler so that it is able to find it.

Before the useBean end tag “ < /jsp:useBean > ,” there is a code block bracketed by “ < % ” and “ % > ”. Since “bean” is the name given in the useBean statement to refer to the ShoppingCartBean, bean is used to call into the ShoppingCartBean API. The specific statements in this block of code are initialization statements that must be executed before making any further ShoppingCartBean API calls. These statements are used inside the useBean tags to make sure that they are called directly after the bean is included in the page.

The first statement in the code block above is “bean.setSession(session);”. This calls the ShoppingCartAPI function setSession and passes the variable session. The variable session is implicitly made available in all JSP pages (Common JSP behavior). The variable session refers to the HttpSession object defined by the Java class com.java.servlet.http.HttpSession (itself part of the Java servlet API). This session object is specific to the current user’s session and can be used to store and retrieve the session state. Transact Server has put a session state into the session object, and so the session is passed into the setSession method of the ShoppingCartBean. This gives the ShoppingCartBean API access to this state information.

The format of the second statement,“bean.setRequest(request);”, is similar to that of “bean.setSession(session);”. Request is another variable which is implicitly made available to all JSP pages. This statement passes the request variable to the ShoppingCartBean, giving it access to the information contained therein. The request variable refers to the HttpRequest object generated for the HTTP request that brought the user to the ShoppingCart.jsp template. The request variable contains the usual information encapsulated by an HTTP request: any HTML form variables, URL parameters, and the URL itself being the most useful.

The final statement, “bean.getQuote();”, gets the information about the current quote from the database. Once this statement is executed, you can start accessing information about the quote, as explained in the following section.

Accessing Header Data

You can access data in the Quote Header or custom fields in the Quote Header.

Quote Header Access Functions

Once the getQuote function has been called, data in the quote can be accessed and displayed using the QuoteBean API functions. Quote Header information is identified by Transact as the cart form parameters (fields) whose names begin with "HEADER_".

The functions that provide access to quote header data have names beginning with "getQuote" and return a String. Examples are getQuoteName, getQuoteID, and getQuoteAccountID. For the quote name, you must use the form parameter name "QUOTE_NAME". A code-level example of this technique is as follows:

```
<td><font face="VERDANA" size= -2>
<b>Quote Name:</b>
</font></td>
<td><font face="VERDANA" size= -2>
<input type="text" name="QUOTE_NAME" value="<%=
bean.getQuoteName() %>">
</font>
```

Certain fields in the header are not updatable, because they should not be changed. These fields are as follows: account, date created, date modified, user_id.

Customer-Created Quote Header Fields

You can add custom fields to the header by adding a form element and giving it a name prefixed with "HEADER_". To retrieve the value entered into that field, use the function getQuoteHeader and pass in the name of the field without the "HEADER_" prefix. To add a Notes field onto the header, for example, you would include a new form element and prefill the value with whatever the user had previously entered using the getQuoteHeader function as follows:

< input type = "textbox" name = "HEADER_NOTES" value = " < % =
bean.getQuoteHeader("NOTES") % > " >

When anything is entered into the HEADER_NOTES field it will automatically be saved with the quote, and the value entered will be shown on the quote by virtue of the getQuoteHeader("NOTES") function call.

Example of Defaulting Address Info from LDAP

The Transact LDAP structure can contain account and address information accessed by Transact for display on the Shopping Cart or for inclusion in exported XML. For further explanation of Transact's usage of LDAP, refer to the sections on LDAP in this guide. Transact accesses LDAP information in a read-only manner. Yet, information provided by the LDAP system for default account billing and shipping information may need to be modified for use on an individual quote. If you are using LDAP default information on the Shopping Cart, display it only if there is no quote-specific information already stored. Quote-specific information is stored in a custom header field. An example of this technique is shown below:

```
<% if (bean.getQuoteHeader("SHIPPING_ADDR") != "") { %>
   <td colspan=2><textarea rows=5 name="HEADER_SHIPPING_ADDR">
   <%=  bean.getQuoteHeader("SHIPPING_ADDR") %>
</textarea></td>
<% }else { %>
<td colspan=2><textarea rows=5 name="HEADER_SHIPPING_ADDR"><%=
bean.getShippingName() %>
<%= bean.getShippingStreet() %>
<%= bean.getShippingCity() %>, <%= bean.getShippingState() %>
<%= bean.getShippingZip() %>
<%= bean.getShippingCountry() %>
</textarea></td>
<% } %>
```

In the above example, we first check for the existence of user-entered data in the quote header field "SHIPPING_ADDR." If the user has entered data into this field on the quote, then we display this information in a textarea, using a call to the getQuoteHeader function to access the entered data. If no data has been entered into the "SHIPPING_ADDR" header field, we display formatted default information from LDAP using calls to getShippingName, getShippingStreet, getShippingCity, and so on to retrieve the information from the LDAP server.

Accessing Line Item Data

Since there are potentially multiple line items in a quote, a standard iteration is used to access each line item in succession. The format is always:

```
<%
    while ( bean.hasMoreLineItems() )
    {
             bean.nextLineItem(); %>
```

The while statement begins the iteration. It allows the next statements to be repeated until the condition inside the parenthesis is false, exactly as it would be in JavaScript or any other programming language. The condition statement is a call to the ShoppingCartBean named hasMoreLineItems, so the loop will continue until we have gone through all the line items in the quote. The first statement inside the loop is another call to the ShoppingCartBean to get the next line item, named nextLineItem.

Once inside this loop, any information about the current line item may be accessed. The loop is ended with an end curly bracket within the JSP code block tags `<% } %>`. Line item index (a counter of which line item is being displayed), linkback URL, and line item description are a few of the pieces of line item information which you may want to display on the cart. A full listing of function calls providing line item data access are given in the API Appendix.

Custom fields may not be created at the line item level, but they may be created at the part level, which provides essentially identical functionality.

Accessing Part (Subitem) Data

Just as a quote consists of multiple line items, a line item consists of multiple parts. And, using a method similar to the one in which we iterate through the line items in the quote, we must iterate through the parts for the line item. Since the parts belong to the current line item being accessed in the line item loop, we iterate through them within the line item loop, producing a nested loop. This may sound complicated, but it is actually fairly simple. The code sample in the above section simply expands to the one shown below:

```
<%
    while ( bean.hasMoreLineItems() )
    {
        bean.nextLineItem();
%>
<!-- Line Item specific HTML goes here -->
<%
        while ( bean.hasMoreParts() )
        {
            bean.nextPart();
%>
<!-- Part specific HTML goes here -->
```

From within the part section of the cart, you can make calls into the ShoppingCartBean API to get specific information about that part to display on the shopping cart interface. Examples of part information that are provided through the API include description, price, and part number.

Feature table data is also available through the ShoppingCartBean API. It is accessed using a methodology similar to the retrieval of custom-defined header fields. You can use the function getPartField and pass in the name of the column, and the function will return the corresponding value. An example is shown below.

```
<%= bean.getPartField(“DESC”) %>
```

You can also create part-level custom fields using this function. In other words, if the field name you provide to the function is not one that is stored for that part, it will create it for you and any data the user may enter for that field (if it is editable) will be stored with the cart. Any changes that are made to part level data through the shopping cart will not be reflected on linkback. The data used for linkback is stored separately and is not editable through the shopping cart, in order to guarantee configuration validity on linkback.

Since feature tables differ across pagesets and there is only a single cart interface to display the myriad pagesets, the ShoppingCartBean includes an API function which returns all matching field names in the feature table data for the current part.

For example, if you had a “PRM” column in a feature table in all of your pagesets, but the feature tables were all named differently, you would use this function to access the column so that you would not need to specify the table name (“TABLE.PRM”).

The function is called getMatchingPartFields. Since this function returns all matching field/value pairs, more than one value may be returned. To allow for this possibility, the function returns an array of key value pair objects. The objects have a key field and a value field which are both Strings. You access the array, and the fields in the objects, exactly as you would in JavaScript.

An example is shown below. For a more detailed example, see the getMatchingPartFields entry in “getMatchingPartFields” on page 214.

```
<%  KeyValuePair [] matches =  bean.getMatchingPartFields(“PRM”);
if (matches.length > 0)  {
%>
<td>PRM: <%=matches[0].value %> </td>
<% } %>
```

The first statement in the example declares an array of KeyValuePair objects as the return value for the getMatchingPartFields function call. This array is populated by any and all fields and values which have “PRM” as part of their field name. The next statement looks to see if any matching fields were returned for this part. “matches.length” returns the number of entries in the array and if it is greater than zero, then there is at least one matching field. Since we know that there are no other fields in our data model containing “PRM,” we can assume that the first array record has the value we are interested in, and we can print it into a table cell labeled “PRM: “. That’s what the next lines do. First, we close off the code block, since we want to print something out to the screen. Then we make our HTML table cell with the label we want and print the value of the first array record using “ < % = matches[0].value % > ”. The “matches[0]” specifies the first KeyValuePair object in the array, and the “.value” specifies to access the value field (“.key” would give us the field name).

Totals and Discounts

There are two functions in the ShoppingCartBean for getting total information:

- getQuoteTotal
- getQuoteSubtotal

They take no arguments, and you call them in the same way that you call any function in the ShoppingCartBean:

```
< % = bean.getQuoteTotal() % >
```

These functions return a formatted price string. The only concern about using these functions is that you must call the getPartPrice or the getPartSubPrice function in your ShoppingCart JSP page within the hasMoreParts loop. The totals are calculated while the parts are being displayed on the cart page for performance reasons (to avoid a second traversal through the contents of the quote).

There are also two ShoppingCartBean functions related to discounts:

- getLineItemDiscount
- getQuoteDiscount

Neither takes an argument and each returns a formatted number string (for example “5.00”). To have the entered discount percentage apply to the total and subtotal automatically, use these functions to set and retrieve the discount percentage. The percentage entered should be a number between 0 and 100 (decimals are acceptable as well, for example “5.75”). Call the line item discount function to allow a discount at the line item level, and call the quote discount function to allow a discount for the entire quote. These functions may be used together.

Shopping Cart Buttons

The following buttons, cart submit actions, are supported by the server-side shopping cart:

Update	Applies any changes the user has made to the cart contents on the server.
Save	Prompts the user for a name and makes the quote accessible from the quote list.
Clear	Creates a new quote and displays it in place of the current quote.
Submit	Submits the cart to the third-party order management system for checkout processing.
Printable Quote	Displays quote in printable text format.

If you want to display these actions as HTML buttons, you must use the following names for the button elements. Otherwise, you must add a URL parameter with this name to the URL when it is clicked and a non-null value.

SUBMIT ACTION	BUTTON/PARAMETER NAME
Update	UPDATE_QUOTE
Save	Call SaveQuote() function in the onClick handler of the button or link. If you want to bypass the SaveQuote() functionality, use SAVE_NEW_QUOTE to save a new copy of the quote, or UDATE_QUOTE to modify the current quote.
Clear	CLEAR_QUOTE
Submit	Call SubmitCart (inLink) function in the onClick handler.
Printable Quote	Call ViewQuote() function in the onClick handler of the button or link.

Submit the shopping cart to the server whenever any of these actions is selected. The two JavaScript functions, SaveQuote and SubmitCart, submit the cart to the server before returning.

Update Action

Below is an example of using an HTML link for the Update action, dynamically adding the correct parameter name, and submitting the cart form to the server.

```
<a href=""
onClick="document.cart_form.action+='&UPDATE_QUOTE=update’;
document.cart_form.submit(); return false;">
<img border=0 src="transactserver/updateorder_btn.gif"></a>
```

In the onClick event, the UPDATE_QUOTE parameter name is added to the URL already defined for the form action, and is given a non-null value (‘update’). Then JavaScript is used to submit the form.

Clear Action

Below is an example of an HTML link used for the Clear action, which is almost identical in syntax to the Update action above.

```
<a href=""
onClick="document.cart_form.action+='&CLEAR_HEADER=clear';
document.cart_form.submit();">
<img border=0 src="transactserver/clearinfo_btn.gif"></a>
```

In the onClick event, we add the CLEAR_HEADER parameter name to the URL already defined for the form action, and give it a non-null value (‘clear’). Then we use JavaScript to submit the form.

Save Action

Below is an example of using an HTML link around an image for the save action. Since save needs to prompt the user for a quote name, we call the SaveQuote JavaScript function included in the ShoppingCart JSP, which gets the quote name from the user and then submits the cart form using JavaScript. If the quote name were a field on the shopping cart JSP, you could modify the SaveQuote JavaScript function to check for a value in this field and then submit the cart form.

```
<a href="" onClick="SaveQuote(); return false;">
<img src="transactserver/saveorder_btn.gif" border=0></a>
```

Submit Action

Below is an example of using the SubmitCart JavaScript call which submits the contents of the cart to the Order Management System as an XML message over HTTP. The SubmitCart JavaScript call is included in the ShoppingCart JSP.

```
<a href="" target="_new" onClick="SubmitCart(this);">
<IMG border=0 src="transactserver/submitorder_btn.gif"></a>
```

Error Handling

Errors can occur as the ShoppingCart is being processed and they may prevent the user's submit action from completing successfully. If this situation occurs, details need to be communicated to the user as clearly as possible.

onLoad Error Handling

If an error has occurred before the page has finished processing, it can be caught and conveyed to the user as part of the page's onLoad event. The shopping cart recognizes the types of errors listed in this section, which you handle by defining a JavaScript function in ShoppingCart.jsp with the same name and no arguments. These functions are most likely to be implemented to display a JavaScript alert box telling the user more about the error, its possible consequences, and what they should do next. Your error handling function will be called automatically in the onLoad handler by calling the ShoppingCartBean function getCartOnLoad as follows:

```
<body bgcolor=#FFFFFF onLoad="<%= bean.getCartOnLoad() %>">>
```

ConfigChangedError

This is a concurrency error: Another user changed the saved configuration this user is trying to add to the cart. This error should rarely occur. The user should try to reopen the saved configuration to see the changes made by the other user, and add it to the cart again.

The default definition of this error handling function is:

```
function ConfigChangedError() {
   alert("The configuration you wanted to add to your cart\n has
been changed by another user!\nPlease try again!");
}
```

QuoteChangedError

This is a concurrency error. It is thrown when two users are modifying the same saved quote simultaneously. The user who updates the quote first does so successfully. The second user to try to update the quote gets this error and sees the quote as the first user has saved it. The second user may then reapply his or her changes and update again.

The default definition of this error handling function is:

```
function QuoteChangedError() {
  alert("This quote was changed by another user!\nThe changes they
made are shown here.\nPlease make your changes again!");
}
```

NoPermissionToChangeError

If the current user is viewing a quote which has been sent by email, then that quote was created by another user and unless he or she has group or super permission granted to them by Transact, that user cannot modify the quote. If the user attempts to make a change to the quote, it will not be saved and Transact returns this error. Users can create their own copy of the quote by choosing “Save New” on save, and can make any desired changes to their own copy.

The default definition of this error handling function is:

```
function NoPermissionToChangeError() {
  alert("Sorry, this quote was created by another user and you do
not have permission to modify it.  Click Save Quote and choose
Save New to save your own copy of this quote.");
}
```

QuoteDeletedError

If the current quote has been deleted, then the user can no longer modify or view it (it no longer exists). The creator of the quote is the only user with permission to delete it.

The default definition of this error handling function is:

```
function QuoteDeletedError() {
  alert("This quote has been deleted from the database by the user
who created it! It is no longer valid.");
}
```

AddToCartError

If an error occurs while a user is adding to the cart, the configuration the user tried to add will not appear in the cart interface. You should communicate this to the user. Then, the user can try to add the configuration to the cart again.

The default definition of this error handling function is:

```
function AddToCartError() {
  alert("An error occurred on the server while adding to the
cart!\nPlease try again or contact your system administrator!");
}
```

DeleteError

This error is thrown when users try to delete a line item from a quote they did not create. Only the creator of the quote has delete permission. The line item they tried to delete remains on the quote.

The default definition of this error handling function is:

```
function DeleteError() {
  alert("Sorry, but you can't remove items from another user's
cart!\nSave a new copy of this cart if you want to remove line
items.");
}
```

ServerError

This error is a catch-all for all the other unspecified errors that might happen during the initialization of the Shopping Cart.jsp page that are not described above. If this error is thrown, changes that the user made to the cart in their last operation might not be displayed.

The default definition of this error handling function is:

```
function ServerError() {
  alert("An error occurred during the operation you tried to
perform! Please try again or contact your system administrator.");
}
```

finishUp Error Handling

There are many further occasions for an error to occur after the onLoad handler is processed. That is why, after the finishUp function is called, the ShoppingCart.jsp template should check to see if any other error conditions have been raised. There are two functions which the ShoppingCartBean API provides to determine if another error has occurred. They should both be called, and an appropriate JavaScript error handling function should be called to display the error message to the user.

getQuoteDeletedError

If the quote was deleted by another user while the page was being displayed, this ShoppingCartBean API function returns a Boolean value of true. An example of the usage of this function is shown below:

```
<% if (bean.getQuoteDeletedError()) { %>
<script language="JavaScript">
  QuoteDeletedError();
</script>
<% } %>
```

getQuoteErrorCondition

If any other error has occurred during the processing of the page, this ShoppingCartBean API function returns true. An example of this function is:

```
<% if (bean.getQuoteErrorCondition()) {
%>
  <script language="JavaScript">
      ServerError();
   </script>
<% }%>
```

Shopping Cart Template Requirements

The following elements must be included in the ShoppingCart.jsp template.

useBean Inclusion of ShoppingCartBean

The useBean JSP call to include the ShoppingCartBean class is a required part of the ShoppingCart template.

setRequest, setSession

Within or immediately after the useBean call, the setRequest and setSession functions of the ShoppingCartBean API must be called, with arguments of the JSP request and session variables, respectively.

getQuote

After calling setRequest and setSession, the getQuote function must be called. This function retrieves all the information from the database about the quote currently being displayed to the user through the shopping cart interface.

getCartOnLoad

In the body tag of the ShoppingCart.jsp template, the onLoad event handler should get its JavaScript function call string from the ShoppingCartBean API function getCartOnLoad. Other JavaScript calls may be included in the onLoad handler after the getCartOnLoad functions.

SetQuoteID (JavaScript function)

The JavaScript function calls returned from getCartOnLoad always contain a call to the JavaScript function SetQuoteID. This function is provided in the ShoppingCart.jsp template by default and should not be removed. The purpose of this function is to set the current quote ID in the context of the eAdvisor application session.

cart_form (HTML form)

The ShoppingCart.jsp template must contain an HTML form which must be named "cart_form." The HTML form should contain form elements for all editable quote fields.

getQuoteFormAction

The action parameter of the form should get its value from a call to the ShoppingCartBean API function getQuoteFormAction(). The action of a form defines the destination for the form submit. The getQuoteFormAction not only defines the URL for the form submit, but also includes extra URL parameters. The template author may add to these parameters by concatenating a new parameter string to the form action, but the parameters already included should not be removed or modified.

QUOTE_NAME (form variable)

A form variable called QUOTE_NAME must be included in the cart_form. The value of this form element must be supplied by means of a call to the ShoppingCartBean API function getQuoteName(). The form variable may be of type "hidden" to prevent it from displaying on the cart interface, or it may be made visible on the template.

finishUp

After all the quote line items and parts have been iterated through, a call must be made to the ShoppingCartBean API function finishUp(). This function writes any changes the user has made to the database.

Submitting the Cart to an Order Management System

When users want to check out the contents of their shopping cart and create an order, the contents of the shopping cart will need to be transferred to an order management system for processing and fulfillment. The order management system may continue to ask the user for more information (such as credit card or other payment information) to continue the checkout transaction process. There are two principal methods for passing information to an order management system: XML and HTML forms. Both use HTTP as the protocol for transmitting information to the order management system, and both use the Order Management System URL property as the target for the HTTP message.

Default XML Format

Refer to Appendix G, “Additional Code” for the DTD used by the shopping cart to produce XML output. It is nearly identical to the Transact configuration DTD, but contains extra data elements to display quote header information and totals.

Specifying an XSLT Stylesheet

Specify the XSLT stylesheet URL or absolute filepath in the properties editor in the Transact Cart XSLT Stylesheet property. See “XSLT Style Sheet Example” on page 223 for an example of an XSLT stylesheet that converts between the default XML DTD and a cXML DTD.

HTML Form-Based Cart Submission

You can use the ShoppingCartBean to write an HTML form dynamically for submission to a server-side program on checkout. An example of this technique is given in Appendix D. Pass the parameter “quote_id” with the quote ID of the current shopping cart on the URL to the JSP page containing the checkout form. Using standard HTML techniques, the checkout JSP page might be loaded into the window displaying the shopping cart contents, perhaps with a message that the checkout is processing, or it could be loaded into a hidden HTML frame, which then displays to the user any HTML returned from the program that processed the form.

Printable Order (the View-Only Cart)

The view-only cart should contain the same information as the Shopping Cart itself, but all fields are displayed as plain text and as such are not editable by the user. The view-only cart plays a role in two scenarios:

- Users click open from the quote list when they already have an active shopping cart and choose to view the quote instead of use it as their shopping cart.
- Users click a button on the shopping cart to display a formatted view of the cart that they may print.

The view-only cart template is formatted for easy side-by-side comparison with the Shopping Cart. The Java bean used to access view-only cart information is in fact the same as for the Shopping Cart (ShoppingCartBean). The JSP template representing the view-only cart is called ViewQuote.jsp. The structure of the JSP template for the view-only cart is for all purposes identical to the shopping cart template.

The Quote List

The Quote List, similar to the configuration list, is an interface which allows a user to access saved quotes.

QuoteList Functionality

The Quote List is an interface through which a user may access previously saved quotes. From the list interface, users may restore quotes, delete quotes they have created, or share quotes with others through email. The Quote List functionality parallels the Transact Config List. The Quote List displays all quotes which are available to the user, so if users have group or super-user permission in Transact they will see quotes created by other users as well as their own.

QuoteListBean Initialization

The QuoteListBean must be included in the QuoteList.jsp template just as the ShoppingCartBean does in the ShoppingCart.jsp template. There are also the familiar setSession and setRequest functions which must pass the session and request objects into the bean. The getQuoteList function gets all the quote list information from the database just as the getQuote function does in the ShoppingCartBean. But before the getQuoteList function is called, you can call some functions to specify the sorting of the list. This is exactly the same as the ConfigList. Following is the useBean statement from the QuoteList template and an explanation of each call:

```
<jsp:useBean id="quoteList"
   scope="page"
   class="com.siebel.isscda.wl.transact.QuoteListBean">
<%
    quoteList.setSession(session);
    quoteList.setRequest(request);
    quoteList.setSortField("DateCreated");
    quoteList.setSortOrder("desc");

    quoteList.getQuoteList();
%>

</jsp:useBean>
```

The useBean statement itself is exactly like the one for ShoppingCartBean. It gives a name in the “id” parameter, which we use to refer to the QuoteListBean in the QuoteList template from now on. We used the name “quoteList” in the code snippet above. The class parameter gives the fully qualified name of the QuoteListBean class so that the JSP compiler can find the code to use for the QuoteListBean API calls.

Inside the useBean tags, we defined a code block. It is useful to have a code block in the useBean tags to put initialization statements related to the bean. This way, we know that these statements will be executed before any calls into the bean may be made.

The first statements inside the code block are calls to setSession and setRequest. The setSession call is passed the JSP session object, and the setRequest call is passed the JSP request object. You do not need to do anything to have access to the session and request objects. They are automatically made available to you in every JSP page.

The next two statements are also related in their purpose. They set the sort by field and the sort order for the quotes in the quote list. If the sort order is set to “asc,” then quotes will be listed in order from least to greatest according to the value in the field specified in the setSortField call. If the sort order is set to “desc,” then the quotes are ordered from greatest to least according to the value in the sort field. The sort field may be specified as one of the following:

- **Name** is the quote name
- **DateCreated** is the date the quote was created
- **AccountID** is the account assigned to the quote
- **ID** is the numeric unique identifier of the quote

The last call inside the code block must be made after the four calls described above. The call is getQuoteList and takes no arguments. This call retrieves all the quote list information for the current user from the database.

Iterating Through the User’s Quote List

The code for iterating through the list of the user’s saved quotes is similar to that used to work through the Transact ConfigList.jsp template. Each quote that is available on the user’s quote list is displayed by iterating through the entire list with a while loop. Inside the while loop, you can access information pertaining to the current quote in the list and display it to the user.

The form of the while loop is:

```
<% while (quoteList.nextQuote()) { %>
<tr>
<td>
<font face=Verdana size=-1><%= quoteList.getID() %></font>
</td>
…//display more quote information…
<%}%>
```

The general approach here, similar to the method for displaying shopping cart line items, is for the while loop to contain the HTML needed for formatting and displaying the desired information pertaining to a single line item. You would put headers for each field displayed on the list above the start of the loop in the Quote List template.

Error Handling

The server may return one of two errors during quote list processing. You can test for the occurrence of errors by using JSP at the beginning of the QuoteList template, and then handle them by displaying an alert message to the user with JavaScript. The alert message should give the user an idea of why the error occurred and the possible consequences of the error condition. The default code in the QuoteList.jsp template for handling these errors is:

```
<% if (quoteList.errorDeletedAnotherUserQuote()) { %>
   alert("Sorry, you cannot delete a quote created by another
user!");
<% } else if (quoteList.errorOccurred()) { %>
   alert("An error occurred on the server while retrieving your
list!\n Please try again or contact your system administrator.");
<% } %>
```

You can test the first error condition with the QuoteList API function errorDeletedAnotherUserQuote, a function which returns true or false. Transact only allows quotes to be deleted by their creator to simplify concurrent access. The call to quoteList.errorDeletedAnotherUserQuote() returns true if the user just attempted a delete from the quote list on a quote that the user does not own. Transact returns a message to the user that explains why the operation is disallowed.

The second error condition is more general and can be caused by any trappable but nonrecoverable error on the server side. A database error that prevents successful deletion or selection from one of the quote tables is the most frequent cause. The API function you use to detect that the QuoteList operation has thrown an error is the errorOccurred() function. The default handling for this error is to alert the user that an error occurred on the server and to instruct the user to try again.

What You Can Display in the Quote List

You can display the following in the Quote List:

Quote ID	Unique identifier of the quote. Retrieved with QuoteList API function getID().
Quote Name	Name provided by the user when quote was saved. Retrieved using the QuoteList API function getName().
Account ID	Account ID associated of saved quote. Retrieved using the QuoteList API function getAccountID().
Index	Quote's numbered position in list. Retrieved using the QuoteList API function getIndex().
Date Created	Date when quote was created. Returned in locale-specific format using the QuoteList API function getDateCreated().
Email URL	URL string that displays the EmailQuote.jsp template used to email the quote to another user. Retrieved using the QuoteList API function getEmailURL().
Delete URL	URL string that deletes quote from database. Retrieved with QuoteList API function getDeleteURL(). Use the function doDelete (included in the QuoteList JSP template) to prevent user from deleting the current quote. For example: `<a href="" onClick="doDelete('<%= quoteList.getDeleteURL() %>'); return false;">Delete</a>`
Restore Quote	Restores a quote. You must call the JavaScript function top.siebel.RestoreQuote with the quote ID as an argument. This function prompts users on whether to use the selected quote as their shopping cart or see a view-only version of the quote. Once quote becomes the current shopping cart, any previous shopping cart contents are lost (unless previously saved).

EmailQuote.jsp template

The EmailQuote.jsp template is shown from the quote list when the user clicks on the link to email a quote. A pop-up window then appears, containing a form that provides fields used to create an email message. When the form is submitted, the email message is sent out through the SMTP (email) server specified in the Transact Server Administration tool. The user must provide the recipient's email address, subject heading, and message body. The message body is prefilled with the URL that will allow recipients to restore the quote when they click on it from their email reader.

Transact API for Siebel eAdvisor A

Use the Transact API functions to access the Transact functionality from anywhere in your Siebel application.

You can use pre-event call out/override points (COP_<function name>) just before an event to override the default behavior. See Appendix B, “Transact Server Callout/Override Points,” for more information.

AddToCart

AddToCart function adds the current configuration to the shopping cart.

Syntax

```
OL.AddToCart(target)
```

Arguments

target	To use a frame, supply the name of the frame in which you want the cart to appear: OL.AddToCart('framename') If you want the cart to appear in its own window, use: OL.AddToCart('_new')

Usage

This function is commonly called from the onClick event handler of a link, but it can also be referred to from an image map or called from another user-defined JavaScript function. See *Siebel Interactive Designer Administration Guide* for instructions on adding link objects to your Siebel application.

For information on posting a form from Add to Cart, see "Posting a Form from Add to Cart" on page 80.

Example

```
<INPUT type="button" value="Add To Cart"  name=add
onClick="OL.AddToCart('_new');">
```

ConfigList

ConfigList function displays the configuration list.

Syntax

```
OL.ConfigList(target)
```

Arguments

target	To use a frame, supply the name of the frame in which you want the configuration list to appear: OL.ConfigList('framename') If you want the configuration list to appear in its own window, use: OL.ConfigList('_new')

Usage

This function is commonly called from the onClick event handler of a link, but it can also be referred to from an image map or called from another user-defined JavaScript function. See *Siebel Interactive Designer Administration Guide* for instructions on adding link objects to your Siebel application.

Example

```
<a href="" onClick="OL.ConfigList('ol_ui.mainArea');return false;"
>List Configs</a><br>
```

RestoreConfig

You can create a link on the Configuration List page to link back to and open saved configurations. Call RestoreConfig on an onClick event, sending the HTML link as an argument.

Example

```
OL.restoreConfig(this)
```

SaveConfig

SaveConfig function saves a configuration to the configuration list. SaveConfig automatically calls GetSaveConfig, displaying a Save dialog box in which the user enters a name for the configuration.

Syntax

```
OL.SaveConfig()
```

Usage

This function is commonly called from the onClick event handler of a link, but it can also be referred to from an image map or called from another user-defined JavaScript function. See *Siebel Interactive Designer Administration Guide* for instructions on adding a link object in your Siebel application.

ShowCart

ShowCart function displays the shopping cart.

Syntax

```
OL.ShowCart(target)
```

Arguments

target	To use a frame, supply the name of the frame in which you want the cart to appear: OL.showCart('framename') If you want the cart to appear in its own window, use: OL.showCart('_new')

Usage

This function is commonly called from the onClick event handler of a link, but it can also be referred to from an image map or called from another user-defined JavaScript function. See *Siebel Interactive Designer Administration Guide* for instructions on adding a link object in your Siebel application.

Example

```
<a href="" onClick="OL.ShowCart('_new'); return false;">Show Cart</
a>
```

Error Messages

The following sections describe how to display error messages.

COP.InvalidI temAdded

The COP.InvalidItemAdded is an Order API function you can define to display an error message when a user attempts to order an invalid configuration.

Syntax `COP.InvalidItemAdded()`

Usage The COP.InvalidItemAdded() callout point is called if you try to order an invalid configuration. This function is not called by default; you must manually define it.

The Order API is mainly used for internal purposes. It shields Transact from the Siebel application engine's internal representation of the configuration, and removes any data that is extraneous to the purpose of restoring the configuration state. COP.InvalidItemAdded is the one Order API function you will need to define.

ServerError

You can define ServerError with custom code. If an error occurs on the server side, instead of displaying the default server messages, your client-side function will be called and display the error you defined to the user. This is a static message, so if you choose to implement this function, you can provide only one message for all server errors.

Syntax `OL.ServerError`

Transact NotAccessible

In the app_config.js file, the message config variable, TRANSACT_NOT_ACTIVE_MSSG, sets an error message to be displayed when Transact is not turned on. When a user attempts to access Transact functionality and Transact has not been turned on, an alert box displays the message:

```
Sorry, the action you have requested is currently unavailable.
Please contact your system administrator.
```

Transact Server Callout/Override Points B

Use the Transact Server callout/override points to override the default Transact Server functionality and implement your own. Use the callout/override point before a function to override the default behavior.

See *Siebel Interactive Designer Administration Guide* for more information on callout/override points.

COP_AppDataVersionCheck

Define this function to prevent out-of-date applications from opening. See "Version Checking" on page 103 for more information.

Syntax

```
COP_AppDataVersionCheck(obj)
```

Example

```
{
   var retVal = confirm("The item you are opening was created in
version \n" + obj['appDataVersion'] + "\n but your current
application is version \n" + appVersion['currAppDataVersion'] + ".
Do you wish to continue?");
   return retVal;
}
```

COP_Before AddToCart

This callout point could be used to ask for confirmation before adding an item to the cart. If the function returns *true*, the Add To Cart operation will continue, but if it returns *false*, the item will not be added to the cart.

Example

```
function COP_BeforeAddToCart(dataobj;target) {
  return (confirm("Are you sure you want to add this item to your
cart?"));
}
```

COP_Before RestoreConfig

This callout point could be used to ask for confirmation before restoring a configuration. If the COP_BeforeRestoreConfig function returns *true*, the configuration will be restored. If it returns *false*, the restore operation will be canceled.

Example

```
function COP_BeforeRestoreConfig(data_state) {
   return confirm('Do you really want to open this configuration?');
}
```

COP_Before SaveConfig

Define COP_BeforeSaveConfig to allow users to confirm that they want to save the configuration. Returning *false* cancels the save, and returning *true* allows the save to continue.

Example

```
function COP_BeforeSaveConfig(data_state) {
   return (confirm('Are you sure you want to save this 
configuration?'));
}
```

COP_PagesetVersionCheck

Define this function to prevent out-of-date page sets from opening. See “Version Checking” on page 103 for more information.

Syntax

```
COP_AppDataVersionCheck(obj)
```

Example

```
{
   var retVal = confirm("The item you are opening was created in 
version \n" + obj['linkBackVersion'] + "\n but your current 
application is version \n" + obj['pagesetVersion'] + ".  Do you wish 
to continue?");
   return retVal;
}
```

OR_ConfigSavedSuccess

Define this function to override the default functionality that displays an alert box with the message:

```
Your configuration has been saved!
```

You may choose to display a custom message or just remove the alert.

Syntax `OL.OR_ConfigSavedSuccess()`

Usage If this function is defined, it is called following the successful save of a configuration.

Example

```
function OR_ConfigSavedSuccess(){
alert(“The configuration was saved successfully”);
}
```

OR_GetSaveConfigName

OR_GetSaveConfigName function overrides the default dialog box for saving a configuration.

Syntax `OL.OR_GetSaveConfigName()`

Usage OR_GetSaveConfigName overrides the default functionality provided by GetSaveConfigName. GetSaveConfigName is automatically called from SaveConfig when you save a configuration. A dialog box opens, asking the user to name the configuration, and gives you the option to update the configuration if it already exists, saving it as a new configuration, or canceling the operation.

Using OR_GetSaveConfigName, you can change the prompt text or allow the user to enter a configuration name by another method. For example, you could create a text field into which the user can enter a configuration name directly on the Inputs page.

To use this function, write code that obtains the name of the configuration from the user, and then call GotSaveConfigName() using the configuration name as an argument. Like GetSaveConfigName, OR_GetSaveConfigName is automatically called from SaveConfig.

Example

```
function OR_GetSaveConfigName(data_state) {

   if (ol_ui.mainArea.document.forms[0].config_name.value == "") {
    alert('Please enter a name for the configuration to save!');
   OL.GotSaveConfigName(null);
   }
   else {
   OL.GotSaveConfigName(ol_ui.mainArea.document.forms[0]
.config_name.value);
   }
}
```

GotSaveConfig Name

By sending a null name to the GotSaveConfigName, you can cancel the save.

Syntax

```
OL.GotSaveConfigName(string name)
```

Arguments

string name	OL.GotSaveConfigName('server config')

Usage

Call GotSaveConfigName after obtaining the configuration name in OR_GetSaveConfigName.

OR_Transact NotActive

Define this function to override the default functionality that displays an alert box when Transact is turned off. You may choose to display a custom page or just remove the alert.

Example

```
function OR_TransactNotActive(data_state) {
   alert('Sorry, this feature is not available!');
}
```

ConfigList API C

The ConfigList API is created in the ConfigList bean. After importing the ConfigList bean from the ConfigList JSP page (using the useBean tag), you can call the functions in this Appendix from the ConfigList JSP page to customize the configuration list.

See Chapter 3, “Working with Configurations,” for information on importing the ConfigList bean functions and modifying the JSP page to customize the configuration list.

In JSP, the tags < % = and % > are used to print the enclosed text on the page. The tags < % and % > are used for code that you do not want displayed after evaluation.

anyConfig

anyConfig function tests whether there are more configurations to display on the list.

Syntax

```
<bean name>.anyConfig()
```

Usage

The anyConfig function returns false if the list is empty.

Example

```
<% while (listBean.anyConfig())
{ %>
//code for going through list
<% } %>
```

createList

createList function displays the list.

Syntax

```
<bean name>.createList(HTTPSession session, HttpServletRequest
request)
```

Arguments

HTTPSession session	Built-in JSP variable to refer to the session object
HttpServletRequest request	Built-in JSP variable to refer to the request object.

Usage

You must call this function before the list can be displayed. The sort order should be set before the list is created.

Example

```
configList.createList(session,request);
```

getAccountId

getAccountId function returns the account ID for current configuration in the list.

Syntax

```
<bean name>.getAccountId()
```

Usage

Call this function to display the account identifier associated with the current saved configuration on the list.

Example

```
<td><%= listBean.getAccountId() %></td>
```

getDateCreated

getDateCreated function gets the date when the current configuration in the list was created.

Syntax `<bean name>.getDateCreated()`

Usage Call this function to display the creation date associated with the current saved configuration on the list.

Example

```
<td><%= listBean.getDateCreated() %></td>
```

getDeleteURL

getDeleteURL function provides a Delete link from the configuration list.

Syntax

```
<bean name>.getDeleteURL()
```

Usage

Call the getDeleteURL function from a button or link to allow the user to delete the configuration from the database. The list reloads to display an updated list, excluding the deleted configuration.

Example

```
<a href="<%= configList.getDeleteURL() %>">Delete</a>
```

getDescription

getDescription function returns a FAMILY DESCRIPTION for the page set for the current configuration in the list.

Syntax

```
<bean name>.getDescription()
```

Usage Call this function to display the description associated with the current saved configuration on the list.

Example

```
<td><%= listBean.getDescription() %></td>
```

getEmailURL

getEmail URL function returns the URL to display the EmailConfig.jsp page to the user. The EmailConfig.jsp page allows users to email the saved configuration to any recipient they designate. The link that represents the getEmailURL call should target a new window in order to properly display the EmailConfig.jsp page.

Before using this functionality, the application designer must set up the EmailConfig.jsp page and set the SMTP Server Name in the property editor.

Syntax

```
getEmailURL()
```

Example

```
<%= bean.getEmailURL() %>
```

getIndex

getIndex function adds an index item to the configuration list.

Syntax

```
<bean name>.getIndex()
```

Usage

The getIndex function returns the index item (adding one to the previous index number) for the current configuration in the list. The first configuration in the list will be 1, the second 2, and so on.

Example

```
<td> <%= listBean.getIndex() %> </td>
```

getName

getName function returns the name the user provided when the configuration was saved for the current configuration in the list.

Syntax

```
<bean name>.getName()
```

Example

```
<td> <%= configList.getName() %> </td>
```

getRestoreURL

getRestoreURL function returns URL to restore the current configuration in the list.

Syntax

```
<bean name>.getRestoreURL()
```

Usage

The getRestoreURL function obtains the URL for the configuration and calls the Order API Linkback function to restore the configuration to the Configuration list. You will need to call OL.RestoreConfig from an OnClick event.

Example

```
<a href =”<%=configList.getRestoreURL() %>
target=”_new><%=ConfigList.getUniqueId() %></a>
```

getSortField

getSortField function sorts the configuration list by specific fields.

Syntax

```
<bean name>.getSortField()
```

Usage

getSortField returns the following sorted fields: Name, DateCreated, DateSubmitted, Description, and Account ID.

The sort order is ascending or descending.

Example

```
<td><%= listBean.getSortField() %></td>
```

getSortOrder

getSortOrder function determines the sort order of the Config list.

Syntax `<bean name>.getSortOrder()`

Usage getSortOrder returns Asc (Ascending) or Desc (Descending) for the sort order of the list. getSortOrder is a sibling function to getSortField.

Example

```
<td><%= listBean.getSortOrder() %></td>
```

getUniqueId

getUniqueId function returns the unique ID for the current configuration in the list.

Syntax

```
<bean name>.getUniqueId()
```

Usage

Use getUniqueId to return an internal ID for the database record of the configuration. If a problem arises, you can use this ID to locate the record for the configuration in your database.

Example

```
<td><%= listBean.getUniqueId() %></td>
```

getUserId

getUserId function returns the user identification (user ID, name, login, and so on) for the current configuration in the list.

Syntax

```
<bean name>.getUserId()
```

Example

```
<td><%= listBean.getUserId() %></td>
```

nextConfig

nextConfig function in the while loop of a ConfigList JSP page moves to the next configuration before calling any get methods to return information about that configuration.

Syntax

```
<bean name>.nextConfig()
```

Usage nextConfig returns false when there are no more configurations in the list.

Example

```
<% while (configList.nextConfig()) {%>...
```

setSortField

setSortField function sets the sorting of the Config list before createList is called.

Syntax

```
<bean name>.setSortField(String sort_field)
sort_field must be one of these values, "DateCreated",
"DateSubmitted", "Name", and "ID".
```

Usage In the following example, the list of saved configurations will appear sorted by date, used in conjunction with setSortOrder.

Example

```
configList.setSortField(“DateCreated”);
```

setSortOrder

setSortOrder function specifies whether the list is sorted in ascending or descending order according to the value in the sort field.

Syntax

```
<bean name>.setSortOrder(String sort_order)
```

Arguments

Asc	Sorts the list in ascending order.
Desc	Sorts the list in descending order.

Usage

setSortOrder orders the configuration list in ascending or descending order (for letters or numbers) for the field you have chosen to sort by, used in conjunction with setSortField. This function must be called before createList.

Example

```
configList.setSortOrder(“Desc”);
```

Email Bean API D

The Email Bean API is created in the Email bean. After importing the Email bean from the Email JSP page (using the useBean tag), you can call the following functions from the Email JSP page to customize the Transact Server email interface.

See Chapter 3, “Working with Configurations,” for information on accessing email functionality from the Configuration list.

In JSP, the tags < % = and % > are used to print the enclosed text on the page. The tags < % and % > are used for code that you do not want printed.

getAction

getAction function returns the action used to submit your email form.

Syntax

```
public String getAction(String pagename)
```

Arguments

String pagename	The name of your Email JSP page.

Usage

The getAction function returns the action for the email form. It takes the name of your Email JSP page as an argument.

Example

```
<form name="Email" action="<%=Email.getAction("EmailConfig.jsp")
%>" method="post">
```

getErrorMessage

getErrorMessage function returns the error message posted when an email has failed to be sent. However, if the recipient mail server is down or the recipient email address is incorrect, the API cannot send the error messages back to Transact server.

Syntax

```
public String getErrorMessage()
```

Usage

If the email failed to be sent (for example, sendMail returns false), call the getErrorMessage function to get a description of the error that occurred.

Example

```
<script language=JavaScript>
alert("<%=Email.getErrorMessage() %>");
</script>
```

getMailSent

getMailSent function determines whether an email was sent successfully. However, if the recipient mail server is down or the recipient email address is incorrect, the API cannot send the error messages back to Transact server.

Syntax

```
public boolean getMailSent()
```

Usage

The getMailSent function returns a true/false Boolean value indicating whether or not the email was sent successfully.

Example

```
<% if (!Email.getMailSent()) { %>
<script language = “JavaScript”>
alert (‘Your mail was not sent!’);
</script>

<% }%>
```

getRestoreConfigURL

getRestoreConfigURL function restores the configuration.

Syntax

```
public String getRestoreConfigURL()
```

Usage

The getRestoreConfigURL returns the URL which will restore the configuration. This URL must be included in the “message” parameter of the sendMail function in order to successfully email the saved configuration.

Example

```
<p><textarea name="Message" rows="4" cols="80">
<%= Email.getRestoreConfigURL()) %></textarea></p>
```

sendMail

sendMail function sends the email.

Syntax

```
public boolean sendMail(String to, String subject, String message)
```

Arguments

String to	**The recipient of the email**
String subject	The subject for the email.
String message	The email message.

Usage

The sendMail function sends the email message given in the “message” parameter to the recipient specified in the “to” parameter with the subject in the “subject” parameter. This function returns true if the message is sent successfully or false if an error is encountered during send.

Example

```
<% if (request.getParameter("To") != null) {
   boolean mail_sent = Email.sendMail(request.getParameter("To"),
request.getParameter("Subject"), request.getParameter("Message")));
}
%>
```

setRequest

setRequest function sets the request variable of your JSP page.

Syntax

```
public void setRequest(HttpServletRequest request)
```

Arguments

HttpServletReq uest request	Built-in JSP variable to refer to the request object.

Usage

Call the setRequest function with the built-in request variable of your JSP page. The setRequest function must be called before calling the sendMail function.

Example

```
<% Email.setRequest(request); %>
```

setSession

setSession function sets the variable of your JSP page.

Syntax

```
public void setSession(HttpSession session)
```

Arguments

HTTPSession session	Built-in JSP variable to refer to the session object.

Usage

Call the setSession function with the built-in session variable of your JSP page. This function must be called before calling sendMail.

Example

```
<% Email.setSession(session); %>
```

ConfigAccess Bean API E

The ConfigAccess Bean API is created in the ConfigAccess bean. After importing the ConfigAccess bean from the Post Form JSP page (using the useBean tag), you can call the following functions from the Post Form JSP page to customize the content of the form post to the Order Management System.

See "Posting a Form from Add to Cart" on page 80 to see a sample form created using the ConfigAccess Bean API.

In JSP, the tags < % = and % > are used to print the enclosed text on the page. The tags < % and % > are used for code that you do not want printed.

NOTE: Before accessing the ConfigAccessBean API in your JSP page, you must include the bean in your JSP page using a useBean tag.

Example

```
<!-- BEA WebLogic -->
<jsp:useBean id="bean" scope="page"
class="com.siebel.isscda.wl.transact.ConfigAccessBean">
<!-- BEA WebLogic -->

<!-- IBM WebSphere -->
<jsp:useBean id="bean" scope="page"
class="com.siebel.isscda.ws.transact.ConfigAccessBean">
<!-- IBM WebSphere -->
</jsp:useBean>
```

Functions

Function	Description	Syntax	Example
finishUp	Call this function last in the page.	public void finishUp()	< % bean.finishUp() % >
getBillingCity	Call this function to get the city associated with the account billing address.	public String getBillingCity()	< % = bean.getBillingCity() % >
getBillingName	Call this function to get the name associated with the account billing address.	public String getBillingName()	< % = bean.getBillingName() % >
getBillingCountry	Call this function to get the country associated with the account billing address.	public String getBillingCountry	< % = bean.getBillingCountry() % >
getBillingState	Call this function to get the state associated with the account billing address.	public String getBillingState()	< % = bean.getBillingState() % >
getBillingStreet	Call this function to get the street address associated with the account billing address.	public String getBillingStreet()	< % = bean.getBillingStreet() % >
getBillingZip	Call this function to get the ZIP Code associated with the account billing address.	public String getBillingZip()	< % = bean.getBillingZip() % >
getConfig	This function must be called to get the desired configuration. It must be called before any of the part accessor methods (for example, getPartID or nextPart).	public void getConfig()	< % bean.getConfig(); % >

Function	Description	Syntax	Example

getMatchingPartFields	This function returns an array of KeyValuePair objects whose keys are the field names and values of the part's config data, where the field name (key) contains or equals the string argument field_match. If you called getMatchingPartFields with "PRICE" as the field_match argument, you would get back, for example, "PETPRICE," "MPRICE," "DISCPRICE," and "PRICE" if those fields were all in the config data for the part.
	
	The KeyValuePair object is an object containing two string fields: key and value. If the key field of the KeyValuePair object is the string "PRICE," the value will be the price.
		public KeyValuePair[] getMatchingPartFields (String field_match)	<% KeyValuePair[] array = bean.getMatchingPartFields("PRICE");
			
			for (int i=0; i<array.length; i++) { %>
			
			field <%=array[i].key %> has value <%= array[i].value %>
			
			<% } %>

getShippingCity	Call this function to get the city associated with the account shipping address.	public String getShippingCity()	<%= bean.getShippingCity() %>
getShippingCountry	Call this function to get the country associated with the account shipping address.	public String getShippingCountry()	<%= bean.getShippingCountry() %>
getShippingName	Call this function to get the name associated with the account shipping address.	public String getShippingName()	<%= bean.getShippingName() %>
getShippingState	Call this function to get the state associated with the account shipping address.	public String getShippingState()	<%= bean.getShippingState() %>
getShippingStreet	Call this function to get the street address associated with the account shipping address.	public String getShippingStreet()	<%= bean.getShippingStreet() %>
getShippingZip	Call this function to get the ZIP Code associated with the account shipping address.	public String getShippingZip()	<%= bean.getShippingZip() %>
getPartConfigData	This function returns the ConfigData for the configuration as a delimited string. The delimiter used is "~|".	public String getPartConfigData()	<%= bean.getPartConfigData() %>
getPartDescr	This function returns a String description of the part.	public String getPartDescr()	<%= bean.getPartDescr() %>

Function	Description	Syntax	Example
getPartExtPrice	This function returns the extended price for the part (price*qty).	public String getPartExtPrice()	< % = bean.getPartExtPrice() % >
getPartExtSubPrice	This function returns the extended subprice for the part (subprice*qty). Syntax public String getPartExtSubPrice() Example < % = bean.getPartExtSubPrice() % >		
getPartField	This function returns the value of the ConfigData field. ConfigData includes all inputs and outputs. Inputs include all widget selections and all related feature table data. Inputs also include all untabled widgets, which are usually text input boxes that are not associated with feature tables. Outputs are all information from the configuration table. Extra information from 0 columns like PRICE or PARTNUM is also included.	public String getPartField(String field)	< % = bean.getPartField("DESC") % >
getPartItemized	This function returns the value of the ITEM_ITEMIZED variable. See *Siebel Interactive Designer Administration Guide* for more information on the ITEM_ITEMIZED variable.	public String getPartItemized()	< % = bean.getPartItemized() % >
getPartParentID	This function returns the unique identifier of this part's parent, if a parent exists.	public int getPartParentID()	< % = bean.getPartParentID() % >
getPartID	This function returns the unique ID of the part.	public int getPartID()	< % = bean.getPartID() % >
getPartIndex	This function returns the number of the part in the list, such that the first part has an index value of 1.	public int getPartIndex()	< % = bean.getPartIndex() % >
getPartLevel	The getPartLevel function returns a number indicating the number of ancestors (parents) this part has in the configuration. The list of parts is traversed from parent to child. A part with no parent will have a level of 0, one parent a level of 1, a parent with a parent a level of 2, and so on.	public int getPartLevel()	< % if (bean.getPartLevel() > 0) { % > < p >

Function	Description	Syntax	Example
getPartNum	This function returns the part's part number as a string.	public String getPartNum()	< % = bean.getPartNum() % >
getPartPrice	This function returns the part's price.	public String getPartPrice()	< % = bean.getPartPrice() % >
getPartQty	This function returns the quantity of the part in the configuration.	public String getPartQty()	< % = bean.getPartQty() % >
getPartSubPrice	This function returns the Part “subprice.” The subprice is where the part's price is the sum of its children's prices.	public String getPartSubPrice()	< % = bean.getPartSubPrice() % >
hasMoreParts	Call this function to determine whether this configuration's list of parts contains more parts to show. This function returns true if there are more parts left in the list. It is used in conjunction with nextPart().	public boolean hasMoreParts()	< % while (bean.hasMoreParts()) { bean.nextPart(); } % > < % } % >
nextPart	Call this function to iterate to the next part in the configuration's list of parts. Used in conjunction with hasMoreParts().	public void nextPart()	< % while (bean.hasMoreParts()) { bean.nextPart(); } % >
setRequest	Call the setRequest function with the built-in request variable of your JSP page. This function must be called before getConfig.	public void setRequest (HttpServletRequest request)	< % bean.setRequest(request); % >
setSession	Call the setSession function with the built-in session variable of your JSP page. This function must be called before getConfig.	public void setSession (HttpSession session)	< % bean.setSession(session); % >

ShoppingCartBean API F

The ShoppingCartBean API is created in the ShoppingCartBean bean. After importing the ShoppingCartBean bean from the ShoppingCart JSP page (using the useBean tag), you can call the functions in this chapter from the ShoppingCart JSP page to customize the ShoppingCart view.

In JSP, the tags < % = and % > are used to print the enclosed text on the page. The tags < % and % > are used for code that you do not want printed.

NOTE: Before accessing the ShoppingCartBean API in your JSP page, you must include the bean in your JSP page using a useBean tag.

Example

```
<jsp:useBean id="bean" scope="page"
class="com.Siebel.isscda.wl.ShoppingCartBean">
</jsp:useBean>
```

General Functions

Function	Description	Syntax	Example
getCartOnLoad	Call this function in the onLoad handler of the BODY tag in the ShoppingCart template.	public String getCartOnLoad()	<body bgcolor=#FFFFFF onLoad="<%= bean.getCartOnLoad() %>">
getQuote	Retrieve all the information from the database about the quote currently being displayed to the user through the shopping cart interface.	public void getQuote()	<%= bean.getQuote() %>
setSession	Pass the JSP session object to the bean.	public void setSession(HttpSession session)	<%= bean.setSession(session) %>
setRequest	Pass the JSP request object to the bean.	public void setRequest(HttpServlet Request request)	<%= bean.setRequest(request) %>
finishUp	After all the quote line items and parts have been iterated through, a call must be made to the ShoppingCartBean API function finishUp(). This function writes any changes the user has made to the database.	public void finishUp()	<%= bean. finishUp () %>
getQuoteErrorCondition	Returns true if an error has happened during the display or update of the shopping cart information.	public boolean getQuoteErrorCondition()	<% if (bean.getQuoteErrorCondition()) { %> <script language="JavaScript"> ServerError(); </script> <% }%>

Function	Description	Syntax	Example
getQuoteChangeError	Returns true if the cart could not be updated because of a concurrency error (another user made changes to the quote before update was clicked). The current quote information is displayed in the shopping cart, including the other user's changes. Further changes may be reapplied and update may be clicked again.	public boolean getQuoteChangeError()	<% if (bean.getQuoteChangeError()) { %> <script language="JavaScript"> ... </script> <% } %>
getQuoteDeletedError	Returns true if the cart is deleted by another user (the creator is the only user allowed to delete the quote).	public boolean getQuoteDeletedError()	<% if (bean.getQuoteDeletedError()) { %> <script language="JavaScript"> QuoteDeletedError(); </script> <% } %>
getQuoteFormAction	The action parameter of the cart_form HTML form in the ShoppingCart.jsp template should get its value from a call to the getQuoteFormAction() method.	public String getQuoteFormAction()	<form name="cart_form" action="<%= bean.getQuoteFormAction() %>" method=POST>

Quote Header Functions

Function	Description	Syntax	Example
getQuoteID	Returns the unique numerical identifier for the quote.	public int getQuoteID()	<td> <font face="VERDANA" size= -2> <b> Quote ID: </b> </ font> </td> <td> <font face="VERDANA" size= -2> <%= bean.getQuoteID() %> </font> </td>
getQuoteUser	Returns the user_id of the user who created the quote.	public String getQuoteUser()	<td> <font face="VERDANA" size= -2> <b> Created By: </b> </font> </td> <td> <font face="VERDANA" size= -2> <%= bean.getQuoteUser() %> </font> </td>
getQuoteDateCreated	Returns the date the quote was created in locale-specific format.	public String getQuoteDateCreated()	<td> <font face="VERDANA" size= -2> <b> Date Created: </b> </ font> </td> <td> <font face="VERDANA" size= -2> <%= bean.getQuoteDateCreated() %> </ font> </td>
getQuoteDateModified	Returns the date the quote was last modified in locale-specific format.	public String getQuoteDateModified()	<td> <font face="VERDANA" size= -2> <b> Date Created: </b> </ font> </td> <td> <font face="VERDANA" size= -2> <%= bean.getQuoteDateCreated() %> </ font> </td>
getQuoteDateSubmitted	Returns the date the quote was submitted to the OMS in locale-specific format.	getQuoteDateSubmitted	<td> <font face="VERDANA" size= -2> <b> Date Created: </b> </ font> </td> <td> <font face="VERDANA" size= -2> <%= bean.getQuoteDateSubmitted() %> </font> </td>

Function	Description	Syntax	Example
getQuoteName()	Returns the name provided by the user when the quote was saved. This always needs to be included as an input on the form, even if it is not visible, so that previous values are not lost, and when a user does provide a name there is a place to put it in the form. The field name must be QUOTE_NAME.	public String getQuoteName()	<input type="hidden" name="QUOTE_NAME" value="<% = bean.getQuoteName() %>">
getQuoteAccountID	Returns the account ID associated with this quote. Should be the same as the account of the user who created the quote.	public String getQuoteAccountID()	<td><font face="VERDANA" size= -2><b>Account ID:</b></font></td><td><font face="VERDANA" size= -2><% = bean.getQuoteAccountID() %></font></td>
getTotalPackages()	Returns the total number of line items in this quote.	public int getTotalPackages()	<% bean.getTotalPackages() %>
getQuoteHeaderPrefix	Returns the prefix string used by Transact ("HEADER_") to identify fields that should be saved as custom header information for the quote. Put your own name for the field after the prefix in the name property of the form element.	public String getQuoteHeaderPrefix()	<td><font face="VERDANA" size= -2><b>Cost Center:</b></font></td><td><input type="text" name="<% =bean.getQuoteHeader( )%>CostCenter"><% = bean.getQuoteHeader("CostCenter") %></font></td>
getQuoteHeader	Returns the value for the header field specified by the name argument. The name should be the name you specified to follow the header prefix. If there is no value for the field in this quote yet, it will show up as empty (returns "").	public String getQuoteHeader(String name)	<td><font face="VERDANA" size= -2><b>Cost Center:</b></font></td><td><input type="text" name="<% =bean.getQuoteHeader( )%>CostCenter"><% = bean.getQuoteHeader("CostCenter") %></font></td>

Function	Description	Syntax	Example
getBillingName	Returns the billing name for the account from LDAP.	public String getBillingName()	< % bean.getBillingName() % >
getBillingStreet	Returns the billing street address for the account from LDAP.	public String getBillingStreet ()	< % bean.getBillingStreet() % >
getBillingCity	Returns the billing city for the account from LDAP.	public String getBillingCity()	< % bean. getBillingCity () % >
getBillingState	Returns the billing state for the account from LDAP.	public String getBillingState()	< % bean. getBillingState () % >
getBillingZip	Returns the billing ZIP Code for the account from LDAP.	public String getBillingZip()	< % bean. getBillingZip() % >
getBillingCountry	Returns the billing country for the account from LDAP.	public String getBillingCountry()	< % bean. getBillingCountry () % >
getShippingName	Returns the shipping name for the account from LDAP.	public String getShippingName()	< % bean.getShippingName() % >
getShippingStreet	Returns the shipping street address for the account from LDAP.	public String getShippingStreet ()	< % bean.getShippingStreet() % >
getShippingCity	Returns the shipping city for the account from LDAP.	public String getShippingCity()	< % bean. getShippingCity () % >
getShippingState	Returns the shipping state for the account from LDAP.	public String getShippingState()	< % bean. getShippingState () % >
getShippingZip	Returns the shipping ZIP Code for the account from LDAP.	public String getShippingZip()	< % bean. getShippingZip() % >
getShippingCountry	Returns the shipping country for the account from LDAP.	public String getShippingCountry()	< % bean. getShippingCountry () % >

Line Item Functions

Function	Description	Syntax	Example

hasMoreLineItems	Returns true while there are still more line items in the quote to be displayed.	public boolean hasMoreLineItems()	< %
			while (bean.hasMoreLineItems())
			{
			bean.nextLineItem();
			
			while (bean.hasMoreParts())
			{
			bean.nextPart();
			//display quote line items……………..
			}
			}
			% >

nextLineItem	Moves to the next line item to be displayed.	public void nextLineItem()	< %
			while (bean.hasMoreLineItems())
			{
			bean.nextLineItem();
			
			while (bean.hasMoreParts())
			{
			bean.nextPart();
			//display quote line items……………..
			}
			}
			% >

getLineItemID	Returns the numeric unique identifier for the line item.	public int getLineItemID()	< % bean.getLineItemID() % >

Function	Description	Syntax	Example
getLineItemIndex	Returns the counter value (index) for the current line item (1, 2, and so on).	public int getLineItemIndex ()	< % bean.getLineItemIndex() % >
getLineItemDescr	Returns the description field for that line item (should come from the pageset variable FAMILY_NAME).	public String getLineItemDescr()	< % bean.getLineItemDescr () % >
getLineItemDiscount	Returns the user-input discount amount for the line item. Should be a number between 0 and 100 with two places of decimal precision.	public String getLineItemDiscount()	< % bean.getLineItemDiscount() % >
getLineItemLinkbackURL	Returns the user-input discount amount for the line item. Should be a number between 0 and 100 with two places of decimal precision.	public String getLineItemDiscount()	< % bean.getLineItemDiscount() % >
getLineItemDeleteURL	Returns the URL the user can click on to remove this line item from the quote.	public String getLineItemDeleteURL()	< % bean.getLineItemDeleteURL( ) % >

Part Functions

Function	Description	Syntax	Example

hasMoreParts	Returns true while the current line item still has more parts to display.	public boolean hasMoreParts ()	< %
			 while (bean.hasMoreParts())
			 {
			 bean.nextPart();
			 //display quote line items.................
			 }
			 % >

nextPart	Returns the next part number in the list for the current line item.	public void nextPart()	< %
			 while (bean.hasMoreParts())
			 {
			 bean.nextPart();
			 //display quote line items.................
			 }
			 % >

Function	Description	Syntax	Example
getPartLevel	Since parts can belong to other parts as "children," this function returns an integer value designating the "nested" level of the current part. Therefore, if the current part is at the top level and is not a child itself, the level would be 0, but if the next part in the list was a child of this part then this function would return 1. Parts are always returned in the order of their family tree hierarchy, such that you begin with a parent, and then its children, and then its children's children, and so on before returning to the parent's siblings. You can use the getPartLevel function to format the display of the parts such that it is more obvious which children belong to which parent, by inserting a certain number of tabs or spacing characters according to the part level.	public int getPartLevel()	< % bean. getPartLevel() % >
getPartID	Returns the numeric unique identifier for the part.	public int getPartID()	< % bean. getPartID() % >
getPartIndex	Returns the numeric index (counter) for the part. Numbering of parts starts over at one for each line item.	public int getPartIndex()	< % bean.GetPartIndex () % >
getPartConfigData	Returns the configuration data string for the part including all config and feature table data that was saved for this part number. The ORDER_SUBVAR variable in the app_config.js file determines which tables contain part numbers, and the data from the currently selected row in that table is associated with the part number it contains.	public String getPartConfigData()	< % bean.getPartConfigData() % >
getLineItemLinkbackURL	Returns the user-input discount amount for the line item. Should be a number between 0 and 100 with two places of decimal precision.	public String getLineItemDiscount( )	< % bean.getLineItemDiscount() % >
getLineItemDeleteURL	Returns the URL the user can click on to remove this line item from the quote.	public String getLineItemDeleteUR L()	< % bean.getLineItemDeleteURL () % >

Function	Description	Syntax	Example
getPartParentID	Returns the unique identifier of the parent item (part), if any. If the part has no parent, returns zero.	public int getPartParentID()	< % bean.getPartParentID() % >
getPartLineItemID	Returns the unique numeric identifier for the line item containing this part.	public int getPartLineItemID()	< % bean.getPartLineItemID() % >
getPartQty	Returns a numeric quantity for the quantity of this part the user has selected. Default value is 1. The example below shows quantity being displayed as an editable, updatable field. The name for the updatable quantity field must be constructed as in the example below.	public String getPartQty()	< td > < font face = "VERDANA" size = -2 > < input type = "text" name = " < % = bean.getPartPrefix() % > _QTY" value = " < % = bean.getPartQty() % > " size = 5 > < /font > < /td >
getPartDescr	Returns the description for this part as given in the data model.	public String getPartDescr()	< tr > < td > < font face = "VERDANA" size = -2 > < t > < % = bean.getPartDescr() % > < / font > < /td >
getPartNum	Returns the part number for this part as given in the ORDER_SUBVAR column in the data model.	public String getPartNum()	< td > < font face = "VERDANA" size = -2 > < % = bean.getPartNum() % > < /font > < /td >
getPartPrice	Returns the price for this part as given in the data model.	public String getPartPrice()	< td > < font face = "VERDANA" size = -2 > < % = bean.getPartPrice() % > < /font > < /td >
getPartSubPrice	Instead of price, you may specify subprice, which means that the price of this part is equal to the sum of the prices of its children parts or, when no children are available, the price is from the data model.	public String getPartSubPrice()	< td > < font face = "VERDANA" size = -2 > < % = bean.getPartSubPrice() % > < /font > < /td >
getPartExtSubPrice	Subprice from above multiplied by part quantity.	public String getPartExtSubPrice()	< td > < font face = "VERDANA" size = -2 > < % = bean.getPartExtSubPrice() % > < /font > < /td >

Function	Description	Syntax	Example
getPartExtPrice	Price from above multiplied by part quantity.	public String getPartExtPrice()	< td > < font face = "VERDANA" size = -2 > < % = bean.getPartExtPrice() % > < /font > < /td > < / tr >
getPartItemized	Returns TRUE or FALSE, whether this part is itemized, from the pageset variable “PAGESET_ITEMIZED.”	public String getPartItemized()	< % bean.getPartItemized() % >
getMatchingPartFields	Returns an array of matching part fields, where the field names either designate user-created custom fields or columns from the data model data tables. Here is an example: The first statement declares an array of KeyValuePair objects as the return value for the getMatchingPartFields function call. This array will be populated by any and all fields and values that have “PRM” as part of their field name. The next statement looks to see if any matching fields were returned for this part. “matches.length” returns the number of entries in the array and if it is greater than zero, we have at least one matching field. Since we know that there are no other fields in our data model containing “PRM,” we can assume that the first array record has the value we are interested in, and we can print it into a table cell labeled “PRM: ”. That is what the next lines do. First, we close off the code block since we want to print something out to the screen. Then we make our HTML table cell with the label we want and print the value of the first array record using “ < % = matches[0].value % > ”. The “matches[0]” specifies the first KeyValuePair object in the array, and the “.value” says to access the value field (“.key” would give us the field name).	public KeyValuePair[] getMatchingPartField s (String key_match)	< % KeyValuePair [] matches = bean.getMatchingPartFields(“PRM”); if (matches.length > 0) { % > < td > PRM: < % = matches[0].value % > < /td > < % } % >

Function	Description	Syntax	Example
getPartField	Same as the above getMatchingPartFields, but does an exact match on the field name, and so returns only a single string value. If no exact match is found, returns empty string (“”).	public String getPartField(String name)	< input type = ”text” name = ” < % = bean.getPart Prefix() % > _Special_Instruc tions” value = ” < % = bean.getPart Field (“_Special_Instructions”) % > ” >
getPartPrefix	Returns the prefix used to save customer-created fields associated with a particular line item (as opposed to the header fields, which only provide a single field entry for the entire quote). This prefix is “ITEM_” concatenated with the current line item index and the current part number index. Use the prefix in the name field of the form input field that you wish to save with the line item, followed by the name you create.	public String getPartPrefix()	< input type = ”text” name = ” < % = bean.getPart Prefix() % > _Special_Instruc tions” value = ” < % = bean.getPart Field (“_Special_Instructions”) % > ” >

Footer Functions

Function	Description	Syntax	Example
getQuoteSubtotal	Returns the subtotal for the quote (equal to the sum of the extended prices).	public String getQuoteSubtotal()	<td colspan=2 align=left><font face="VERDANA" size=-2><b>Subtotal:</b></font></td> <td><font face="VERDANA" size=-2><%= bean.getQuoteSubtotal() %></font></td></tr>
getQuoteDiscount	Returns the discount percent for the quote, which is user-definable. This should be entered as a number between 0 and 100 with up to two decimal places of precision. If no discount percent has yet been defined for this quote, returns zero.	public String getQuoteDiscount()	<td colspan=2 align=left><font face="VERDANA" size=-2><b>Discount %:</b></font></td> <td><font face="VERDANA" size=-2><input type="text" name="QUOTE_DISCOUNT" size=5 value="<%= bean.getQuoteDiscount() %>"></font></td>
getQuoteTotal	Returns the total amount for this quote, including any discount.	public String getQuoteTotal()	<td><font face="VERDANA" size=-2><%= bean.getQuoteTotal() %></font></td>

Additional Code G

This appendix covers additional code used with Siebel Transact. It includes:

- XML Default Data Definition
- XSLT Style Sheet Example
- HTML Form Post of Shopping Cart Contents

XML Default Data Definition

1.1.Data Description

1.1.1.Line Item Data Description

There will be two levels of entities describing a package. The top level entity, called LineItem, is a placeholder to contain all the Items which have been selected from the pageset. The children of the LineItem entity will be called Items and will contain configuration information, and whether the Item is a child of another Item in the LineItem.

LineItem Entity Fields

* Id
* User
* Pageset
* Pageset Version
* Date Created
* Date Submitted
* Name (if any)
* Account Id (if any)
* Quote Id (if any)
* Environment data at the pageset level

Item Entity Fields

* Id
* LineItem Id (references Id field of LineItem Entity above)
* Qty
* Config Data
* Engine Data
* Parent Id (if any)

1.1.2.Quote Data Description

The quote data description is very simple, since a quote is simply a container for one or more LineItems. The LineItems reference their parent quote using the value in the Id field.

Quote Entity Fields

* Id

* Date Created

* User

* Name (if any)

* Account Id (if any)

* Date Submitted

* Header Field Data

1.1.3.Generic LineItem DTD

```
<?xml version="1.0" standalone="yes"?>

<!DOCTYPE document [

<!ELEMENT SiebelLineItem (ShippingAddress?, BillingAddress?, User?,
Item+)>

<!-siebel_id: unique identifier ?

<!-- created_on: timestamp -->

<!-- id: 3rd party unique identifier -->

<!ATTLIST SiebelLineItem

   siebel_id ID #REQUIRED

           created_on #REQUIRED

           id ID #IMPLIED

>

<!-- user element is optional, contains buyer identification
information -->
```

```
<!ELEMENT User>

<!ATTLIST User

          user_id CDATA #REQUIRED

   account_id CDATA #IMPLIED

   session_id CDATA #IMPLIED

   order_id CDATA #IMPLIED

>

<!--Price type examples: quoted, list, discount -->

<!ELEMENT Price (#PCDATA)>

<!ATTLIST Price

          type     CDATA #REQUIRED

          currency CDATA #IMPLIED

>


<!ELEMENT ConfigData (#PCDATA)>

<!ATTLIST ConfigData

   name CDATA #REQUIRED

>

<!-An item must have config data (name/value pairs) associated with
it

     An item can have zero or more items as children

   An item can have zero or more Price elements ?

<!ELEMENT Item (ConfigData+, Price*, Item*)>

<!ATTLIST Item

          part_number CDATA #IMPLIED
   quantity CDATA #REQUIRED

   description CDATA #IMPLIED

>

]>
```

1.1.4.Generic Quote DTD

```
<?xml version="1.0" standalone="yes" ?>

<!DOCTYPE document [

<!ELEMENT SiebelQuote (User?, SiebelLineItem+, Total*)>

<!ATTLIST SiebelQuote

   quote_id ID #REQUIRED

   date_created CDATA #REQUIRED

>

<!-Header field information -->

<!ELEMENT Header (ShippingAddress?, BillingAddress?, HeaderData*)>

<!ELEMENT HeaderData (#PCDATA)>

<!ATTLIST HeaderData

   name CDATA #REQUIRED>


<!-shipping address ?

<!ELEMENT ShippingAddress (Name, Street+, City, State?, 
PostalCode?,Country?>

<!-billing address ?

<!ELEMENT BillingAddress (Name, Street+, City, State?, 
PostalCode?,Country?>
```

```
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT PostalCode (#PCDATA)>
<!ELEMENT Country (#PCDATA)>

<!-Total type examples: subtotal, grand, discount ?
<!ELEMENT Total (#PCDATA)>
<! ATTLIST Total
   type CDATA #REQUIRED
   currency CDATA #IMPLIED
>
(See SiebelLineItem for the rest of the DTD)
```

XSLT Style Sheet Example

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">


  <xsl:output indent="yes"/>


  <!-- ******************************************

  get all the parameters that are passed from outside

  *********************************************** -->

  <xsl:param name="payloadid"/>

  <xsl:param name="timestamp"/>

  <xsl:param name="version" select="'1.0'"/>

  <xsl:param name="locale" select="'en-US'"/>


  <xsl:param name="from_domain"/>

  <xsl:param name="to_domain"/>

  <xsl:param name="sender_domain"/>

  <xsl:param name="from_identity"/>

  <xsl:param name="to_identity"/>

  <xsl:param name="sender_identity"/>

  <xsl:param name="sender_secret"/>

  <xsl:param name="user_agent"/>
```

```
   <!-- ******************************************

   the template that matches the root item

   ************************************************ -->



   <xsl:template match="SiebelQuote">

    <cXML>



        <!-- *******************************************

          setup the attributes for the cXML element

     ************************************************ -->

     <xsl:attribute name="version"><xsl:value-of select="$version"/
></xsl:attribute>

     <xsl:attribute name="payloadID"><xsl:value-of
select="$payloadid"/></xsl:attribute>

     <xsl:attribute name="timestamp"><xsl:value-of
select="$timestamp"/></xsl:attribute>

     <xsl:attribute name="xml:lang"><xsl:value-of select="$locale"/
></xsl:attribute>


     <!-- *******************************************

          setup the values for the variables

     ************************************************ -->

     <xsl:variable name="order_id" select="./User/@order_id"/>


     <!-- *******************************************

          setup the cXML header

     ************************************************ -->
```

```
            <Header>

              <From>

                <Credential>

                   <xsl:attribute name="domain"><xsl:value-of
select="$from_domain"/></xsl:attribute>

                   <Identity>

                     <xsl:value-of select="$from_identity"/>

                   </Identity>

                </Credential>

              </From>

              <To>

                <Credential>

                   <xsl:attribute name="domain"><xsl:value-of
select="$to_domain"/></xsl:attribute>

                   <Identity>

                     <xsl:value-of select="$to_identity"/>

                   </Identity>

                </Credential>

              </To>

              <Sender>

                <Credential>

                   <xsl:attribute name="domain"><xsl:value-of
select="$sender_domain"/></xsl:attribute>

                   <Identity>

                     <xsl:value-of select="$sender_identity"/>

                   </Identity>
            <SharedSecret>
              <xsl:value-of select="$sender_secret"/>
            </SharedSecret>
         </Credential>
         <UserAgent>
           <xsl:value-of select="$user_agent"/>
         </UserAgent>
       </Sender>
     </Header>

     <!-- **********************************************
          setup the cXML OrderRequest (Header & ItemOut)
     *************************************************** -->
     <OrderRequest>
      <OrderRequestHeader>

        <xsl:attribute name="type">new</xsl:attribute>
        <xsl:attribute name="orderID"><xsl:value-of
select="$order_id"/></xsl:attribute>

        <!-- **********************************************
             TOTAL
        *************************************************** -->
        <Total>
          <Money>
```

```
            <xsl:choose>

              <xsl:when test="./Total">

                  <xsl:attribute name="currency"><xsl:value-of
select="./Total[position()=1]/@currency"/></xsl:attribute>

                  <xsl:value-of select="./Total[position()=1]"/>

              </xsl:when>

              <xsl:otherwise>

                  <xsl:attribute name="currency"><xsl:value-of
select="'USDollars'"/></xsl:attribute>

                  <xsl:value-of select="0.0"/>

              </xsl:otherwise>

            </xsl:choose>

          </Money>

        </Total>


        <!-- ***********************************************

             SHIPPING AND BILLING ADDRESSES

        *************************************************** -->

        <xsl:apply-templates select="Header/ShippingAddress"/>

        <xsl:apply-templates select="Header/BillingAddress"/>



        <!-- ***********************************************

             OTHER Siebel SPECIFIC DATA

        *************************************************** -->

        <Extrinsic>
```

```
                        <xsl:attribute name="name">quote_id</xsl:attribute>
                        <xsl:value-of select="./@quote_id"/>
                     </Extrinsic>
                     <Extrinsic>
                        <xsl:attribute name="name">date_created</xsl:attribute>
                        <xsl:value-of select="./@date_created"/>
                     </Extrinsic>
                     <xsl:apply-templates select="./Header/HeaderData"/>

                     <!-- **********************************************
                          ITEM INFORMATION
                     *************************************************** -->
                     <xsl:apply-templates select="SiebelLineItem"/>



                    </OrderRequestHeader>
                  </OrderRequest>


                </cXML>


              </xsl:template>
            <xsl:template match="ShippingAddress">
                <ShipTo>
                 <Address>
                  <Name>
```

```
       <xsl:attribute name="xml:lang"><xsl:value-of
select="$locale"/></xsl:attribute>
       <xsl:value-of select="Name"/>
      </Name>
      <Street><xsl:value-of select="Street"/></Street>
      <City><xsl:value-of select="City"/></City>
      <State><xsl:value-of select="State"/></State>
      <Country><xsl:value-of select="Country"/></Country>
      <PostalCode><xsl:value-of select="PostalCode"/></PostalCode>
     </Address>
    </ShipTo>
  </xsl:template>

  <xsl:template match="BillingAddress">
    <BillTo>
     <Address>
      <Name>
       <xsl:attribute name="xml:lang"><xsl:value-of
select="$locale"/></xsl:attribute>
       <xsl:value-of select="Name"/>
      </Name>
      <Street><xsl:value-of select="Street"/></Street>
      <City><xsl:value-of select="City"/></City>
      <State><xsl:value-of select="State"/></State>
      <Country><xsl:value-of select="Country"/></Country>
      <PostalCode><xsl:value-of select="PostalCode"/></PostalCode>
     </Address>

    </BillTo>

  </xsl:template>


  <xsl:template match="HeaderData">

    <Extrinsic>

      <xsl:attribute name="name"><xsl:value-of select="@name"/></
xsl:attribute>

      <xsl:value-of select="."/>

    </Extrinsic>

  </xsl:template>


  <xsl:template match="SiebelLineItem">

    <xsl:apply-templates select=".//Item">

      <xsl:with-param name="itemUser" select="./User"/>

    </xsl:apply-templates>

  </xsl:template>


  <xsl:template match="Item">

     <ItemOut>

       <xsl:attribute name="quantity"><xsl:value-of
select="@quantity"/></xsl:attribute>

       <xsl:attribute name="lineNumber"><xsl:value-of select="../
@id"/></xsl:attribute>

       <ItemID>

         <SupplierPartID>
```

```
          <xsl:value-of select="@part_number"/>
         </SupplierPartID>
       </ItemID>
       <ItemDetail>
          <UnitPrice>
            <Money>
              <xsl:choose>
                  <xsl:when test="./Price">
                    <xsl:attribute name="currency"><xsl:value-of
select="./Price[position()=1]/@currency"/></xsl:attribute>
                    <xsl:value-of select="./Price[position()=1]"/>
                  </xsl:when>
                  <xsl:otherwise>
                    <xsl:attribute name="currency"><xsl:value-of
select="'USDollars'"/></xsl:attribute>
                    <xsl:value-of select="0.0"/>
                  </xsl:otherwise>
              </xsl:choose>
            </Money>
          </UnitPrice>
          <UnitOfMeasure></UnitOfMeasure>
          <Description>
            <xsl:value-of select="@description"/>
          </Description>
          <xsl:if test="$itemUser">
           <Extrinsic>
```

```
                <xsl:attribute name="name">userID</xsl:attribute>
                <xsl:value-of select="$itemUser/@user_id"/>
             </Extrinsic>
             <Extrinsic>
                <xsl:attribute name="name">accountID</xsl:attribute>
                <xsl:value-of select="$itemUser/@account_id"/>
             </Extrinsic>
             <Extrinsic>
                <xsl:attribute name="name">sessionID</xsl:attribute>
                <xsl:value-of select="$itemUser/@session_id"/>
             </Extrinsic>
            </xsl:if>
            <Extrinsic>
                <xsl:choose>
                    <xsl:when test="./Price">
                       <xsl:attribute name="name">parentID</
xsl:attribute>
                       <xsl:value-of select="../@part_number"/>
                    </xsl:when>
                    <xsl:otherwise>
                       <xsl:attribute name="name">parentID</
xsl:attribute>
                       <xsl:value-of select="0"/>
                    </xsl:otherwise>
                </xsl:choose>
            </Extrinsic>
```

```
          <xsl:apply-templates select="./ConfigData"/>

       </ItemDetail>

     </ItemOut>

  </xsl:template>


  <xsl:template match="ConfigData">

    <Extrinsic>

      <xsl:attribute name="name"><xsl:value-of
select="concat('cfg_', @name)"/></xsl:attribute>

      <xsl:value-of select="."/>

    </Extrinsic>

  </xsl:template>


</xsl:stylesheet>
```

HTML Form Post of Shopping Cart Contents

```
<%@ page

        info="Shopping Cart Form Submission"

        contentType="text/html"

%>

<jsp:useBean id="bean"
class="com.siebel.isscda.wl.transact.ShoppingCartBean">

   <%

     bean.setRequest(request);

     bean.setSession(session);

     bean.getQuote();

   %>

</jsp:useBean>

<!doctype html public "-//w3c/dtd HTML 4.0//en">

<html>

<head>

</head>

<body bgcolor=#FFFFFF onLoad="document.checkout_form.submit();">

<form name="checkout_form" action="http://www.mycompany.com/
checkout.cgi" method=POST>

<input type="hidden" name="QUOTE_NAME" value="<%=
bean.getQuoteName() %>">

<input type="hidden" name="QUOTE_ID" value="<%= bean.getQuoteID()
%>">

<input type="hidden" name="QUOTE_ACCOUNT" value="<%=
bean.getQuoteAccountID() %>">
```

```
<input type="hidden" name="QUOTE_DATE_CREATED" value="<%=
bean.getQuoteDateCreated() %>">

<input type="hidden" name="QUOTE_DATE_MODIFIED" value="<%=
bean.getQuoteDateModified() %>">

<% if (bean.getQuoteHeader("BILLING_ADDR") != "") { %>

<input type="hidden" name="QUOTE_BILLING_ADDR"
value="<%=  bean.getQuoteHeader("BILLING_ADDR") %>">

<% }else { %>

<textarea rows=5 name="HEADER_BILLING_ADDR"><%=
bean.getBillingName() %>

<%= bean.getBillingStreet() %>

<%= bean.getBillingCity() %>, <%= bean.getBillingState() %>     <%=
bean.getBillingZip() %>

<%= bean.getBillingCountry() %></textarea>

<% } %>

<% if (bean.getQuoteHeader("SHIPPING_ADDR") != "") { %>

   <input type="hidden" name="QUOTE_SHIPPING_ADDR"
value="<%=  bean.getQuoteHeader("SHIPPING_ADDR") %>">

<% }else { %>

<textarea rows=5 name="HEADER_SHIPPING_ADDR"><%=
bean.getShippingName() %>

<%= bean.getShippingStreet() %>

<%= bean.getShippingCity() %>, <%= bean.getShippingState() %>    <%=
bean.getShippingZip() %>

<%= bean.getShippingCountry() %>

</textarea>

<% } %>

<%

    while ( bean.hasMoreLineItems() )

    {
        bean.nextLineItem();


        while ( bean.hasMoreParts() )

        {

            bean.nextPart();

%>

<input type="hidden" name="<%= bean.getPartPrefix()%>_DESCRIPTION"
value="<%= bean.getPartDescr() %>">

<input type="hidden" name="<%= bean.getPartPrefix()%>_PART_NUM"
value="<%= bean.getPartNum() %>">

<input type="hidden" name="<%= bean.getPartPrefix()%>_QTY"
value="<%= bean.getPartQty() %>">

<input type="hidden" name="<%= bean.getPartPrefix()%>_PRICE"
value="<%= bean.getPartPrice() %>">

<input type="hidden" name="<%= bean.getPartPrefix()%>_EXT_PRICE"
value="<%= bean.getPartExtPrice() %>">

<%

     }//end while


  }//end while

%>

<input type="hidden" name="QUOTE_SUBTOTAL" value="<%=
bean.getQuoteSubtotal() %>">

<input type="hidden" name="QUOTE_DISCOUNT" value="<%=
bean.getQuoteDiscount() %>">

<input type="hidden" name="QUOTE_TOTAL" value="<%=
bean.getQuoteTotal() %>">

</form>

</body>

</html>
```

Transact Server Localization H

Currently, there is no localized resource bundle and jsp files provided. The user has to manually localize OLResource resource bundle property and jsp files.

Transact Server

For the Transact Server resource bundle file (for example, OLResource_en_US.properties), you can specify locale_language and locale_country from the property editor so the application will look for the file name

OLResource_LOCALE_LANGUAGE_LOCALE_COUNTRY.properties

for displaying error and warning messages.

For example, you can translate English OLResource_en_US.properties provided by the Transact Server into Japanese OLResource_ja_JP.properties for Japanese resource bundle file.

For further information, please refer to the following links for resource bundle file.

http://java.sun.com/products/jdk/1.1/docs/api/java.util.ResourceBundle.html

http://java.sun.com/products/jdk/1.1/docs/api/java.util.Locale.html#_top_

Transact Server JSP Files

For localizing Transact Server jsp files, there is a Meta tag from HTML which can be used to specify the language character set.

For example, if users specify meta tag " < meta http-equiv = 'Content-Type' content = 'text/html; charset = big5' > " from jsp pages, the browser will translate HTML page characters as Chinese big5 characters. Users will need to install a language package based on the browsers needed in order to translate the jsp page correctly.

Additional Tasks

I

This appendix covers additional tasks that you can perform while setting up Siebel Transact. It includes:

- Set Up JDBC and Data Source for WebSphere
- Change the DB2 Connection
- Un-Install Transact LDAP
- Block Display of Shopping Cart

Set Up JDBC and Data Source for WebSphere

Follow these steps to set up the JDBC driver and data source for WebSphere.

To set up the JDBC driver and data source for WebSphere

1. Start WebSphere WAS AdminServer.
2. Install JDBC Driver.
   - **a** From the View menu, select Type.
   - **b** In the left-side view panel, right-click JDBC Driver and select Create.
   - **c** Select Driver and enter a driver name.

     This can be any name. Remember this driver name for the data source (for example, DB2JDBCDriver).
   - **d** Select com.ibm.db2.jdbc.app.DB2Driver if you use DB2. Select oracle.jdbc.driver.OracleDriver for Oracle database.
   - **e** Select jdbc:db2 for URL Prefix for DB2. Modify hostname from URL prefix to Oracle server hostname from Oracle database.
   - **f** Select False for JTA Enabled.
3. Install the Data Source.
   - **a** In the left-side view panel, right-click DataSource and select Create.
   - **b** Enter a DataSource Name. (For example: Sample is already installed with DB2.)
   - **c** Enter the JDBC driver name in the Database Name field.
4. From the View menu, select Topology.
5. In the left-side view panel, right-click DB2JDBCDriver and select Install.
6. Select the machine on which the user wants to install the JDBC driver.
7. Click Browse, select <SQLLIB root>/java/db2java.zip file for db2 or select classes12.zip from <oracle folder>/jdbc/lib for Oracle database, and click Open.

8 Click Install.

The JDBC driver will be installed.

For more information on WAS, visit the following URL:
http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/
index.html

Change the DB2 Connection

Follow these directions if you want to change the DB2 database connection. This requires you to modify the Admin.config file as follows.

To change the DB2 connection

1. For “com.ibm.ejs.sm.adminServer.dbUrl = jdbc:db2:was,” replace “was” with the current DB2 database name.
2. For “com.ibm.ejs.sm.adminServer.dbUser = db2admin,” replace “db2admin” with the current user connecting to the database, as specified above.
3. For “com.ibm.ejs.sm.adminServer.dbPassword = db2admin,” replace “db2admin” with the current password of the user connecting to the database, as specified above.

Un_Install Transact LDAP

Follow these steps to uninstall Transact LDAP:

- Run transact_uninstall.bat.
- Run rincon_uninstall.bat.

Block Display of Shopping Cart

If you do not want to display a shopping cart window after you click AddToCart, follow these steps:

1 Change the value of mar_display_cart from true to false.

2 Modify the siebel.prp file:

- **a** Change the values of "mar_guest ok" from

  "AddToCart,SubmitCart,ShoppingCart.jsp,OpenPackage"

  to

  "AddToCart, SubmitCart, ShoppingCart.jsp, OpenPackage, ViewQuote.jsp, EmailQuote.jsp, EmailConfig.jsp, SavePackage, ConfigList.jsp, QuoteList.jsp, ShoppingCart_orig.jsp, ShoppingCart_Header.jsp, ShoppingCart_Bottom.jsp, ShoppingCart_Middle.jsp"

- **b** You may have different file names for your shopping cart, or you may have one or more files—Replace ShoppingCart_Header.jsp, ShoppingCart_Bottom.jsp, ShoppingCart_Middle.jsp, based upon your configurations.

Index