ATG WEB COMMERCE

Version 10.1.2

Commerce Programming Guide

Oracle ATG
One Main Street
Cambridge, MA 02142
USA

ATG Commerce Programming Guide

Product version: 10.1.2
Release date: 12-17-12
Document identifier: CommerceProgrammingGuide1404301402

Table of Contents

1 Introduction

Oracle ATG Web Commerce serves as the foundation for your online store. It contains everything you need to manage your product database, pricing, inventory, fulfillment, merchandising, targeted promotions, and customer relationships.

This chapter includes the following sections:

Commerce Overview

This chapter introduces you to the major features of Oracle ATG Web Commerce:

Product Catalog

The product catalog is a collection of repository items (categories, products, media, etc.) that provides the organizational framework for your commerce site. Oracle ATG Web Commerce includes a catalog implementation based on the SQL Repository, which you can use or extend as necessary.

You can create and edit all of your repository items through the ATG Control Center, which also allows you to create page templates to display these items (see the *ATG Commerce Guide to Setting Up a Store*), or through Oracle ATG Web Commerce Merchandising (see the *ATG Merchandising Guide for Business Users*).

Purchasing and Fulfillment Services

Oracle ATG Web Commerce provides tools to handle pre-checkout order-processing tasks such as adding items to a shopping cart, ensuring items are shipped by the customer's preferred method, and validating credit card information. The system is designed for flexibility and easy customization; you can create sites that support multiple shopping carts for a single user, multiple payment methods and shipping addresses, or share carts across multiple sites.

As soon as a customer submits an order, the fulfillment framework takes over processing. This system includes a collection of standard services which coordinate and execute the order fulfillment process. Like the purchase process, the fulfillment framework can be customized to meet the needs of your sites.

Commerce also includes an HTML-based Fulfillment Administration page that you can use for:

- Viewing orders that are ready to be shipped.
- Notifying the fulfillment system that an order has been shipped to the customer.
- Notifying the fulfillment system that a shipping group has changed and needs to be reprocessed.
- Printing order information.

Commerce allows you to export customer orders in XML for easy integration with your other systems. Your customers can also create template orders from a new or existing order, and then create a schedule for the same order to be placed regularly during the time frame they specify. For example, a company could set up a scheduled order to buy certain supplies on a monthly basis for the next year.

Inventory Management

The inventory framework facilitates inventory querying and inventory management for your sites. It allows you to:

- Remove items from inventory.
- Notify the store of a customer's intent to purchase an item that is not currently in stock (backorder) or has never been in stock (preorder).
- Make a specific number of items available for customers to purchase, backorder, or preorder.
- Determine and modify the number of items available for purchase, backorder, or preorder.
- Determine when a specific item will be in stock.

Inventory information is stored in the Inventory repository, which is separate from the product catalog. You can use the ATG Control Center (ACC) to view, add and delete inventory items, and edit their property values.

Oracle ATG Web Commerce also includes an HTML-based administration interface for the Inventory Manager. Administrators can use this interface to view the results of the inventory query operations, manipulate the various properties of each item, and notify the system of inventory updates.

Pricing Services

Oracle ATG Web Commerce pricing services revolve around pricing engines and pricing calculators. The pricing engine determines the correct pricing model for an order, individual item, shipping charge, or tax, based on

a customer's profile. The pricing calculator performs the actual price calculation based on information from the pricing engine. These services make it possible to generate prices dynamically under constantly changing business conditions.

The price lists feature allows you to target a specific set of prices to a specific group of customers. Price lists are managed through the ACC (see the *ATG Commerce Guide to Setting Up a Store*) or through Oracle ATG Web Commerce Merchandising (see the *ATG Merchandising Guide for Business Users*). For example, price lists can be used to implement sales prices. They can also be used for business-to-business pricing where each customer can have its own unique pricing for products based on contracts, RFQ and pre-negotiated prices.

Targeted Promotions

Business managers can use Oracle ATG Web Commerce promotions to highlight products and offer discounts as a way of encouraging customers to make purchases. Promotions typically fall into the following categories:

- Specific amount off a particular product
- Specific amount off a whole order
- Percentage amount off a particular product
- Percentage amount off a whole order
- Specific amount or percentage off a product based on an attribute
- Free product or free order
- Substitution (buy product A for the price of product B)
- Free shipping for a specific product

You can create promotions using Oracle ATG Web Commerce Merchandising (see the *ATG Merchandising Guide for Business Users*).

Commerce Services

Oracle ATG Web Commerce provides services for implementing a variety of features on your commerce site.

- **Gift Lists and Wish Lists**

  Gift lists allow customers to register for an event, such as a birthday or wedding, and create a list of products that other site visitors can view. Customers can create an unlimited number of gift lists for themselves. Part of the purchase process allows special handling instructions for gift purchases, such as address security, wrapping and shipping.

  Wish lists allow customers to save lists of products without actually placing the items in their shopping cart. A wish list is similar to a gift list, except that it is only accessible to the person who created it. Customers can access their wish lists and purchase items from it at any time.

- **Comparison lists**

  Comparison lists enable customers to select multiple product SKUs and compare them side-by-side.

- **Gift Certificates and Coupons**

You can set up gift certificates as an item in your product catalog. When a customer purchases a gift certificate, it is delivered via e-mail to the recipient, who, in turn, can use it to pay for purchases on the site.

Coupons are similar to gift certificates, except that they are a type of promotion (20% of an order over $100, for example) sent to specific customers. Customers redeem gift certificates and coupons entering a claim code during the checkout process.

- **Cost Centers**

  Cost centers allow your customers to track internal costs by designating parts of their organization as cost centers, enabling them to track costs by department.

- **Order Approvals**

  You can identify customers for whom approvals are required, and check for the conditions that trigger an approval for an order, such as when an order limit is exceeded. After an approver has reviewed the order, if approved, the order proceeds through checkout.

- **Scheduled Orders**

  You can create orders to be fulfilled repeatedly on a specific schedule, or construct and save orders to be placed at a later date.

- **Invoicing**

  This feature gives your customers the option of being invoiced for orders they place.

- **Requisitions**

  Requisitions work with the order approval process, enabling your customers to attach requisition numbers to orders, then submit them for approval within their organization, improving your customers' ability to track internal activities.

- **Contracts**

  Contracts allow you to associate a particular catalog, price list(s), and payment terms with a specific organization.

You can use Oracle ATG Web Commerce Merchandising (see the *ATG Merchandising Guide for Business Users*) or the ACC (see the *ATG Commerce Guide to Setting Up a Store*) to manage repository items for these features.

Reporting

Oracle ATG Web Commerce is fully integrated with Oracle ATG Web Commerce Business Intelligence, and includes a default set of reports that can provide essential information on store performance. See the *ATG Reports Guide* for detailed information on these reports. See the *Preparing to Use Commerce Reporting* (page 489) chapter of this guide for configuration information.

Multisite Integration

Oracle ATG Web Commerce's multisite feature allows you to build and launch new sites quickly, and to manage brands, country stores, and other differentiators efficiently across multiple channels. This section describes some of the aspects of multisite that are important in a Commerce application.

- Site Context—Within a user's session, the site context identifies what catalogs, products, or SKUs are available to the user, which price list to apply, and which shopping cart to use.
- Site Membership—Defines the sites to which a catalog and its items belong. These items can include catalogs, categories, products, SKUs, and catalog folders. Catalogs and other items can belong to more than one site.
- `SiteIdForItemDroplet` and `SiteLinkDroplet`—These platform droplets (see the *ATG Page Developer's Guide*) are useful for Commerce developers. Items that appear in multiple catalogs can be displayed together; when a customer selects one, you can specify which site's version of the details to use.
- Shopping Cart—The cart tracks the site on which it was created (when the customer adds the first item), on which each item was added, and on which the most recent activity occurred.
- Scheduled Orders—These orders include site information when creating and pricing orders.
- Gift, Purchase, and Wish Lists—All of these track the site on which they were created and on which each item was added.
- Shared Carts and Wish Lists—You can configure shopping carts and gift/purchase/wish lists to be shared among sites.
- Searching—Search form handlers are site-aware and can be constrained by site.
- Reports—All Commerce reports include site information. See the *ATG Reports Guide*.

Information on the multisite uses of Commerce features can be found throughout this guide, where applicable. See the *ATG Multisite Administration Guide* for general information on implementing multisite in Oracle ATG Web Commerce applications.

Finding What You Need

Oracle ATG Web Commerce is a comprehensive product that provides the tools you need to create a commerce Web site that's customized to meet the particular needs of your business. Instructions for working with Commerce can be found in a variety of books. Here's a key to finding the information you need:

Tasks	Audience	Instructions
Installing Oracle ATG Web Commerce	System Administrators, Programmers	*ATG Installation and Configuration Guide* Also see *Configuring and Populating a Production Database* (page 7) in this guide.
Installing Oracle ATG Web Commerce databases in a production environment.	Site Administrators	See *Configuring and Populating a Production Database* (page 7) in this guide. Users who also have Oracle ATG Web Commerce Merchandising should see the *ATG Merchandising Administration Guide* instead.

Tasks	Audience	Instructions
Installing database tables in support of Oracle ATG Web Commerce Merchandising	Site Administrators	*ATG Merchandising Administration Guide*
Extending Oracle ATG Web Commerce programmatically by creating subclasses and modifying repositories.	Programmers	This guide.
Building JSPs that use Oracle ATG Web Commerce servlet beans.	Page Developers	*ATG Commerce Guide to Setting Up a Store*
Assembling applications that include Oracle ATG Web Commerce.	Site Administrators	*ATG Platform Programming Guide*
Working with promotions, price lists, abandoned orders, scenarios, and cost centers.	Business Users	*ATG Commerce Guide to Setting Up a Store*
Creating a catalog and populating it with categories, products and SKUs using the ATG Control Center. Configuring the fulfillment and inventory tools provided with Oracle ATG Web Commerce.	Business Users	*ATG Commerce Guide to Setting Up a Store*
Developing a catalog and its categories, products, SKUs in Oracle ATG Web Commerce Merchandising.	Business Users	*ATG Merchandising Guide for Business Users*
Review database tables, session backup procedures, JMS messages, and recorders.	Site Administrators	Appendices in this guide.
Working with the Motorprise Reference Application.	All	*ATG Business Commerce Reference Application Guide*
Working with the Oracle ATG Web Commerce Reference Store.	All	*ATG Commerce Reference Store Overview*

2 Configuring and Populating a Production Database

The MySQL database included with the Oracle ATG Web Commerce platform (Windows only) is provided so that customers can become familiar with Oracle ATG Web Commerce products. You will need to install a production-ready database before you can begin building your Web application. The following sections describe how to create and configure your production database:

The information in this chapter focuses specifically on Commerce databases. For general information on production database configurations, requirements, and performance enhancements, refer to the *ATG Installation and Configuration Guide*.

Use the Configuration and Installation Manager (CIM) for database configuration. CIM ensures that all product dependencies are taken into account and that all scripts are run in the correct schema. It also greatly simplifies data source configuration. If you are using CIM, most of the information in this chapter can be used for reference only.

Warning: If your product stack includes Oracle ATG Web Commerce Merchandising, and you are not using CIM to configure your database, see the *ATG Merchandising Administration Guide* for instructions on creating the database tables you need.

Configuring Oracle ATG Web Commerce with CIM

The Configuration and Installation Manager (CIM) helps to simplify product configuration by walking you through the required steps. This ensures that all necessary steps are completed and are done in the correct order. You can use CIM to get an installation running quickly and easily.

CIM handles the following configuration steps:

- Creates database tables and imports initial data as described in this chapter, including those for the reporting data ware house.
- Creates data sources according to the database connection information you supply (as described in the *ATG Installation and Configuration Guide*), including those needed for the reporting data warehouse.
- Creates and configures Oracle ATG Web Commerce servers, including a lock manager (as described in the *ATG Installation and Configuration Guide* and the *ATG Platform Programming Guide*) and a data warehouse loader server.
- Assembles your application EAR files for each Oracle ATG Web Commerce server (as described in the *ATG Platform Programming Guide*).
- Deploys EAR files to your application server.

See the CIM help and the *ATG Installation and Configuration Guide* for additional information on CIM.

To configure Commerce using CIM, do the following:

1. Install your application server.
2. Install your Oracle ATG Web Commerce applications, using the appropriate installers for each.
3. Create your databases and set up accounts (note that you do not need to do this if you are configuring the included MySQL demo database).
4. To start CIM, go to `<ATG10dir>/home/bin` and type:

   ```
   cim
   ```

   Note: Type H at any prompt for additional information as you proceed through the configuration steps. Note also that CIM menus change dynamically depending on which products you have installed and which selections you have previously made in your configuration.
5. Select the products you want to configure.
6. Select add-ons. Add-ons are applications that support other applications, but which are unlikely to be used on their own.
7. Select any demo applications you want to install.
8. Select whether you want to configure a switching or a non-switching data source. See Creating Tables for a Switching Schema for more information on switching.
9. Select your application server.
10. Configure your database. This involves the following steps:

    - Configure data sources for each database. A data source is a configuration file that contains the connection information for your database. CIM automatically calculates which databases are required to support your installation. (For example, if you are not using switching, you probably only need to configure Publishing and Production Core data sources; if you are using switching, you must also configure your two switching data sources.)

      For the MySQL demo database, use the following values:

      User name: (see table that follows for user names)

      Password: (see table that follows for passwords)

      Host: localhost

Port: 3306

Database Name: (see table that follows for database names)

Database URL: (automatically generated)

JNDI Name: (use the CIM-generated name)

Driver Path: <ATGdir>/MySQL/mysql-connector-java-5.1.15-bin.jar

Database Name	Account User Name	Password
production_core	prod	Welcome1
publishing	pub	Welcome1
switchingA	switchA	Welcome1
switchingB	switchB	Welcome1
agent	agent	Welcome1
datawarehouse	dw	Welcome1

- Create schemas. CIM automatically runs the scripts that create your database schemas and imports any initial data required. Before you can import data into the MySQL demo database, you may be prompted to create passwords for user accounts.

11. Configure your server instances. See the *ATG Installation and Configuration Guide* for information on server instances. CIM automatically calculates the minimum number of server instances required to run your installed products. For most Commerce installations, you need at least a publishing server and a production server.

12. Assemble and deploy your application EAR files to your application server.

Note that CIM does **not** configure the following:

- Your ATG Content Administration topography. See the *ATG Content Administration Programming Guide*.
- Your Oracle ATG Web Commerce Search `IndexingOutputConfig` component. See the *ATG Search Installation and Configuration Guide*. CIM does handle some Search configuration options, such as whether you plan to index by product or by SKU, but you will most likely have to do additional configuration.

Creating Database Tables

If you have not used CIM to create your Oracle ATG Web Commerce tables, you can create them using the scripts described in this section.

1. Create tables for Oracle ATG Web Commerce platform by following the instructions provided in the *ATG Installation and Configuration Guide*.

2. Create tables for Oracle ATG Web Commerce by running the `dcs_ddl.sql` script from the following directory:

   `<ATG10dir>/DCS/sql/install/`*`database-vendor`*

   Note that if you want to run the Motorprise Reference Application, you can instead follow the instructions provided in Creating Motorprise Reference Application Tables (page 12).

The `dcs_ddl.sql` script includes the subscripts listed in the table below. If necessary, you can run these subscripts individually from the following directory:

`<ATG10dir>/DCS/sql/db_components/`*`database-vendor`*

---

production_core schema

- `claimable_ddl.sql`
- `commerce_user.sql`
- `dcs_mappers.sql`
- `inventory_ddl.sql`
- `order_ddl.sql`
- `contracts_ddl.sql`
- `organization_ddl.sql`
- `order_markers_ddl.sql`
- `user_giftlist_ddl.sql`
- `user_promotion_ddl.sql`
- `invoice_ddl.sql`
- `priceLists_ddl.sql`
- `product_catalog_ddl.sql`
- `custom_catalog_ddl.sql`
- `promotion_ddl.sql`

publishing schema

- `commerce_user.sql`
- `inventory_ddl.sql`
- `user_promotion_ddl.sql`
- `order_ddl.sql`
- `contracts_ddl.sql`
- `user_giftlist_ddl.sql`
- `dcs_mappers.sql`

- `order_markers_ddl.sql`
- `invoice_ddl.sql`
- `organization_ddl.sql`

The following scripts can be found in the `<ATG10dir>/DCS/sql/db_components/`*`database-vendor`* directory:

- `versioned_claimable_ddl.sql`
- `versioned_commerce_site_ddl.sql`
- `versioned_priceLists_ddl.sql`
- `versioned_product_catalog_ddl.sql`
- `versioned_custom_catalog_ddl.sql`
- `versioned_promotion_ddl.sql`

For descriptions of individual Oracle ATG Web Commerce database tables, see Appendix B, *Oracle ATG Web Commerce Databases* (page 539).

If you are using an Oracle or MSSQL database with Commerce, see Using Oracle ATG Web Commerce with an Oracle Database (page 12) and Using Oracle ATG Web Commerce with an MSSQL Database (page 14) respectively for important information.

Creating Tables for a Switching Schema

A common practice for Oracle ATG Web Commerce installations is to use a switching database configuration. This means in essence that you have two copies of your database, only one of which is online at any given time. Updates can be made to the offline database, which is then switched to be the live database while the second database is updated. If you are not planning to use database switching, you can run all of your Commerce database creation scripts against a single database, as described in the previous section.

Using a switching schema entails configuring three data sources. If you are using CIM, this configuration is handled for you. See Configuring a Database Switch (page 18) for information.

If you want to set up your database for switching, you must take this into account when creating database tables. Not all of your tables need to be switched, so there is no need to create all tables in both locations. If you are using CIM, database tables are automatically created in the correct schemas, but if you want to create your tables manually, the following scripts should be run for your switching database:

- `priceLists_ddl.sql`
- `product_catalog_ddl.sql`
- `custom_catalog_ddl.sql`
- `promotion_ddl.sql`

The following scripts are run against the (non-switching) production core database:

- `claimable_ddl.sql`
- `commerce_user.sql`
- `dcs_mappers.sql`

- `inventory_ddl.sql`
- `order_ddl.sql`
- `contracts_ddl.sql`
- `organization_ddl.sql`
- `order_markers_ddl.sql`
- `user_giftlist_ddl.sql`
- `user_promotion_ddl.sql`
- `invoice_ddl.sql`

Creating Motorprise Reference Application Tables

You can configure your database to work with Oracle ATG Web Commerce and the Motorprise reference application by running a single script, `motorpriseall_ddl.sql`, from the following directory:

```
<ATG10dir>/MotorpriseJSP/sql/install/database-vendor
```

Alternatively, to configure just the database tables for the Motorprise reference application, run the `motorprise_ddl.sql` script from the same directory. If necessary, you can run these subscripts individually:

Script name	Purpose
`b2b_auth_cc_ddl.sql`	Creates tables for Motorprise profile extensions for authorization
`b2b_custom_catalog_ddl.sql`	Creates tables for Motorprise catalog extensions
`b2b_user_orddet_ddl.sql`	Creates tables for Motorprise profile extensions for order details
`german_catalog_ddl.sql`	Creates tables for the Motorprise product catalog extensions for German content
`japanese_catalog_ddl.sql`	Creates tables for the Motorprise product catalog extensions for Japanese content

Using Oracle ATG Web Commerce with an Oracle Database

To use Oracle ATG Web Commerce with an Oracle database, you need to configure the storage parameters for several tables and configure your Commerce catalog for full text searching. This section covers these configuration tasks:

- Configuring Storage Parameters (page 13)
- Configuring a Catalog for Oracle Full Text Searching (page 13)

Configuring Storage Parameters

The SQL scripts that configure Oracle ATG Web Commerce databases on Oracle do not set storage parameters to control how free database space is allocated. You should spread your tablespaces across several disk drives and disk controllers. The size of the tablespaces needed to store Commerce tables depends on your specific requirements in terms of the catalog items you have, the expected number of user profiles and Web site visitors, and the expected transaction volume.

To allocate space, you need to specify the initial extent and the incremental extent for the tables that are likely to expand significantly. The initial extent parameter limits the amount of space that is reserved initially for a table's data segment. The incremental extent limits the additional space that is reserved when the segment's initial data blocks become full, and more space is required.

To begin, you can set the extent sizes for the tablespaces to be 512K with `pctincrease` equal to 50. If you are loading the Motorprise reference application on an Oracle database, you should configure the storage parameters for the following Commerce tables:

Table Name	Initial Extent	Incremental Extent
`DCS_CATEGORY`	1M	1M
`DCS_PRODUCT`	1M	1M
`DCS_SKU`	1M	1M
`DCS_MEDIA_BIN`	4M	2M
`DSS_SCENARIO_INFO`	4M	2M

Note: These storage parameters are guidelines only. As previously mentioned, the optimal settings for your database may vary depending on the expected number of user profiles, Web site visitors, and catalog items, as well as the expected transaction volume.

To specify a storage parameter, include it in the STORAGE clause of the CREATE TABLE statement for these tables. For additional information about configuring storage parameters, see your Oracle documentation.

Configuring a Catalog for Oracle Full Text Searching

If your product catalog is stored in an Oracle database, you must configure the catalog to properly handle full text searching. There are two main steps involved in this configuration:

1. Set up the proper ConText full text indexes on the appropriate columns in the database.
2. Make sure the `simulateTextSearchQueries` property of each product catalog repository component is set to `false`.

These steps are described in more detail in the sections that follow.

Setting Up the ConText Indexes

The SQL Repository has built-in support for Oracle's ConText full text search engine, which processes queries and returns information based on the content or themes of text stored in a text column of an Oracle database.

To enable full text searching on columns, you must create ConText indexes for the columns. See your Oracle documentation for information about how to do this.

Note: By default, an Oracle database rebuilds a full-text index after each commit. This behavior can cause a full deployment to hang indefinitely. To prevent this, you should configure ConText indexing to occur at regular intervals, using the following format:

```
CREATE INDEX schema-index-name ON schema-table (column)
INDEXTYPE IS CTXSYS.CONTEXT PARAMETERS('SYNC (EVERY "interval-string" ');
```

If you are using the default product catalog, index these tables:

DCS_PRODUCT

DCS_CATEGORY

If you have imported the Motorprise product catalog into Oracle, index these tables:

DCS_PRODUCT

DCS_CATEGORY

DBC_CATEGORY_DE

DBC_PRODUCT_DE

In each of these tables, create a ConText index on the DESCRIPTION, LONG_DESCRIPTION, and DISPLAY_NAME columns.

Configuring the Repository Components for Full Text Searching

To enable a SQL Repository to use full text searching in an Oracle database, the `simulateTextSearchQueries` property of the SQL Repository component must be set to `false`. Make sure this property is set to `false` for any SQL Repository component that connects to an Oracle database.

If you are using the default product catalog, the SQL Repository component for the catalog has the Nucleus address `/atg/commerce/catalog/ProductCatalog`.

Using Oracle ATG Web Commerce with an MSSQL Database

If your Oracle ATG Web Commerce product catalog is stored in a Microsoft SQL Server database, you must configure the database and catalog to properly handle full text searching. There are four steps involved in the configuration process:

1. Set up the proper full text indexes on the appropriate columns in the database.
2. Modify the template definition file.
3. Set the `simulateTextSearchQueries` property of each product catalog repository component to `false`.
4. Configure each search form handler component.

See the subsections that follow for information on completing each step.

Setting Up the MS SQL Full Text Indexes

To enable full text searching on columns, you must create full text indexes for the columns. See your Microsoft SQL Server documentation for specific information on performing this step.

If you are using the default Oracle ATG Web Commerce product catalog, index these tables:

DCS_PRODUCT

DCS_CATEGORY

If you have imported the Motorprise product catalog into MS SQL, index these tables:

DCS_PRODUCT

DCS_CATEGORY

DBC_CATEGORY_DE

DBC_PRODUCT_DE

For each of these tables, create an index for the LONG_DESCRIPTION, DESCRIPTION and DISPLAY_NAME columns.

Modifying the Template Definition File

If you include any full text search queries in the XML template definition file (using the `<query-items>` tag), verify that the queries use the appropriate format for MS SQL Full Text Query. For example:

```
<query-items item-descriptor="product">
   description MATCHES "Ethernet" USING "MSSQL_TSQL"
</query-items>
```

For more information on template definition files, see the *ATG Repository Guide*.

Configuring the Repository Components for Full Text Searching

To enable a SQL repository to work with a full text search engine, the `simulateTextSearchQueries` property of the SQL repository component must be set to `false`.

If you are using the default product catalog, the SQL repository component for the catalog has the Nucleus address `/atg/commerce/catalog/ProductCatalog`.

Configuring the Search Form Handlers

If your sites use components of class `atg.commerce.catalog.custom.CatalogSearchFormHandler` to build search forms for full-text searching, you must configure these components to generate full-text queries in the appropriate form. To do this, each component must be configured as follows:

1. Set the `searchStringFormat` property to `MSSQL_TSQL`.
2. Set the `allowEmptySearch` property to false.

By default, Oracle ATG Web Commerce includes five instances of this class in `/atg/commerce/catalog/`:

- `AdvProductSearch`

- `CatalogSearch`
- `CategorySearch`
- `ProductSearch`
- `ProductTextSearch`

If you've created your own instances of this class, be sure to also set the properties of those components as described above.

Also note that the Motorprise Store Solution Set uses its own instance of `atg.commerce.catalog.custom.CatalogSearchFormHandler`, which is located in Nucleus at `/atg/projects/b2bstore/catalog/SearchCompare`. If you have imported the Motorprise product catalog into MS SQL, configure this component as described above as well.

Transferring Product Catalog and Price List Data Using Copy and Switch

The database Copy and Switch features assist you in moving your product catalog and price lists data from one environment to another, for example, from a staging environment to a production environment. The database Copy feature enables you to copy product catalog and price lists data from one database to another. The database Switch feature enables you to switch the product catalogs and price lists on your Web sites to use a different data source.

Both database Copy and database Switch are Dynamo Application Framework (DAF) features that can be used with any database. However, Oracle ATG Web Commerce provides a user interface for performing a database copy or switch. This user interface requires configuration before you perform each type of update for the first time, as described in these sections:

- Configuring a Database Copy (page 16)
- Performing a Database Copy (page 18)
- Configuring a Database Switch (page 18)
- Performing a Database Switch (page 20)

Note: For additional information about setting up your database servers, see the *ATG Installation and Configuration Guide*. For information about Commerce database tables, see Appendix B, *Oracle ATG Web Commerce Databases* (page 539) in this manual.

Configuring a Database Copy

This section describes the configuration steps you must take before performing a database copy of your product catalog and price lists data. You must complete this process before performing an initial database copy. However, you don't need to complete this process before subsequent database copies as long as the source and destination databases remain the same.

Follow these configuration steps to prepare to copy the product catalog and price lists data from one database to another:

1. Create the destination database and all destination tables.

   Most interactions between the Oracle ATG Web Commerce and a database are done through JDBC, but the `DBCopier` in this process instead uses native commands according to the database manufacturer. It executes a new process and uses the vendor's bulk copy tools to copy the tables from one database to another. This means that the `DBCopier` and, therefore, the JVM running Dynamo must have the proper environment set up as specified by the vendor. See your vendor documentation for these specifications. For information on the API for the specific `DBCopier` you are using, you can refer to the *ATG Platform API Reference*.

2. Verify that the environment is set up correctly. Make sure that all the necessary environmental variables are set as specified by the vendor. Verify that all the necessary drivers or client tools are installed.

3. Create a `DBCopier` component with which to copy the product catalog and price lists data. The class from which you instantiate the `DBCopier` depends on the database you are using. The following are subclasses of `atg.adapter.gsa.DBCopier` and are in package `atg.adapter.gsa`:

   - `BcpDBCopier` (for Microsoft)
   - `DB2DBCopier`
   - `OracleDBCopier`

   For more information on these `DBCopier` classes, refer to the *ATG Platform API Reference*.

   Alternatively, you can use one of the preconfigured `DBCopier` components that are included with Oracle ATG Web Commerce. They are:

   - `ProductCatalogMsSqlDBCopier`

     Note: By default, the `ProductCatalogMsSqlDBCopier.maxTextOrImageSize` property is set to a negative value (-1) in order to force the copier not to use the `-T` option with the `bcp` utility; the `-T` option does not work with MS SQL. For more information on the `maxTextOrImageSize` property, refer to the `BcpDBCopier` class (from which `ProductCatalogMsSqlDBCopier` is instantiated) in the *ATG Platform API Reference*. For more information on the `bcp` utility, refer to your database vendor documentation.

   - `ProductCatalogOracleDBCopier`
   - `ProductCatalogDB2DBCopier`

   These `DBCopier` components are located in Nucleus at `/atg/commerce/jdbc/`.

4. Check that the directory specified in the `directory` property of the `DBCopier` component is available in the file system used by ATG.

5. Verify that the `tables` property of the `DBCopier` component includes all the necessary tables to copy in the correct order. The tables in the destination database will be updated according to the list of tables in this property.

6. Optionally, to obtain information on the activity of the `DBCopier`, you can set the `loggingDebug` property of the `DBCopier` component to true.

7. Create an instance of `atg.droplet.sql.DBCopyFormHandler` to handle the database copy. Alternatively, you can use the preconfigured `DBCopyFormHandler` included with Oracle ATG Web Commerce. It is located in Nucleus at `/atg/commerce/jdbc/ProductCatalogCopierForm`.

8. Perform the database copy from the Copy Product Catalog and Price Lists page, as described in Performing a Database Copy (page 18).

Performing a Database Copy

This section describes the process to copy the contents of one database to another database. It can be used, for example, to copy the updated contents of the product catalog/price lists database on an Administration server to the product catalog/price lists database on a Staging server.

Note: You must have read, write and delete permissions, as well as import and export permissions, to perform the database copy. If you do not have the appropriate permissions, ask your database administrator to copy the database.

To perform a database copy of the product catalog and price lists data:

1. Make sure the configuration file of the `DBCopier` you are using has been updated to account for any database tables you have created or deleted. (See Configuring a Database Copy (page 16) for a code example of a `DBCopier` configuration file.)

2. Access the main Commerce Administration page of the Dynamo Administration UI using the port appropriate for your application server. See the *ATG Installation and Configuration Guide* to find the default port. For example, on JBoss the default URL is:

   `http://hostname:8080/dyn/admin/atg/commerce/admin`

   Note: Your application must include the Dynamo Administration UI in order for you to access this page.

3. Click the Copy Commerce Data link.

   The system displays the Product Catalog and Price Lists Copy page.

4. Enter the Server Name, Username, and Password for the Source database.

5. Enter the Server Name, Username, and Password for the Destination database.

6. Click the Copy button.

   The contents of the Source database are copied into the Destination database. Note that the time it takes to perform the database copy depends on the size of the database.

Configuring a Database Switch

When you perform a database switch, you use a `SwitchingDataSource` to switch between two product catalog/price lists data sources. The `SwitchingDataSource` can switch from one underlying data source to another. For example, the method could switch from a test database to a production database.

This section describes the configuration required before performing a database switch. Use CIM to configure data source components. Once you have configured these components, you do not need to change them unless your data sources change.

The Nucleus components used in this example are located at `/atg/commerce/jdbc`.

Follow these configuration steps to prepare to switch the data source used by the product catalog and price lists repositories:

1. Create the underlying data sources. Alternatively, you can use the data sources provided with the default configuration of Oracle ATG Web Commerce. They are:

   `/atg/commerce/jdbc/ProductCatalogDataSourceA` and `/atg/commerce/jdbc/ProductCatalogDataSourceB`.

These data sources are used as the example data sources in this procedure.

2. Configure the `dataSource` property in each of the data sources.

   The `dataSource` property contains a reference to the data source that stores information for connection to a database. By setting this property for `ProductCatalogDataSourceA` and `ProductCatalogDataSourceB`, you control to what database each data source points.

3. Create a `SwitchingDataSource`. Alternatively, you can use `ProductCatalogSwitchingDataSource`, which is located in Nucleus at `/atg/commerce/jdbc/`. This `SwitchingDataSource` is provided with the default configuration of Oracle ATG Web Commerce.

   A `SwitchingDataSource` is a `DataSource` that can switch between two or more underlying data sources. In a `SwitchingDataSource`, all `DataSource` method calls are passed through to the `DataSource` specified in the `currentDataSource` property, which is defined at run time. For more information on `SwitchingDataSource`, see the *ATG Platform API Reference* and the *ATG Installation and Configuration Guide*.

   The following example shows the configuration for the `ProductCatalogSwitchingDataSource` included with Commerce.

   ```
   $class=atg.service.jdbc.SwitchingDataSource
   #
   # A map from data source names to data sources
   #
   dataSources=\
   DataSourceA=/atg/commerce/jdbc/ProductCatalogDataSourceA,\
   DataSourceB=/atg/commerce/jdbc/ProductCatalogDataSourceB
   #
   # The name of the data source that should be used
   #
   initialDataSourceName=DataSourceA
   ```

   The `dataSources` property of a `SwitchingDataSource` is a mapping of data source names to the Nucleus path of each data source.

   The `initialDataSource` is the data source used.

4. Configure the `ProductCatalog` repository to use the `SwitchingDataSource`. By default, the `ProductCatalog` repository does not use this data source, so you must add the following to the `ProductCatalog.properties` file at `localconfig/atg/commerce/catalog/`:

   ```
   dataSource=/atg/commerce/jdbc/ProductCatalogSwitchingDataSource
   ```

5. Configure the `PriceLists` repository to use the `SwitchingDataSource`. By default, the `PriceLists` repository does not use this data source, so you must add the following to the `PriceLists.properties` file at `localconfig/atg/commerce/pricing/priceLists/`:

   ```
   dataSource=/atg/commerce/jdbc/ProductCatalogSwitchingDataSource
   ```

6. Configure the `switchingDataSource` property of the `ProductCatalogSwitcher` form handler (class `atg.droplet.sql.SwitchDataSourceFormHandler`) to use the `SwitchingDataSource`. The Switch UI uses the `ProductCatalogSwitcher` form handler to perform the database switch; the `switchingDataSource` property specifies the specific `SwitchingDataSource` controlled by the form handler.

7. Perform the database switch as described in Performing a Database Switch (page 20).

Performing a Database Switch

This section describes the process of switching the data source used by the product catalog and price lists on your Web sites.

When you perform a database switch, you must do so on each DRP server instance in an Oracle ATG Web Commerce server cluster. This is necessary because the `SwitchingDataSource` components on separate instances do not synchronize themselves. Consequently, for each DRP server instance, you need to connect to the server's Admin port, access the Commerce Administration UI, and perform the database switch.

Follow these steps to switch the data source currently used by the product catalog and price lists on your Web sites.

1. Access the main Commerce Administration page of the Dynamo Administration UI using the port appropriate for your application server. See the *ATG Installation and Configuration Guide* to find the default port. For example, on JBoss the default URL is:

   `http://hostname:8080/dyn/admin/atg/commerce/admin/en/index.jhtml`

   Note: Your application must include the Dynamo Administration UI module in order for you to view this page.

2. Click the Switch Commerce Data link.

   The system displays the Product Catalog and Price Lists Switch page.

   The names and paths of the available data sources are listed. The data source currently used is displayed below the table.

3. Select the name of the data source to which you want to switch from the drop-down list at the bottom of the page.

4. Click the Prepare for Switch button.

   The system prepares for the switch by sending events to each of the repositories that are using the switching data source. Each repository performs any functions needed to prepare for a switch.

   When preparation is complete, the following message displays at the top of the Switch the Product Catalog's and Price List's Data Sources: Switch page: "Now that you have prepared the data source, you can finish the switch."

5. Click the Switch button to complete the switch.

   When the switch is complete, the page displays the following message: "You have finished switching the data source used by the Product Catalog and the Price Lists." The data source to which you just switched is now listed as the current data source.

Destroying Database Tables for Oracle ATG Web Commerce

This section describes how to destroy the Oracle ATG Web Commerce database tables. When you destroy database tables, you need to destroy them in the opposite order you used for creating them.

To destroy all core Commerce tables, run the `drop_dcs_ddl.sql` script from the following directory:

```
<ATG10dir>/DCS/sql/install/database-vendor
```

The `drop_dcs_ddl.sql` script is derived from the subscripts listed in the table below. If necessary, you can run these subscripts individually from the following directory:

```
<ATG10dir>/DCS/sql/uninstall/database-vendor
```

File Name	Description
`drop_claimable_ddl.sql`	Destroys the schema for the Claimable repository
`drop_commerce_user.sql`	Destroys the tables for the credit card profile extensions
`drop_dcs_mappers.sql`	Destroys the table for handling shopping cart events
`drop_inventory_ddl.sql`	Destroys the tables for the inventory system
`drop_order_ddl.sql`	Destroys the tables for the purchase process
`drop_order_markers_ddl.sql`	Destroys the tables that contain order markers
`drop_priceLists_ddl.sql`	Destroys the tables for the price lists
`drop_product_catalog_ddl.sql`	Destroys the tables for the product catalog
`drop_promotion_ddl.sql`	Destroys the tables for the promotions
`drop_reporting_views.sql`	Destroys the views for reporting
`drop_reporting_views1.sql`	Destroys additional views for reporting
`drop_reporting_views2.sql`	Destroys additional views for reporting
`drop_reporting_views3.sql`	Destroys additional views for reporting
`drop_user_giftlist_ddl.sql`	Destroys the tables for the Giftlist Services
`drop_user_promotion_ddl.sql`	Destroys the tables for the promotions profile extensions
`drop_contracts_ddl.sql`	Destroys the tables for the contracts repository
`drop_invoice_ddl.sql`	Destroys the tables for the invoice repository
`drop_organization_ddl.sql`	Destroys the tables for the organization extensions

Destroying Motorprise Reference Application Tables

You can destroy the database tables for the Oracle ATG Web Commerce platform, Oracle ATG Web Commerce, and the Motorprise reference application by running a single script, `drop_motorpriseall_ddl.sql`, from the following directory:

```
<ATG10dir>/MotorpriseJSP/sql/install/database-vendor
```

Alternatively, to destroy just the database tables for the Motorprise reference application, run the `drop_motorprise_ddl.sql` script from the same directory. If necessary, you can run each subscript individually:

Script name	Purpose
`drop_b2b_auth_cc_ddl.sql`	Destroys tables for the Motorprise profile extensions for authorization
`drop_b2b_custom_catalog_ddl.sql`	Destroys tables for the Motorprise catalog extensions
`drop_b2b_user_orddet_ddl.sql`	Destroys tables for the Motorprise profile extensions for order details
`drop_german_catalog_ddl.sql`	Destroys tables for the Motorprise product catalog extensions for German content
`drop_japanese_catalog_ddl.sql`	Destroys tables for the Motorprise product catalog extensions for Japanese content

3 Using and Extending the Product Catalog

This chapter describes the product catalog definition and explains how to extend it to address your commerce site's requirements. Oracle ATG Web Commerce allows you to set up your product catalog so different customers see different information about the products they view, or different products altogether. The default catalog provides sufficient functionality for many sites. However, you may want to extend or modify this catalog definition to include additional item types or properties.

You create and modify catalog items through the ATG Control Center, as described in the *ATG Commerce Guide to Setting Up a Store*, or through the Oracle ATG Web Commerce Merchandising application, as described in the *ATG Merchandising Guide for Business Users*.

This chapter includes the following sections:

Production and Development Modes

You can run Oracle ATG Web Commerce in either production mode or development mode. The mode that you run in determines how values for catalog-related properties are obtained, which significantly affects your Commerce application's performance.

- **development mode**: Uses *derived* properties so that you can preview a product catalog on a web site while you're making changes without having to run the `CatalogMaintenanceService` (see *Using the Catalog Maintenance System* (page 51) in this guide). Development mode makes updates incrementally so you can preview your changes throughout the development process.

  Development mode overrides the definitions of certain properties in the catalog repository that are normally computed by the batch service, and these properties are derived on-the-fly. Development mode is more resource-intensive than production mode because these properties have to be computed at the time they are referenced, rather than being pre-computed by the batch service.

- **production mode**: Uses *computed* properties. This mode uses properties pre-computed by the `CatalogMaintenanceService`, so performance is superior to development mode.

EAR files are assembled with slight differences for each mode; for information, see the *ATG Platform Programming Guide*.

Product Catalog Repository

Oracle ATG Web Commerce uses a SQL repository to define the product catalog. Before reading this chapter, you should be familiar with the repository as described in the *ATG Repository Guide*.

A catalog repository is similar to any other SQL repository. There are three main parts:

1. The database schema on your SQL database server.
2. The repository component, which is of class `atg.adapter.gsa.GSARepository`.
3. The repository definition file, which is an XML file that defines the relationship between the SQL database and the repository component.

A given user can only have permission to view one catalog. This catalog can be assigned in the catalog property of the user's profile, or derived from the user's `parentOrganization`.

If you are using Oracle ATG Web Commerce's multisite feature, you can assign catalogs to sites. The categories, products, and SKUs in that catalog inherit membership in that site from the catalog.

Catalogs consist of `rootCategories` and `rootSubCatalogs`. A catalog's `rootCategories` combine with the `rootCategories` of its `rootSubCatalogs` to make up the list of the catalog's `allRootCategories`. For example:

Catalog A:
```
rootCategories = category1, category2
```

Catalog B:
```
rootCategories = category3, category4
rootSubCatalogs = CatalogA
```

When a user of Catalog B views the `allRootCategories`, they see all of the root categories of Catalog B (category3 and category4) as well as the root categories of the subcatalogs (category1 and category2) meaning there appear to be four root categories.

The catalog repository component is located at `/atg/commerce/catalog/ProductCatalog`. You can extend the product catalog or create a different catalog structure in several ways:

- To modify the product catalog by adding or removing items or properties, change the standard repository definition file, then use the `startSQLRepository` script to generate the database schema.
- To design your catalog from scratch, write the repository definition file, then use the `startSQLRepository` script to generate the database schema.
- To use an existing database schema, write a repository definition file that corresponds to that schema.

If you replace the standard repository definition file or extend it through XML file combination, you must configure the SQL repository component appropriately.

Catalog Properties

Catalogs allow you to create complicated product structures that are specifically tailored to different users. They form the base of the hierarchy used for navigating your commerce site. Catalogs can contain catalogs and categories.

The following table describes the catalog properties. It uses these abbreviations:

- CCS—CatalogCompletionService (see *Using the Catalog Maintenance System* (page 51) in this guide)
- CMS—CatalogMaintenanceService (see *Using the Catalog Maintenance System* (page 51) in this guide)
- GSA—Generic SQL Adapter (see the *ATG Repository Guide*)
- ACC—ATG Control Center (see the *ATG Commerce Guide to Setting Up a Store*)

Property	How it is set	Description
`allRootCategories`	CCS	Lists of all the root categories in the catalog, including the `allRootCategories` in the `rootSubCatalogs`. This is used for display purposes.
`allRootCategoryIds`	Implicitly set by the CCS	List of the repository IDs of all root categories for the catalog. Read-only. This property refers to the same database table as `allRootCategories`.
`ancestorCategories`	CCS	All categories that are connected to this catalog through its children. This property allows the `CatalogCompletionService` to find all the sub-catalogs of a category that is being added to a catalog or another category. By maintaining this property, the system can query for catalogs that contain the category being added in the `ancestorCategories`, and update those catalogs' ancestor-catalogs properties accordingly.
`ancestorCatalogsAndSelf`	Derived	The combination of `directAncestorCatalogsAndSelf` and `indirectAncestorCatalogs`.

Property	How it is set	Description
`creationDate`	Implicitly set by GSA when catalog is created	Date the catalog was created.
`directAncestorCatalogsAndSelf`	CCS	All the catalogs that use this catalog's `allRootCategories` as their root categories. This property, along with `indirectAncestorCatalogs`, compiles a complete list of all ancestor catalogs within each catalog. The ancestors are divided into direct and indirect lists to make it easier to compute the `allRootCategories` property. A `rootCategory` of a catalog only belongs in the `allRootCategories` list of an ancestor catalog if it is a direct ancestor.
`displayName`	ACC	Name used for the catalog in other Oracle ATG Web Commerce applications. Required.
`id`	Can be set in the ACC	Repository ID for the catalog. If not set, the GSA generates the value.
`indirectAncestorCatalogs`	CCS	All the ancestor catalogs that do not use `allRootCategories` as their root categories. (See `directAncestorCatalogsAndSelf` for further explanation).
`lastModifiedDate`	Implicitly set by GSA when catalog is modified	Date the catalog was last modified.
`parentCategories`	Implicitly through `category.subCatalogs`	The set of categories that have this catalog as a subcatalog. Refers to the `subCatalogs` property of the category, and allows a catalog to find all of its parent categories. Adding a product to another category's child list automatically updates the `parentCategories` property.
`rootCategoryIds`	Implicitly set in the ACC	Repository IDs of top-level categories in the catalog. Read-only. Refers to the same database table as `rootCategories`.
`rootCategories`	ACC	List of the top-level categories in the catalog.
`rootSubCatalogs`	ACC	List of catalogs whose root categories are also root categories of the catalog (for use in combining catalogs).
`siteIds`	CCS	If you are using Oracle ATG Web Commerce's multisite feature, the IDs of those sites to which the catalog belongs.

Property	How it is set	Description
`subCatalogIds`	Optionally implicitly computed by the CCS	Repository IDs of catalogs contained within the catalog, including `rootSubCatalogs` and their `subCatalogs`, and the `subCatalogs` of any categories in the catalog. Read-only. This property refers to the same database table as `subCatalogs`.

Categories and Products

Categories organize your catalog into a hierarchy that provides a navigational framework for your commerce site. A category can contain catalogs, other categories, and products.

For example, you could have a category called Fruit, which contains two products, Apples and Pears, and also contains another category, Citrus Fruit. The Citrus Fruit category could then include products called Lemons, Limes, and Oranges.

A product is a navigational end-point in the catalog. In this example, Oranges is an end-point; it cannot contain other categories or products. However, products do not represent the items that customers actually purchase. The purchased items are called *stock keeping units* (SKUs). A product can have several different SKUs associated with it, representing different varieties, sizes, and colors. For example, if you have a product called Oranges, some of the SKUs associated with it might be Valencia, Navel, and Blood Orange. For more information about SKUs, see SKU Items and SKU Links (page 38) later in this chapter.

The hierarchy defined by products and categories is not rigid. Each category or product can be the child of one or more categories.

This section uses the following abbreviations:

- CCS—CatalogCompletionService (see *Using the Catalog Maintenance System* (page 51) in this guide)
- CMS—CatalogMaintenanceService (see *Using the Catalog Maintenance System* (page 51) in this guide)
- GSA—Generic SQL Adapter (see the *ATG Repository Guide*)
- ACC—ATG Control Center (see the *ATG Commerce Guide to Setting Up a Store*)

Defining Root Categories

With any group of categories, you need to know where to start when navigating. That starting point is called a root category.

Designate a category as a root category by setting the value of the `rootCategories` property of the catalog to include the category you want to be considered the top level of the product catalog. The `allRootCategories` property of the user's catalog specifies all the categories in a catalog's `rootCategories` property.

Note: Root categories of a "root sub catalog" are also considered root categories. For example, if Catalog A has Catalog B as a "Sub catalog at root" then `allRootCategories` of Catalog B are included in the `allRootCategories` of Catalog A.

See the *Catalog Navigation and Searching* chapter of the *ATG Commerce Guide to Setting Up a Store* for an example of using this targeter to find root categories.

Category Properties

The following table describes the category item properties in the catalog.

Property	How it is set	Description
`ancestorCategories`	CMS/CCS	All the categories that can be used to navigate to this category (through `category.childCategories`), regardless of catalog. (`category.childCategories` is a combination of `fixedChildCategories` and `subCatalogsChildCategories`).
`ancestorCategoryIds`	Implicitly set by CMS and CCS	`RepositoryIDs` of the ancestor categories. Read-only. Uses same database table as `ancestorCategories`.
`auxiliaryMedia`	ACC	Additional media to be displayed with this category.
`catalog`	Derived in development mode Set by CMS in production mode.	The catalog that owns this category. This property is no longer used, but remains available for backward compatibility.
`catalogs`	CCS	All the catalogs that include some path to this category; this property is used to determine whether an end user has permission to view this category.
`categoryInfos`	ACC (optional)	Map from `catalogId` to a `categoryInfo`.,
`childCategories`	Derived	List of all categories that are children of this category; a merge of `fixedChildCategories`, `dynamicChildCategories`, and `subCatalogsChildCategories`. Read-only.
`childCategoryGroup`	ACC	Name of the content group that contains the list of `dynamicChildCategories`.
`childProductGroup`	ACC	Name of the content group that contains the list of `dynamicChildProducts`.
`childProducts`	ACC	List of all products that are children of this category; a merge of `fixedChildProducts` and `dynamicChildProducts`. Read-only.
`creationDate`	Implicitly set by GSA	Date this category was created. Read-only.

Property	How it is set	Description
`defaultParentCategory`	CCS	This field is no longer actively used. The `parentCategory` derivation uses this value if the `parentCategoriesForCatalogMap` does not provide a value.
`description`	ACC	Short descriptive text for display with this category.
`displayName`	ACC	Name used for the category on the site. Required.
`dynamicChildCategories`	Derived	List of the categories in the content group specified by the `childCategoryGroup` property. Read-only.
`dynamicChildProducts`	Derived	List of the products in the content group specified by the `childProductGroup` property. Read-only.
`dynamicRelatedCategories`	Derived	List of the categories in the content group specified by the `relatedCategoryGroup` property. Read-only.
`endDate`	ACC	Date this category will no longer be available, if a collection filter is implemented to use this property. See the *ATG Personalization Programming Guide* for information on filtering.
`fixedChildCategories`	ACC	List of child categories of this category. Used by catalog administrator to explicitly set the descendant categories of a category.
`fixedChildProducts`	ACC	List of child products of this category. Used by catalog administrator to explicitly set the descendant products of a category.
`fixedRelatedCategories`	ACC	Static list of categories related to this category.
`id`	ACC or GSA	`RepositoryID` for this category. If this property is not set through the ACC during creation, it is implicitly set by GSA.
`keywords`	ACC	Set of words that can be used in searching for this category.
`largeImage`	ACC	Large image associated with the category.

Property	How it is set	Description
`longDescription`	ACC	Detailed descriptive text for display with this category.
`parentCatalog`	CCS	The parent catalog of this category. Only holds a value if the catalog is a `rootCategory` of the given catalog. This is used in the `ParentCatalog` derivation. This property is no longer used, but remains available for backward compatibility.
`parentCatalogs`	Reference to `catalog.rootCategories`	The set of all catalogs that have this category as a root category.
`parentCategory`	Derived	The parent category of this category. Derived from `parentCategoryForCatalog`; if that is null, derived from `defaultParentCategory`.
`parentCategoriesForCatalog`	CMS or ACC	The parent category for each non-root category. There can be more than possible parent category, in which case the CMS selects one arbitrarily. This is used to derive the value in the `parentCategory` property.
`parentCategoryForCatalog`	Derived	The parent category in the context of the current catalog. Calculated from the `parentCategoriesForCatalog map`.
`relatedCategories`	Derived	List of all categories that are children of this category; this represents a merge of `fixedRelatedCategories`, `dynamicRelatedCategories`, and the `categoryInfo` property `catalogsRelatedCategories`. Read-only.
`relatedCategoryGroup`	ACC	Name of the content group that contains the list of categories that `dynamicRelatedCategories` is set to.
`siteIds`	CMS on production server; CCS on asset management server	If you are using Oracle ATG Web Commerce's multisite feature, the IDs of those sites to which the category belongs.
`smallImage`	ACC	Small image associated with the category.

Property	How it is set	Description
`startDate`	ACC	Date this category becomes available, if a collection filter is implemented to use this property.
`subCatalogs`	ACC	List of catalogs whose root categories will be considered child categories of this category. Used by catalog admin to explicitly set direct descendant catalogs of a category.
`subCatalogsChildCategories`	Derived	The Collective Union of the `allRootCategories` of each catalog in `subCatalogs`. Used to compile the `childCategories` property, which includes the complete list of all descendant categories of a category.
`template`	ACC	JSP used to display this category.
`thumbnailImage`	ACC	Thumbnail image associated with this category.
`type`	N/A	Provided for subclassing purposes; use to indicate if an item belongs to the superclass or a subclass. Read-only.
`version`	Implicitly set by GSA	Integer that is incremented automatically each time the category is updated; used to prevent version conflict.

categoryInfo Properties

`categoryInfo` objects are optional. You can create `categoryInfo` objects through the ACC if you want to keep catalog-specific information for a category.

Property	How it is set	Description
`version`	Implicitly set by GSA	Integer that is incremented automatically each time the `categoryInfo` is updated; used to prevent version conflict.

Product Properties

The following table describes the product item properties in catalog:

Property	How it is set	Description
`ancestorCategories`	CMS	All the categories that you can navigate through (via `category.childCategories` and `category.childProducts`) to this category, regardless of catalog. Used for hierarchical search.
`ancestorCategoryIds`	Implicitly set by CMS	`RepositoryIDs` of the ancestor categories. Read-only. This property uses the same database table as `ancestorCategories`.
`auxiliaryMedia`	ACC	Additional media to be displayed with this product.
`catalogs`	Derived (development only)	In development, this is the Collective Union of catalogs for each category in `parentCategories`. This is not queryable. In production, this value is set by the `CatalogMaintenanceService`. Used to determine if an end user has permission to view this product.
`catalogsRelatedProducts`	derived	Generated Set of related products that are only shown to users of a particular catalog. Read-only.
`childSKUs`	ACC	List of child SKUs of this product. Used by catalog administrator to explicitly set child SKUs of a product.
`creationDate`	Implicitly set by GSA	Date this product was created. Read-only.
`description`	ACC	Short descriptive text for display with this product.
`displayableSkuAttributes`	ACC	List of properties of the product's SKUs that can be displayed by the `DisplaySkuProperties` servlet bean.
`displayName`	ACC	Name used for the product on the site. Required.
`dynamicRelatedProducts`	derived	List of the products in the content group specified by the `relatedProductGroup` property. Read-only.
`endDate`	ACC	Date this product will no longer be available, if a collection filter is implemented to use this property. See the *ATG Personalization Programming Guide* for information on filtering.
`fixedRelatedProducts`	ACC	Static list of products related to this product.
`id`	ACC (optional)	`RepositoryID` for this category. Can be set in ACC upon creation of product. If it is not set through the ACC, it is implicitly set by GSA
`keywords`	ACC	Set of words that can be used in searching for this product.
`largeImage`	ACC	Large image associated with the product.

Property	How it is set	Description
`longDescription`	ACC	Detailed descriptive text for display with this product.
`manufacturer`		
`parentCategories`	Implicit GSA reuse	Reuses the same table as `category.fixedChildCategories`. Therefore adding a product to another category's child list automatically updates the `parentCategories` property.
`parentCategoriesForCatalog`	CMS	The `parentCategory` for each catalog. If this property is null and there is more than one possible parent category, the CMS chooses one arbitrarily. Used to derive the value for the `parentCategory` property.
`parentCategory`	derived	In development, the product's `ParentCategory` is derived by inspecting each category in `parentCategories`. In production, the `CatalogMapDerivation` is used to get the correct parent.
`productInfos`	ACC	Map from `productId` to a `productInfo`.
`relatedProductGroup`	ACC	Name of the content group that contains the list of `dynamicRelatedProducts`.
`relatedProducts`	ACC	List of all products that are children of this product; a merge of `fixedRelatedProducts`, `dynamicRelatedProducts`, and the `productInfo` property `catalogsRelatedProducts`. Read-only.
`siteIds`	CMS or derived	If you are using Oracle ATG Web Commerce's multisite feature, the list of IDs for the sites to which the product belongs. On a production server, this is calculated by the CMS; on an asset management server, it is derived from a union of the `siteIds` of the parent categories of the product.
`smallImage`	ACC	Small image associated with the product.
`startDate`	ACC	Date this product becomes available, if a collection filter is implemented to use this property. See the *ATG Personalization Programming Guide* for information on filtering.
`template`	ACC	JSP used to display this product.
`thumbnailImage`	ACC	Thumbnail image associated with this product.
`type`	Not used out-of-the-box	Provided for subclassing purposes; use to indicate if an item belongs to the superclass or a subclass. Read-only.

Property	How it is set	Description
`version`	Implicitly set by GSA	Integer that is incremented automatically each time the product is updated; used to prevent version conflict. Read-only.

productInfo Properties

`productInfo` objects are optional. You can create `productInfo` objects using the ACC if you wish to keep catalog-specific information for a product.

Property	How it is set	Description
`catalogsRelatedProducts`	ACC	Related products that are only shown to users of the catalog that maps to this `productInfo`.
`version`	Implicitly set by GSA	Integer that is incremented automatically each time the product is updated; used to prevent version conflict. Read-only.

Defining Relationships between Categories and Products

The SQL Repository allows you to define properties that return repository items. The value of one of these items is the object ID. The SQL Repository can transform the ID into the repository item and return the actual object. This mechanism enables you to define the relationship between categories and products in many ways.

Most properties of the SQL Repository return values retrieved from the database. The SQL Repository also provides the ability to compute values of certain properties through Java code. Properties computed dynamically are called user-defined properties. This mechanism is used for the default definitions of several properties of the category and product items, to allow for dynamic definition of item relationships. For more information about user-defined properties, see the *SQL Repository Item Properties* chapter of the *ATG Repository Guide*.

Deriving the childCategories and childProducts Properties

The hierarchical relationships between categories and products are determined by their properties. The `childCategories` and `childProducts` properties of a category define the list of categories and products that are children of the category.

The `childCategories` property is a list of all categories that are children of the category. This is a user-defined property that is computed by the `atg.repository.NotEmptyChooserPropertyDescriptor` class, which sets the value of the property by merging the lists of categories in the `fixedChildCategories` and `dynamicChildCategories` properties. In the repository definition file, the definition of the `childCategories` property looks like this:

```
<property name="childCategories"
  property-type="atg.repository.NotEmptyChooserPropertyDescriptor"
  data-type="list" component-item-type="category" writable="false"
```

```
  queryable="false">
  <attribute name="properties"
    value="fixedChildCategories,dynamicChildCategories"/>
  <attribute name="merge" value="true"/>
</property>
```

This structure enables a page developer to refer to one named property (in this case, `childCategories`) whose value is assembled from different sources.

The `fixedChildCategories` property is an explicit list of categories you specify. The `dynamicChildCategories` property is a user-defined property that is computed by the `atg.repository.GroupMembersPropertyDescriptor` class. This class looks at the `childCategoryGroup` property, which specifies the name of a content group, retrieves the list of categories in that content group, and sets the value of `dynamicChildCategories` to that list.

This mechanism enables you to use business rules to determine the list of child categories. For example, you can create a content group that consists of categories that share a particular attribute, and set the value of `childCategoryGroup` to the name of this content group.

The `childProducts` property of the category is computed in the same way, using the `fixedChildProducts`, `dynamicChildProducts`, and `childProductGroup` properties.

For example, suppose your site has a category called Hats, and some of the hats are available all year, while others are seasonal. You could set the `fixedChildProducts` property of the Hats category to a list of the hats that are available all year. You could also create a content group called Seasonal Hats, which contains a list of hats that changes from season to season, and set the `childProductGroup` property of the Hats category to the name of this content group.

When a user accesses a page that refers to `childProducts`, Oracle ATG Web Commerce computes the current value of `childProducts` as follows:

1. Finds the current set of products in the content group specified in `childProductGroup`, and sets `dynamicChildProperties` to that set of products.
2. Sets `childProducts` to the merge of the set of products in `dynamicChildProducts` and the set of products in `fixedChildProducts`.

For information about creating content groups, see the *ATG Personalization Guide for Business Users*.

Deriving the relatedCategories and relatedProducts Properties

In addition to the `childCategories` property, the category item has a property named `relatedCategories`. This property defines a list of categories that are related to the category, but which do not form a hierarchy with it. Related categories are useful for cross-selling. For example, if the Fruit category has Vegetables as a related category, you can use this relationship to display a link to the Vegetables category on the Fruit page.

The `relatedCategories` property is a user-defined property that is derived in a similar way to the `childCategories` property. The `relatedCategories` property is computed by the `atg.repository.NotEmptyChooserPropertyDescriptor` class, which sets the value of the property by merging the lists of categories in the `fixedRelatedCategories` and `dynamicRelatedCategories` properties. The `dynamicRelatedCategories` property is also a user-defined property, which the `atg.repository.GroupMembersPropertyDescriptor` class computes by retrieving the list of categories in the content group specified by the `relatedCategoryGroup` property.

The product item has a `relatedProducts` property that is computed in the same way, using the `fixedRelatedProducts`, `dynamicRelatedProducts`, and `relatedProductGroup` properties.

Removing the SQL Repository Definitions of User-Defined Properties

If you do not plan to use dynamically related products or categories on your commerce site, you can remove these properties from the repository definition. System performance improves when you simplify the data model to use only fixed relationships. For example, if all categories are explicitly related, you can remove the definitions of the `childCategories`, `dynamicChildCategories`, and `childCategoryGroup` properties, and just use the `fixedChildCategories` property, which you can then rename as `childCategories`.

Specifying Template Pages for Categories and Products

You can display categories and products on your commerce site using JSPs. Rather than requiring you to create a separate JSP for each page on your commerce site, you can to create template pages that fill in the product's properties dynamically. You can then specify which template to use as a property of the category or product. This enables you to display any item without knowing in advance what template the item uses.

For example, the following portion of a JSP uses a `CategoryLookup` servlet bean to retrieve the current category, and a `ForEach` servlet bean to create links to all of the child products of the category. The URL for each link is found by looking at the `template.url` property of the product being linked to.

```
<dsp:droplet name="/atg/commerce/catalog/CategoryLookup">
<dsp:param param="itemId" name="id"/>

<dsp:oparam name="output">
 <dsp:droplet name="/atg/dynamo/droplet/ForEach">
   <dsp:param param="element.childProducts" name="array"/>
   <dsp:oparam name="outputStart">
   <p><b>Child Products:</b>
   </dsp:oparam>
   <dsp:oparam name="output">
   <dsp:getvalueof var="templateURL" param="element.template.url"/>
<dsp:a href="${templateURL}">
   <dsp:valueof param="element.displayName"/>
   <dsp:param param="element.repositoryId" name="itemId"/>
   </dsp:a>
   </dsp:oparam>
 </dsp:droplet>
</dsp:oparam>
</dsp:droplet>
```

For information about the `CategoryLookup` and `ProductLookup` servlet beans, see the *Catalog Navigation and Searching* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Media Properties

Media elements such as images and JSPs can be associated with a category or product. Both item types have a `template` property, as well as three image properties: `thumbnailImage`, `smallImage`, and `largeImage`. In addition to the explicit properties that store media, a Map property called `auxiliaryMedia` allows you to store any number of other media elements. You can also extend the category or product item to include additional properties, as discussed in the Extending the Category and Product Item Types (page 37) section.

Associating Products with SKUs

The `childSKUs` property of a product is a list of all the SKUs that are children of the product. A product called Apples might have several different SKUs that represent different varieties of apples, such as McIntosh, Delicious, and Granny Smith.

The product item also has a property named `displayableSkuAttributes`, which you can use to specify a list of the SKU properties that can be displayed using the `DisplaySkuProperties` servlet bean. For more information about this servlet bean, see the *ATG Commerce Guide to Setting Up a Store*.

Extending the Category and Product Item Types

By default, there is only one type of category and one type of product in the catalog. These item types are sufficient for many commerce sites. However, you may find you need to extend the catalog by creating additional properties for these item types, or by creating additional item types.

To create additional properties for an item type, you can either add new tables to the schema or modify the database schema to add columns to the database tables for that item type.

Note: Creating separate tables reduces the chance of complications during future Oracle ATG Web Commerce upgrades; however, it can negatively affect performance. If you have a large product catalog, performance may be of more importance.

After altering your database, make the corresponding additions to the repository definition file. For greater flexibility, you can create new item types. There are two ways to create a new item type:

- Create an entirely new type, whose definition is independent of existing types.
- Create an item type that is a sub-type of an existing type. A sub-type inherits the properties from its parent item type, and can include additional properties that are unique to the sub-type.

The default category and product item definitions include an enumerated property named `type`, which you can use to create item sub-types:

```
<property name="type" data-type="enumerated" column-name="product_type"
      writable="false" hidden="true"> </property>
```

To create a product sub-type, add an option value to the product `type` enumeration:

```
<property name="type" data-type="enumerated" column-
  name="product_type" writable="false" hidden="true">
  <option value="option1"/>
  <option value="option2"/>
</property>
```

Then add a new item-descriptor to the repository definition file, and set its super-type attribute to the name of the parent item type:

```
<item-descriptor name="item-name" super-type="type" sub-type-
          value="option1">
          <!-- properties -->
```

```
        </item-descriptor>
```

Then add your new sub-type to the appropriate property of the `/atg/commerce/CatalogTools` component:

```
catalogFolderItemTypes=catalogFolder
catalogItemTypes=catalog
productItemTypes=product
categoryItemTypes=category
SKUItemTypes=sku,configurableSku
```

For example, if you create a new subtype called `customCategory`, add the following to the `/atg/commerce/CatalogTools` component::

```
categoryItemTypes+=customCategory
```

The new item type inherits all of the defined properties of the parent item type, and can add properties not defined for the parent item type. Inheritance is a very convenient mechanism for creating new item types, especially for creating types that are closely related to each other. However, you should avoid using too many levels of inheritance. Queries against items whose properties span multiple sub-types may require joins of all of the tables in the hierarchy. If you use these kinds of queries, keep in mind that performance decreases as the number of tables joined increases.

For more information, see the *Item Descriptor Inheritance* section of the *SQL Repository Data Models* chapter of the *ATG Repository Guide*.

SKU Items and SKU Links

A product is a navigational end-point in the catalog. However, customers do not actually purchase the product; they purchase a SKU (stock keeping unit). A product can have several different SKUs associated with it, representing varieties, sizes, and colors.

The properties of a SKU are used for display purposes, similar to products and category properties. The properties are also used to integrate with other Oracle ATG Web Commerce systems, such as pricing and fulfillment.

A SKU usually represents an indivisible unit that can be purchased. However, the catalog includes a SKU link item type that you can use to create *SKU bundles*, which are virtual SKUs that are composed of several other SKUs. Bundles allow the product catalog to offer a SKU that can be purchased as a single item, although it is treated as multiple items in fulfillment.

You can also create SKUs as configurable if they have components that might vary depending on customer preferences, such as with computers, which could include different memory or hard drives, or a car with optional features. See the Configurable SKUs (page 43) section of this chapter.

SKU Properties

The following table describes the SKU item properties in the catalog:

Property	How it is set	Description
`auxiliaryMedia`	ACC	Additional media to be displayed with the SKU.
`bundleLinks`	ACC	List of SKU links that make up the SKU bundle; if null, SKU is not a bundle.
`catalogs`	CMS or Derived	In development this is the Collective Union of catalogs for each product in `parentProducts`. This is not queryable. In production, this value is set by the `CatalogMaintenanceService`. Used to determine if an end user has permission to view this SKU.
`catalogsReplacementProducts`	Derived	Replacement products that are only shown to users of a particular catalog. Read-only.
`creationDate`	Implicitly set by GSA	Date the SKU is created. Read-only.
`description`	ACC	Short descriptive text for display with the SKU.
`displayName`	ACC	Name used for the SKU on the site. Required.
`dynamicAttributes`	ACC	Additional attributes of the SKU.
`endDate`	ACC	Date the SKU is no longer available, if a collection filter is implemented to use this property. See the *ATG Personalization Programming Guide* for information on filtering.
`fixedReplacementProducts`	ACC	Static list of products related to this SKU.
`fulfiller`	ACC	Fulfiller who will ship the item.
`id`	ACC	Repository ID for this SKU. Can be set in ACC upon item creation, otherwise implicitly set by GSA.
`largeImage`	ACC	Large image associated with the SKU.
`listPrice`	ACC	Default price of the SKU before any discounts or promotions.
`manufacturer_part_number`		
`onSale`	ACC	Boolean property that indicates if the item is on sale.
`parentProducts`	implicit GSA reuse	Reuses the same table as `product.childSkus`. Therefore adding a SKU to a product's child list will automatically update the `parentProducts` property.
`replacementProducts`	derived	Products to suggest as replacements if the item is out of stock. Read-only.
`salePrice`	ACC	Price of the SKU if `onSale` property is `true`.

Property	How it is set	Description
`siteIds`	CMS or derived	If you are using Oracle ATG Web Commerce's multisite feature, the IDs of the sites to which the SKU belongs. On the production server, this is calculated by the CMS; on the asset management server, it is derived from the union of the `siteIds` of the SKU's parent products.
`skuInfos`	ACC	Map from `SKU` to a `SKUInfo`.
`smallImage`	ACC	Small image associated with the SKU.
`startDate`	ACC	Date on which the `SKU` is available, if a collection filter is implemented to use this property.
`template`	ACC	JSP template used to display the SKU.
`thumbnailImage`	ACC	Thumbnail image associated with the SKU.
`type`	Not used out-of-the-box	Provided for subclassing purposes; use to indicate if an item belongs to the superclass or a subclass. Read-only.
`version`	Implicitly set by GSA	Integer that is incremented automatically each time the SKU is updated; used to prevent version conflict. Read-only.
`wholesalePrice`	ACC	Wholesale price of the SKU.

SKUInfo Properties

You can create `SkuInfo` objects using the ACC if you wish to keep catalog-specific information for a SKU.

Property	How it is set	Description
`catalogsReplacementProducts`	ACC	These are replacement products that are only shown to users of the catalog that maps to this `SKUInfo`.
`version`	Implicitly set by GSA	Integer that is incremented automatically each time the `SKUInfo` is updated; used to prevent version conflict. Read-only.

SKU Link Properties

The following table describes the SKU link properties in the product catalog:

Property	Description
`creationDate`	Date this SKU link was created.
`description`	Short descriptive text for display with this SKU link.
`displayName`	Name used for the SKU link on the site.
`endDate`	Date this SKU link will no longer be available, if a collection filter is implemented to use this property.
`item`	SKU that is being bundled.
`quantity`	Quantity of the SKU specified by the `item` property.
`startDate`	Date on which this SKU link becomes available, if a collection filter is implemented to use this property.
`version`	Integer that is incremented automatically each time the SKU link is updated; used to prevent version conflict.

Using SKU Media Properties

The SKU item has the same set of media properties that categories and products have: `template`, `thumbnailImage`, `smallImage`, `largeImage`, and `auxiliaryMedia`. Your sites may not need all of these properties. For example, at most commerce sites, each SKU would not be displayed in its own template page. More commonly, a product's template page displays all of the child SKUs of the product.

Categories, products, and SKUs all have the same set of media properties in order to give you as much flexibility as possible. Depending on how your sites are organized, you might want to associate certain media items with products rather than SKUs, or vice versa. For example, if SKUs are differentiated by a visible characteristic such as color, you might want to have different images for each SKU, rather than having a single set of images associated with the parent product. If each product has only one SKU, you could even change the repository definition to remove the product item type entirely.

Using SKU Price Properties

The SKU item has four price properties: `wholesalePrice`, `listPrice`, `onSale`, and `salePrice`. These list and sale price calculators use these properties in the Pricing Engine.

`ListPrice` is required, but the `wholesalePrice` and `salePrice` can be undefined. `WholesalePrice` is not used by the pricing calculators. When the sale price is defined and `onSale` is set to `true`, the pricing calculators assume the SKU is on sale and adjust the price. Placing the prices at the SKU level allows each individual SKU to have its own price, but adds complexity to pricing administration. If your product catalog does not need to price each SKU separately, then the `listPrice`, `salePrice`, and `onSale` properties can be moved to the product item type to simplify maintenance. If the price properties are moved, the list and sale price calculators will require a small change in configuration. For more information, see the *Commerce Pricing Calculators* (page 139) chapter of this manual.

Using the SKU Fulfiller Property

The fulfillment engine uses the fulfiller property to determine how to manage the fulfillment of the purchase. This property is an enumerated value. By default, it defines a `HardgoodFulfiller`. For more information, see the *Configuring the Order Fulfillment Framework* (page 413) chapter.

Creating SKU Bundles

The `bundleLinks` property designates a SKU as a bundle, rather than an individual item. The `bundleLinks` property is a list whose elements are of another item type called a SKU link. If the SKU is not a bundle, then `bundleLinks` is null.

Each SKU link includes a SKU and a quantity. For example, a SKU link might represent three red shirts. A SKU bundle can include any number of SKU links. For example, a SKU bundle might consist of two SKU links, one that represents three red shirts and another that represents one black hat. Note that the `bundleLinks` property of the SKU item holds only SKU links, not SKU items. However, you can create a SKU link whose quantity is 1, and include that SKU link in the `bundleLinks` list.

When the fulfillment system recognizes that a SKU is a bundle, it fulfills the purchase by performing the fulfillment of each quantity of items defined in the SKU link. For more information, see the *Configuring the Order Fulfillment Framework* (page 413) chapter.

Extending the SKU Item Type

The base SKU definition does not include any configuration properties such as size, color, etc. Depending on the requirements of your sites, you may be able to differentiate the SKUs for a product through the description properties of the SKUs. For example, if the SKUs are differentiated only by color, the description of each SKU could mention the color. However, this approach can become awkward if there are several different configuration variables.

You may want to add properties to the SKU item type, or create sub-types of the SKU item type that include additional properties. If you create a sub-type, you must add it to the `SKUItemTypes` property of the `/atg/commerce/catalog/CatalogTools` component in order for it to be available to the PMDL rules used in promotions. See Extending the Category and Product Item Types (page 37) in this guide for information about extending catalog item types. See the *ATG Commerce Guide to Setting Up a Store* and the *ATG Merchandising Guide for Business Users* for information on promotions.

An alternate approach, which does not require modifying the SKU item definition, is to store configuration information in the `dynamicAttributes` property. The property is a Map that stores a key/value pair for each attribute you define. For example, you could specify the `dynamicAttributes` property of a SKU as `{color=red, size=small}`. This mechanism is very flexible, because it allows each SKU to have its own set of configuration attributes.

One disadvantage of this approach is that data is stored less efficiently than if you explicitly add configuration properties to the definition of the SKU item. If you add properties, each property corresponds to a column in a table, so each SKU can be represented by a single row in the table. If you use `dynamicAttributes`, each SKU is represented by one row per attribute. So if `dynamicAttributes` specifies three attributes (e.g., size, color, and length), the SKU will be represented by three rows in a table. The additional rows can have a negative effect on performance. Therefore, if you need to define a large number of attributes, explicitly adding properties to the SKU item definition is a better approach.

Configurable SKUs

A configurable SKU holds other SKUs, in a different way than linked SKUs do. An example of a configurable item is a computer; customers can order them with varying amounts of memory, hard drive types, video cards, etc. Configurable items consist of a "base SKU" and a number of optional subSKUs for the user to choose among.

The following repository items are used when working with configurable SKUs:

Item	Description
`configurableSku`	Base SKU for the product, to which other products can be added as options, such as a computer.
`configurableProperty`	A category of subSKUs, such as Memory or Modems. The `configurableProperty` holds the list of `configurationOptions`.
`configurationOption`	The SKU representing the actual product option, such as a particular model of hard drive.
`foreignCatalog`	An external system with which Oracle ATG Web Commerce communicates.

Catalog Folders

You can use catalog folders to organize catalogs. Catalog folders are used only in the administrator user interface, not in the commerce site itself. The following table describes the folder item properties in the product catalog:

Property	Description
`creationDate`	Date this folder was created.
`description`	Short descriptive text for display with this folder.
`endDate`	Date this folder will no longer be available, if a collection filter is implemented to use this property. See the *ATG Personalization Programming Guide* for information about filtering.
`name`	File name of the folder.
`parentFolder`	Folder that contains this folder; if null, this folder is a root folder.
`path`	Complete pathname of the folder.
`startDate`	Date this folder becomes available, if a collection filter is implemented to use this property.

Property	Description
`siteIds`	If you are using Oracle ATG Web Commerce's multisite feature, the IDs of the sites to which the catalog folder belongs.
`version`	Integer that is incremented automatically each time the folder is updated; used to prevent version conflict.

Folders and Media Items

As described in the *SQL Content Repositories* chapter of the *ATG Repository Guide*, a SQL Repository can be configured as a content repository. A content repository is composed of folder and content repository items. The product catalog defines folder and media item types that correspond to these two parts of the repository.

You can use folders to organize media elements. Folders are used only in the administrator user interface, not in the commerce site itself. Both the folder and media items define several administrative properties: `version`, `creationDate`, `startDate`, `endDate`, and `description`. These properties are used as an aid to catalog administrators, not intended for display on the site.

The `media` item is similar to an abstract Java object; it cannot be instantiated itself. The item type is marked as abstract by tagging it as expert in the repository definition file. As part of this abstract definition, two properties are defined: `data` and `url`. These properties are intended to be overridden in the sub-types.

The `media` item includes a property named `type` that is used to specify the media sub-type of the item. The product catalog includes three sub-types:

- media-external: This item type references a piece of content that is stored outside the database. The content can be either be either a binary file or a text file.
- media-internal-binary: This item type can be used to store binary objects (such as images) in the catalog database.
- media-internal-text: This item type can be used to store text files (such as JSPs) in the catalog database.

For information about uploading media elements to your product catalog, see *ATG Commerce Catalog Administration* of the *ATG Commerce Guide to Setting Up a Store*.

Folder Properties

The following table describes the folder item properties in the product catalog:

Property	Description
`creationDate`	Date this folder was created.
`description`	Short descriptive text for display with this folder.

Property	Description
`endDate`	Date this folder will no longer be available, if a collection filter is implemented to use this property. See the *ATG Personalization Programming Guide* for information about filtering.
`name`	File name of the folder.
`parentFolder`	Folder that contains this folder; if null, this folder is a root folder.
`path`	Complete pathname of the folder.
`startDate`	Date this folder becomes available, if a collection filter is implemented to use this property.
`version`	Integer that is incremented automatically each time the folder is updated; used to prevent version conflict.

Media Item Properties

The following table describes the media item properties in the product catalog:

Property	Description
`url`	Relative URL that can be used in a JSP to access the media item.
`data`	Defines whether a media item is binary or text.
`mimeType`	A user-defined property that returns the content type value by examining the `url` property.
`name`	File name for this media item.
`path`	Complete path for this media item.
`parentFolder`	Folder that contains this media item.
`startDate`	Date this media item becomes available, if a collection filter is implemented to use this property.
`endDate`	Date this media item will no longer be available. if a collection filter is implemented to use this property.
`description`	Short descriptive text for display with this media item.

Using Media-External Properties

In the media-external item type, the `data` property is a user-defined property computed by the `atg.repository.FilePropertyDescriptor` class. If the `url` property is a relative URL, then `data` is a `java.io.File` object, which references the file.

As part of defining a `ContentRepositoryItem`, the item definition must supply a length and a last modified value. In a media-external item, these values are automatically extracted from the `File` object returned from `data`.

Using Media-Internal Properties

The media-internal sub-types differ only in the way the `data` property is defined. The `data` property is either `binary` (for media-internal-binary items) or `text` (for media-internal-text items).

The media-internal item types define their own `length` and `lastModified` properties. The `mimeType` property is a user-defined property that is computed by the `MimeTyperPropertyDescriptor` class, which determines the value from the `name` property.

The `url` property of the media-internal item types is a user-defined property computed by the `atg.distributor.DistributorPropertyDescriptor` class. When a customer accesses a media-internal-binary or media-internal-text item, the `/atg/commerce/catalog/ContentDistributorPool` service extracts the content from the database and generates a URL to make the item accessible to a web browser.

For more information about configuring content distributors, see the *Content Distribution* chapter of the *ATG Platform Programming Guide*.

Internationalizing the Product Catalog

If you want to internationalize your product catalog, you can extend the Commerce repository to support translated versions of some properties. This section outlines the internationalization strategy that is implemented as part of the Oracle ATG Web Commerce Reference Store. This "best practice" offers the following benefits over other approaches:

- Applications can switch between international and non-international modes without requiring any JSP page changes. The same property names are used in the JSP page code and each repository derives the appropriate language as necessary.
- No database schema changes are required to add additional languages.

To extend your repository to include internationalized values, follow these steps:

1. Decide which properties of which item types you want to translate. This example uses properties of the SKU item type, but you may need to translate product, category, and even catalog properties, depending on your catalog structure.
2. For each item type you are internationalizing, add new properties corresponding to that item type's translatable properties.

   For example, you want to provide internationalized versions of four `SKU` properties: `displayName`, `description`, `size`, and `color`. You would add four new properties (`displayNameDefault`, `descriptionDefault`, `sizeDefault`, and `colorDefault`) to the `SKU` item descriptor.

   The new properties refer to the original properties' database columns and represent the default text for the properties (thereby allowing us to redefine the original properties as derived properties).

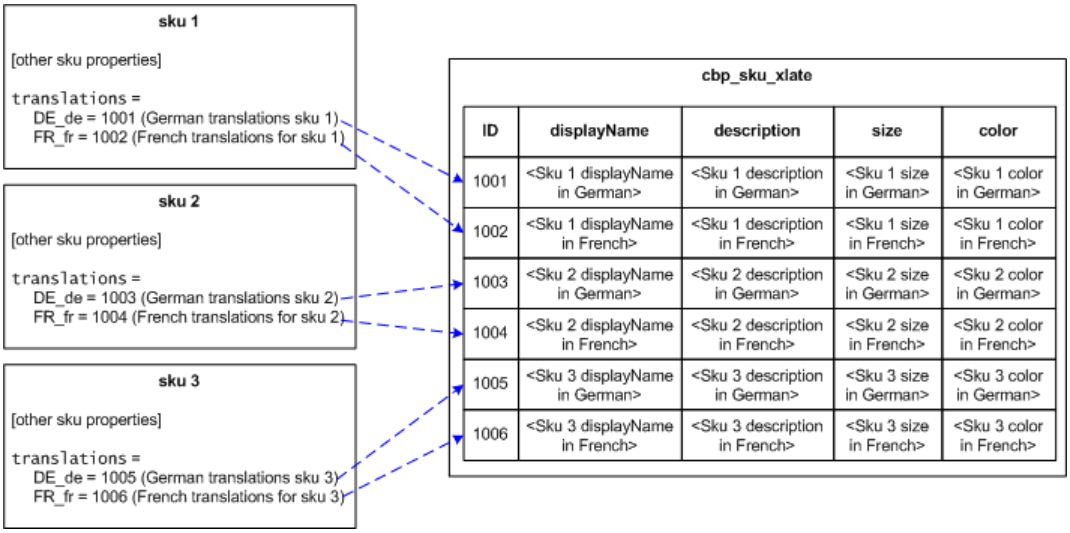
   These four properties refer to the `display_name`, `description`, `sku_size`, and `color` columns, where default-language text for the content are stored.

3. Add another property to the item type. This property (with a name such as `translations`) is a map whose key is a `locale` and whose value is an item of type *`baseType`*`Translation`, described below. Note that the `locale` key does not have to be a fully qualified locale.

4. Define a set of helper item types for all existing item types that have translatable properties. Our example uses the naming convention *`baseType`*`Translation`, where *`baseType`* refers to an existing item type; for example, create a `skuTranslation` item type to correspond with the `sku` item type, a `productTranslation` item type for the `product` item type, and so on.

   `baseType``Translation` items function as containers for locale-specific content. As such, each *`baseType`*`Translation` item type has properties that correspond to the translatable properties of its base item type. In our example, we have selected four SKU properties for translation (`displayName`, `description`, `size`, and `color`). Therefore, the `skuTranslation` item also has four properties for `displayname`, `description`, `size`, and `color`. Each *`baseType`*`Translation` item type has its own table in the database, where each row represents a single *`basetype`*`Translation` item with a unique ID. For example, the `cbp_sku_xlate` table contains all the `skuTranslation` items, the `cbp_prd_xlate` table contains all the `productTranslation` items, and so on.

Every base item (SKU, product, category) is tied, through its `translations` property, to one or more *`baseType`*`Translation` items (one for each locale, with the exception of the default locale). The following example shows three `sku` items and six corresponding `skuTranslation` items which contain translated content for two locales, French and German.



To create the relationships that connect a base item to its *`baseType`*`Translation` items, change the definitions of the translatable properties in the existing item types. The new definitions should specify that each translatable property is a derived property whose value is determined as follows:

- Use the current locale to look up a corresponding *`baseType`*`Translation` item in the `translations` property map. The property derivation attempts to find a best match. First, it searches the `locale` keys for a match on the entire locale with a variant, then it searches for a match on the locale without a variant, and finally it searches on just the language code.

- If a *`baseType`*`Translation` item exists for the current locale, use its value for the property.

- If a *`baseType`*`Translation` item doesn't exist for the current locale, or its value for the property is null, use the *`translatableProperty`*`Default` value instead.

The Oracle ATG Web Commerce Reference Store code uses the `atg.repository.dp.LanguageTranslation` class to implement the derivation.

The following example shows how Commerce Reference Store derives the `sku.displayName` property for a store that has English (default), German, and French translations:

1. `giftListShop.jsp` requests the `sku.displayName` property for a SKU and determines the locale, which in this example is `DE_de`.
2. The catalog repository finds the corresponding `skuTranslation` item using the `translations` property map. Based on the locale, the repository determines that it should reference the German `skuTranslation` item.
3. The catalog repository returns the `displayName` property from the German `skuTranslation` item.
4. If no `skuTranslation` item exists for the locale, it returns the value from `displayNameDefault`.

Catalog Security

Securing a catalog allows certain users to view and edit a catalog while preventing other users from doing so. Oracle ATG Web Commerce implements a security policy for catalogs based on the secured repositories feature (for more information, see the *Secured Repositories* chapter in the *ATG Repository Guide*).

In the catalog security policy, an access control list (ACL) is stored for each individual item (catalog, product, SKU, media) except for category; the ACL of a category is the same as the ACL of the catalog that contains it. The ACL contains lists of users or groups of users, and the permissions they have.

Note: If you are using Oracle ATG Web Commerce Business Intelligence for reporting with secured catalogs, in order for the ProductCatalog logs to be created during deployment, you must create a `server_name/localconfig/atg/reporting/datacollection/commerce/ProductCatalogDeploymentListener.properties` file with the following contents:

```
repositoryPathAliasMap=\
/atg/commerce/catalog/SecureProductCatalog=/atg/commerce/catalog/
ProductCatalog
```

```
sourceRepositoryPathToItemDecriptorNames+=\
/atg/commerce/catalog/SecureProductCatalog=category;product;sku;promotion
```

See the *Preparing to Use Commerce Reporting* (page 489) chapter in this guide for information on data logging.

Importing Product Catalog Content

You may want to maintain your catalogs in an external system and use that content as a basis for your product catalogs, which are organized into repository data sets. The way you approach exporting this content depends

largely on the system that currently holds your content and the data itself. Consult your third-party system documentation for exporting instructions. Oracle ATG Web Commerce includes two importing tools:

- The `startSQLRepository` script imports content formatted in an XML file into content or SQL repositories. See the *startSQLRepository script and Template Parser* section in the *ATG Repository Guide*.
- The Repository Loader is a module that conducts single or scheduled imports of XML, HTML, binary, and file system files into content or SQL repositories. See the *Repository Loader* chapter of the *ATG Repository Guide* for information.

You can also write your own import code. Keep in mind that the XML files parsed by the `startSQLRepository` script are easier to write than their Repository Loader counterparts, however, running the Repository Loader uses less memory than the `startSQLRepository` script.

It's a good idea to familiarize yourself with repositories in general by reading through the *Introduction* and *Repository API* chapters of the *ATG Repository Guide* as well as the documentation on the commerce repositories available to you. The commerce repositories are described in the *Using and Extending the Product Catalog* (page 23) chapter of this guide. Learning about the repositories you'll use will help you understand the format required by the Oracle ATG Web Commerce importing tools, and planning your data organization strategy will minimize the amount of manual tweaking you'll need to do later. The Commerce repositories are extensible, so you have the option of changing them if they don't offer the properties or item types your product catalog requires.

Once you generate the export files, import your content into Commerce using the tool you prefer.

Assigning a Catalog to a User

In order for users to view a catalog, they must have that catalog assigned to their profile. The catalog assumes the existence of a `catalog` property for the `user` repository item. The `catalog` property is a derived property. You can set it in either the user's `myCatalog` property or the `catalog` property of the `contract` for the user's `parentOrganization`. If `myCatalog` is set, the parent organization's setting is ignored.

Oracle ATG Web Commerce adds `CatalogProfilePropertySetter` and `PriceListProfilePropertySetter` components to the `profilePropertySetters` property of the `/atg/dynamo/servlet/dafpipeline/ProfilePropertyServlet` component in the DAF servlet pipeline:

```
profilePropertySetters+=/atg/userprofiling/CatalogProfilePropertySetter,\
                        /atg/userprofiling/PriceListProfilePropertySetter
```

For the profile's `catalog` property, the `CatalogProfilePropertySetter` calls the `determineCatalog` method of the `/atg/commerce/catalog/CatalogTools` component (class `atg.commerce.catalog.custom.CustomCatalogTools`). This method invokes the `/atg/commerce/util/ContextValueRetriever` component, which is the primary mechanism for identifying the catalog to assign so that the appropriate items can be displayed. See ContextValueRetriever Class (page 49) for more information.

ContextValueRetriever Class

The class for the `ContextValueRetriever` component is `atg.commerce.util.ContextValueRetriever`, which holds most of the logic for determining which catalog to assign to each user. It performs a parallel

function for price lists – see PriceListManager (page 214) for more information. It has one property, `useProfile`, which is a boolean that defaults to false. The main method, `retrieveValue`, goes through the following steps. The method does not continue to the next step if it finds a non-null value.

1. It calls the `shouldUseProfile` method (see below). If this method returns true, it retrieves the requested property from the profile.

2. If a site was provided, it retrieves the requested property from the site. See Assigning Price Lists and Catalogs in a Multisite Configuration (page 70) for more information.

If `ContextValueRetriever` returns null, `CustomCatalogTools.determineCatalog` picks up the default catalog and returns that to the `CatalogProfilePropertySetter`.

The out-of-the-box implementation of the `shouldUseProfile` method simply returns the value of the `useProfile` property. By default this property is set to false, which is appropriate for most multisite environments. The false value means that anything that might already be set in the profile is ignored, and values are retrieved instead from the current site or the global (`CatalogTools`) default. If you assign catalogs or price lists via your own pipeline servlet or a scenario, set `useProfile` to true to prevent your profile settings from being overridden with the global defaults.

If your business requires different choices for different customers, you can override `shouldUseProfile`. For example, assume some special customers (for example employees and sales reps) have values pre-set in their profiles, while most customers should get site defaults. The `shouldUseProfile` method can look at any profile property to decide if a customer is special or not and return true for the special customers and false for everyone else. In addition, if you want to add another source for catalogs or price lists, beyond profiles and sites, you can override the `retrieveValue` method.

Note that Oracle ATG Web Commerce provides a single component that handles both catalogs and price lists. This configuration assumes you want to apply the same logic for both types of values, which is common. If you want to apply different logic for catalogs and price lists, you can configure a second Nucleus instance of the `ContextValueRetriever` component, change its configuration or implementation class, and reconfigure either `CatalogTools` or `PriceListManager` to refer to the new instance.

4 Using the Catalog Maintenance System

Relationships among items in a catalog can be very complex. Any time you make changes to the catalog, those relationships might change. Maintaining these relationships is vital in order to allow customers to navigate and search your catalogs. Oracle ATG Web Commerce catalogs use several batch and dynamic services to perform catalog updates and to verify catalog relationships. These services are collectively referred to as the Catalog Maintenance System (CMS).

The CMS updates property values that enable navigation and hierarchical search (see the *Catalog Navigation and Searching* chapter of the *ATG Commerce Guide to Setting Up a Store*). It also verifies the relationships of catalog repository items and properties. These services can be run either on demand or in scheduled mode, and each service performs functions that can be selectively executed using the function names.

This chapter includes the following sections:

Important: If you add items to catalog folders, the catalog maintenance services propagate site membership from those folders to the items added. If you remove items from those folders, however, they remain linked to their sites; the CMS has no way to know if the site association existed before the catalog folder relationship.

Batch Services

These services facilitate catalog updates in batch mode. All batch services can be run on demand or in scheduled mode. The batch mode services are:

- CatalogMaintenanceService (page 53), which includes the following services:
  - AncestorGeneratorService (page 54)
  - CatalogVerificationService (page 55)
- CatalogUpdateService (page 56)

Each service keeps in memory a history of all information, warning and error level messages. The messages are exposed through a set of API methods: `getWarningMessages`, `getErrorMessages`, `getInfoMessages` and

getCurrentMessage. They can be viewed using the View Status option on the Commerce Administration Page. For more information see Running Catalog Maintenance Services (page 59).

Each service creates a global lock at the start of execution and releases it upon completion. This prevents services that use the same lock name from executing simultaneously on the same server, or other servers in the cluster.

All services have the following configurable properties, in addition to their own unique properties:

Property	Description
schedule	Schedule on which the service will execute. If this property is valued, the service will schedule itself accordingly when it is first instantiated.
availableFunctions	A list of function names that the service can perform. Function names identify specific processes that can be performed by the service, such as AGS_GENCATALOGS to generate ancestor catalogs.
functionsToPerformByDefault	This property configures a default set of function names that are executed if none are explicitly provided, which may be the case if the services is running in schedule mode, or through the Java API. It can include any of the functions exposed by the availableFunctions property.
saveMessages	If true, the info, error, and warning messages from the last execution of the service are retained in memory. These messages are used for the maintenance log display on the Dynamo Admin UI. The default is true.
maxItemsPerTransaction	The number of repository items that are updated in a single transaction. This can be used in the case of very large catalogs to spread updates across several transactions.
jobName	The scheduler job name used when the service is scheduled.
jobDescription	The scheduled job description.
lockTimeOut	Time in milliseconds before a timeout when acquiring the global lock.
lockName	The name used for the global lock.

The AncestorGeneratorService, CatalogVerificationService, and CatalogUpdateService all include a catalogProperties property. The catalogProperties property points to the /atg/commerce/catalog/custom/CatalogProperties component, which includes the following properties:

Property	Description
`categoryItemName`	The name of the item type that functions as a category in the catalog. If the catalog has multiple category item types related by inheritance, set this property to the name of the parent type; the service will generate ancestor categories for the subtypes as well as the parent type. Default: `category`
`productItemName`	The name of the item type that functions as a product in the catalog. If the catalog has multiple product item types related by inheritance, set this property to the name of the parent type; the service will generate ancestor categories for the subtypes as well as the parent type. Default: `product`
`ancestorCategoriesPropertyName`	The name of the property used to store the Set of ancestor categories of the item; this must be a property of the item types specified by `categoryItemName` and `productItemName`. Default: `ancestorCategories`
`childCategoriesPropertyName`	The name of the property used to store the Set of categories that the item is the parent category of; this must be a property of the item type specified by `categoryItemName`. Default: `fixedChildCategories`
`childProductsPropertyName`	The name of the property used to store the Set of products that the item is the parent category of; this must be a property of the item type specified by `categoryItemName`. Default: `fixedChildProducts`
`catalogIds`	An array of the IDs of the `Repository` objects that make up the catalog. If your sites use only a single repository for its product catalog, this property can be null. Default: null
`includDynamicChildren`	If true, `AncestorGeneratorService` generates properties for both fixed and dynamic children; if false, it generates properties only for fixed children. The default is false.

CatalogMaintenanceService

Component: `/atg/commerce/catalog/CatalogMaintenanceService`

The `CatalogMaintenanceService` component is a container for the `AncestorGeneratorService` and the `CatalogVerificationService` (note, however, that the `CatalogVerificationService` is disabled by default); it provides a single point of access to those services and a consolidated view of their processing results.

The history log for each of the registered services can be viewed from the Dynamo Admin UI. This log is maintained only in memory and is discarded after a redeployment of Oracle ATG Web Commerce. The Oracle ATG Web Commerce platform logs can be used as a historical reference to the service logs.

You can execute or schedule any functions for either the `AncestorGeneratorService` or the `CatalogVerificationService` from the `CatalogMaintenanceService`. Therefore, this service can be scheduled to sequentially execute other services it contains, as opposed to scheduling each service individually.

`CatalogMaintenanceService` includes the following properties as well as the properties listed at the beginning of the Batch Services (page 51) section:

Property	Description
`availableServices`	An array of Nucleus paths to each contained service component. Each component is resolved by path and added to the `servicesMap` property.
`availableFunctions`	A consolidated list of all the `availableFunctions` provided by the contained services.
`functionsToPerformByDefault`	Configures a default set of function names that are executed when none are provided, such as in schedule mode, or through the Java API. It can be any of the function names exposed by the `availableFunctions` property.

The `CatalogMaintenanceService` has a related `/atg/epub/CatalogMaintenanceHelper` component. This component is used by installations that use ATG Content Administration to manage their catalogs; it listens for ATG Content Administration deployments and runs catalog maintenance services automatically. For Oracle ATG Web Commerce purposes, the component's important property is `extraTriggeringAffectedItemDescriptors`. By default, the `CatalogMaintenanceHelper` listens for changes to catalog items such as `catalog`, `category`, `categoryInfo`, `product`, `productInfo`, `sku`, and `skuInfo`. If you have created custom subtypes based on these item types, you can add them to this property, so that changes to these item types can also trigger catalog maintenance services. For example:

```
extraTriggeringAffectedItemDescriptors+=myCustomDesc1,myCustomDesc2
```

For general information on ATG Content Administration, see the *ATG Content Administration Programming Guide*.

AncestorGeneratorService

Component: `/atg/commerce/catalog/custom/AncestorGeneratorService`.

The `AncestorGeneratorService` component generates ancestor categories for the `product` and `category` item types, and stores the names of these ancestor categories in the `ancestorCategories` property of each product and category.

The `AncestorGeneratorService` updates the following property values for each of the catalog item types. This service must be executed after making catalog updates in order for catalog navigation and search to work correctly.

Item Type	Property Names

Categories	`computedCatalogs`
	`ancestorCategories`
	`parentCategoriesForCatalog`
	`siteIds`

Products	`computedCatalogs`
	`ancestorCategories`
	`parentCategoriesForCatalog`
	`siteIds`

SKUs	`computedCatalogs` `sideIds`

If you have extended the Oracle ATG Web Commerce catalog schema, you can still use `AncestorGeneratorService` to generate ancestor categories, provided that:

- The catalog includes item types that represent categories and products (regardless of the actual names of these item types).
- The item types representing categories and products each have a property for storing the names of ancestor categories (regardless of the name of the property; note, however, that the property must have the same name for each item type).

Available Functions

AGS_GENCATALOGS: generates the `catalogs` properties

AGS_GENPARENTCATS: generates the `parentCategoriesForCatalog` property

AGS_GENANCESTORS: generates the `ancestorCategories` property

AGS_GENPROPERTIES: generates all properties for all items

AGS_GENPROPERTIES_FOR_CATEGORY: generates all properties for categories

AGS_GENPROPERTIES_FOR_PRODUCT: generates all properties for categories and products

CatalogVerificationService

Component: `/atg/commerce/catalog/custom/CatalogVerificationService`

The `CatalogVerificationService` verifies relationships between catalog items. It validates the following property values for each catalog item type:

Item Type	Property Names

Catalogs	`ancestorCategories`
	`allRootCategories`
	`directAncestorCatalogsAndSelf`
	`indirectAncestorCatalogs`

Item Type	Property Names

Categories	`fixedRelatedCategories`
	`categoryInfos`
	`parentCategory`

Products	`fixedRelatedProducts`
	`catalogsRelatedProducts`
	`productInfos`
	`parentCategoriesForCatalog`

SKUs	`fixedReplacementProducts`
	`catalogsReplacementProducts`
	`skuInfos`

Note: This service is disabled by default for performance reasons. Consider running this service periodically if any of the following apply:

- You define cross-sells on each product and want to be sure that your cross-sells don't cross catalogs
- You use breadcrumbs and want to be sure that the defined parent categories on products and categories make sense for your catalog structure
- You want to make sure the incrementally-maintained catalog properties (`ancestorCategories`, `allRootCategories`, `directAncestorCatalogsAndSelf`, `indirectAncestorCatalogs`) have not gotten out of sync

Available Functions

CVS_VERIFYCATALOGS: verifies catalog item properties

CVS_VERIFYCATEGORIES: verifies category item properties

CVS_VERIFYPRODUCTS: verifies product item properties

CVS_VERIFYSKUS: verifies SKU item properties

CatalogUpdateService

Component: `/atg/commerce/catalog/custom/CatalogUpdateService`

The `CatalogUpdateService` updates the same catalog properties that are updated by the dynamic service CatalogCompletionService (page 58), but in batch mode. Because the `CatalogCompletionService` can be disabled and may not be actively updating the catalog property values in real time, this service can be used to batch update them on an as-needed basis.

The `CatalogUpdateService` includes the following configurable properties:

Property	Description
`includeDynamicChildren`	This flag determines whether dynamic children should be calculated as part of processing.

Property	Description
`maxItemsPerTransaction`	Specifies how many items to include in each processing batch.
`projectWorkflow`	Identifies the workflow to use when creating a Content Administration project.
`serviceFunctions`	Sets the list of available functions. By default this includes on CUS_UPDATECATALOGS, which updates all item properties in the catalog.
`catalogIds`	Identifies the catalogs that should be updated.

The service can be run in a versioned Oracle ATG Web Commerce instance. If you use `startSQLRepository` to import catalog data, you can run this service immediately after the import to compute all properties, which can afterwards be maintained by `CatalogCompletionService`.

This service updates the following property values for each of the catalog item types.

Item Type	Property Names

Catalog	`directAncestorCatalogsAndSelf`
	`indirectAncestorCatalogs`
	`ancestorCategories`
	`allRootCategories`
	`siteIds`

Category	`siteIds`
	`computedCatalogs`
	`parentCategoriesForCatalog`

Catalog Folder	`siteIds`

Available Functions

CUS_UPDATECATALOGS: Updates all item properties in the catalog

Dynamic Services

The dynamic services are components that enable catalog properties to be dynamically updated as the catalog structure is modified by an ACC user, BCC user, or a program using the Repository API.

- CatalogChangesListener (page 58)
- PropertiesChangedHandler Components (page 58)

- CatalogCompletionService (page 58)

CatalogChangesListener

Component: `/atg/commerce/catalog/custom/CatalogChangesListener`

This component registers itself at deployment. It is notified for each repository change made to a catalog folder, category or catalog item in the product catalog.

As each change is made, the `CatalogChangesListener` calls the appropriate `PropertiesChangedEventHandler` listed in the `eventHandlers` property. The event handlers then call the `CatalogCompletionService`.

The `CatalogChangesListener` includes the following configurable properties:

Property	Description
`enabled`	Determines whether or not this component is used.

`eventHandlers`	Lists the `PropertiesChangedEventHandler` components that are available for this component. By default, the list includes the following:
	`CatalogFolderPropertiesChangedHandler`
	`CatalogPropertiesChangedHandler`
	`CategoryPropertiesChangedHandler`
	If you want to listen for changes on additional item types, you can implement a new `PropertiesChangedEventHandler` component and add it to the `eventHandlers` map.

PropertiesChangedHandler Components

The `PropertiesChangedHandler` components are registered with the `CatalogChangesListener` component. There are three `PropertiesChangedHandler` components:

- `CatalogFolderPropertiesChangedHandler`
- `CatalogPropertiesChangedHandler`
- `CategoryPropertiesChangedHandler`

Each `PropertiesChangedHandler` listens for changes to its designated property type and then calls the `CatalogCompletionService` to make the necessary changes.

CatalogCompletionService

Component: `/atg/commerce/catalog/custom/CatalogCompletionService`

This component updates catalog item property values in real time based on changes made to the product catalog repository. It is called by the `PropertiesChangedHandler` components when changes are made to the product catalog repository.

Refer to the CatalogUpdateService (page 56) for a list of properties that are maintained by this service.

Running Catalog Maintenance Services

This section describes the options available for running the catalog maintenance services. You can also configure services to run on a scheduled basis.

Running Batch Services from the Commerce Admin Page

Catalog Maintenance batch services should be run the in a staging environment rather than against the production database. They should be run after any structural changes are made to the catalog. For example, it should be run after adding new categories, products, or SKUs.

The catalog batch maintenance services are available from the Commerce Administration page. For more information on accessing the Dynamo Administration UI, see the *ATG Installation and Configuration Guide*.

Important: If you are running ATG Content Administration, you must configure the configure one agent server on your target cluster to run the Catalog Maintenance Services.

There are four options available on the menu; Catalog Update, Catalog Verification, Basic Maintenance and View Status.

Catalog Update

This `CatalogUpdateService` batch updates catalog property values that are normally updated incrementally by the `CatalogCompletionService`. When `CatalogUpdateService` is enabled, all updates to the catalog that are made using the Repository API (this includes the ATG Control Center, the Oracle ATG Web Commerce Business Control Center, and Oracle ATG Web Commerce Merchandising) trigger these properties to be computed and updated dynamically. The default is to run all the functions of the `CatalogUpdateService`.

If the DCS catalog is updated using some process other than the Repository API, or if `CatalogUpdateService` is disabled, you can run `CatalogUpdateService` manually to batch update the appropriate property values.

To run `CatalogUpdateService` from the Commerce Administration Page, click on the Catalog Update link. Click on the Start Process button at the bottom of the Catalog Update page.

Catalog Verification

This process verifies catalog component relationships for accuracy. The default is to run all the functions of the `CatalogVerficationService`.

To run `CatalogVerficationService` from the Commerce Administration Page, click on the Catalog Verification link. Click on the Start Process button at the bottom of the Verify Catalog page.

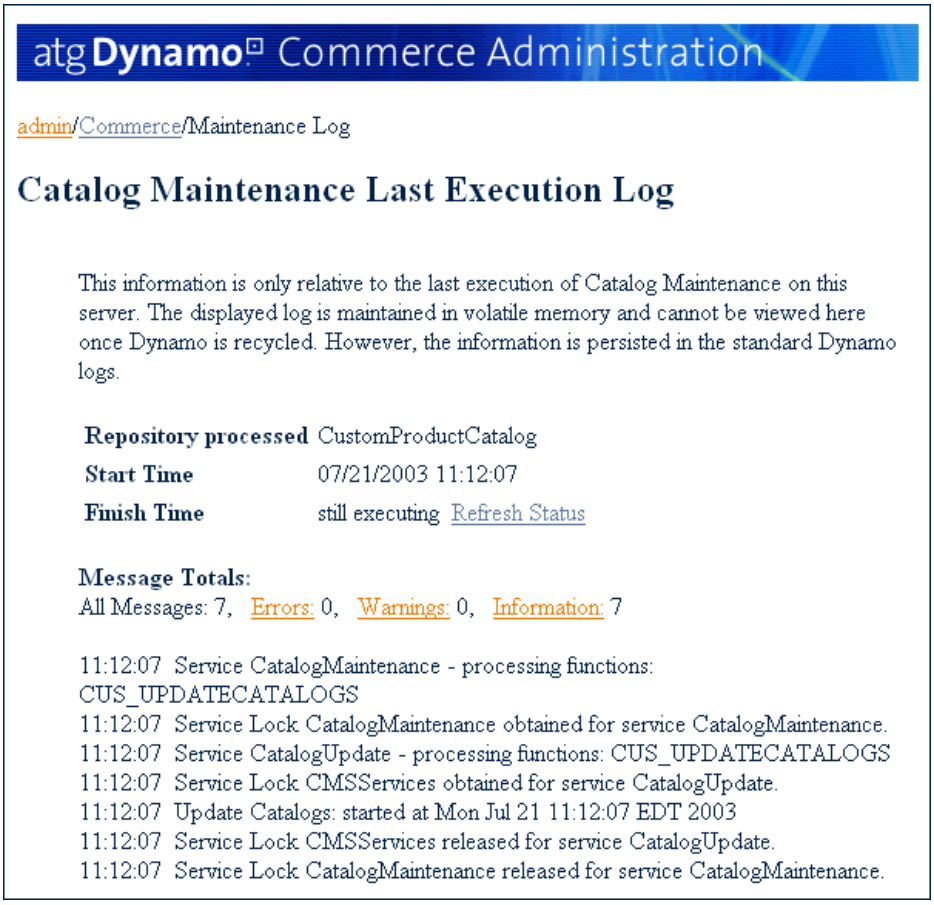
Basic Maintenance

This process executes the standard batch maintenance services against the DCS catalog.
The default configuration runs all the functions of the `AncestorGeneratorService` and `CatalogVerificationService`.

To run Basic Maintenance from the Commerce Administration Page, click on the Basic Maintenance link. Click on the Start Process button at the bottom of the Basic Maintenance page.

View Status

The View status page let you view the information, errors and warnings from the last execution of maintenance on the server. To view the status log from the Commerce Administration Page, click on the View Status link. The status from the last execution displays. For example:



Running a Batch Service from the ACC

To use the ACC to execute a service:

1. In the Components > by Path window, select the component by path: `/atg/commerce/catalog/CatalogMaintenanceService`.
2. Choose File > Open Component.

   The Component Editor opens.
3. If the component is not currently running, choose Component > Start Component.
4. Select the Methods tab.
5. Click the Invoke button next to the `performService` method.

Batch Maintenance Form Handler

Component: `/atg/commerce/catalog/RunServiceFormHandler`

There is one form handler that is used to execute the batch services from the Commerce Administration Pages. This form handler executes the `CatalogMaintenanceService`, passing a configurable set of functions for each available option. The execution of the `CatalogMaintenanceService` is started in a new thread, then the View Status page is shown. The View Status can be refreshed to monitor the progress of the batch maintenance run.

Property information:

Property	Description
`basicMaintenanceFunctions`	These are the function names passed to the `CatalogMaintenanceService` for the execution of the Basic Catalog Maintenance option.
`verifyFunctions`	These are the function names passed to the `CatalogMaintenanceService` for the execution of Verify Catalog option.
`updateFunctions`	These are the function names passed to the `CatalogMaintenanceService` for the execution of Update Catalog option.

Running Dynamic Services

The dynamic service components are started automatically. You can disable the `CatalogCompletionService` by setting the service's `enabled` property to `false`. If you set the `enabled` property to `false`, the component still starts up with Oracle ATG Web Commerce, but does not do anything if called.

5 Oracle ATG Web Commerce Profile Extensions

Oracle ATG Web Commerce extends the profile configuration and classes of the Personalization module in several ways to add functionality. Commerce components rely on these extensions to perform their logic. Removing or modifying the profile additions may require modifying logic in Commerce.

This chapter includes the following sections:

Profile Repository Extensions

The Personalization module's Profile Repository is an instance of the Generic SQL Adapter. It is located in `/atg/userprofiling/ProfileAdapterRepository`. In this adapter, the Personalization module defines a base `user` Item Descriptor. This user has many general attributes defined, such as first and last name, e-mail address, date of birth and home address. See the *Standard User Profile Repository Definition* section of the *Setting Up a Profile Repository* chapter in the *ATG Personalization Programming Guide* for more information on the attributes included.

Commerce property values can be determined in either of the following two ways:

- Retrieved directly from the database. This case is straightforward, and is used when properties always have unique values per individual.
- Derived from the user's organization. In this case, the property can be set at any level in the organization and overridden for individual cases only when necessary.

In many cases, Commerce provides two versions of the property, which permits you to use whichever method you find appropriate.

User Properties

Oracle ATG Web Commerce adds attributes to the `user` item descriptor to support gift lists, wish lists, promotions, address books, credit card collection and other features. Most properties are common business domain information; explanatory notes are provided for some properties.

Database-backed Properties	Derived Properties	Notes
`activePromotions`		Stores the list of promotions that can be used by the user in pricing their orders.
`allowPartialShipment`		This Boolean value can be used as the user's default setting for allowing partial shipments. If your sites support multiple shipping groups, you can give the customer the option of allowing a shipping group to be automatically split if some items cannot be shipped together (e.g. some items are `backorderId`). See the *Configuring the Order Fulfillment Framework* (page 413) chapter for information on allowing partial shipments.
`approvalRequired`	`derivedApprovalRequired`	
`approvers`	`derivedApprovers`	
`billingAddress`	`derivedBillingAddress`	
`costCenters`	`derivedCostCenters`	
`creditCards`	`derivedCreditCards`	
`currentLocation`		An enumerated attribute to be used with targeting. In the associated JSPs, use a `<setvalue>` call to set the current location. For example: `<dsp:setvalue bean="Profile.currentLocation" value="home"/>`
`daytimeTelephoneNumber`		
`defaultCarrier`		

Database-backed Properties	Derived Properties	Notes
defaultCostCenter	derivedDefaultCostCenter	
defaultCreditCard	derivedDefaultCreditCard	
expressCheckout		
giftlistAddresses		Stores addresses that can be associated with specific gift lists.
giftlists		Stores the list of gift lists created by a user when they register an event.
myCatalog	derivedCatalog catalog	
myPriceList	derivedPriceList priceList	
mySalePriceList	salePriceList	
orderPriceLimit	derivedOrderPriceLimit	
otherGiftlists		Stores the list of gift lists from which a customer is currently shopping. These are registries of other customers that have been accessed by the customer using the gift list search feature.
preferredVendors	derivedPreferredVendors	
purchaseLists		
secondaryAddresses	derivedSecondaryAddresses	
shippingAddress	derivedShippingAddress	
usedPromotions		Stores promotions that can no longer be used. A promotion is moved to the used list if it was created with a limited number of uses, and the user has reached the threshold.
wishlist		Stores the index to the default wish list created for a user.

The billing and shipping addresses are `contactInfo` objects, and the secondary addresses are a map of `contactInfo` objects. It is intended that the `billingAddress` and `shippingAddress` attributes are for "default" address values for the user. The user can create nicknames for other addresses and store those as

the key/value pairs in the `secondaryAddresses` map. The `defaultCarrier` indicates the user's preferred shipping service.

The `giftlist` repository is located in `/commerce/gifts/Giftlists`. For more information on gift lists and wish lists, see the *Configuring Commerce Services* (page 71) chapter.

Organization Properties

Oracle ATG Web Commerce adds the following attributes to the `organization` item descriptor to support the order approval, contracts, and cost center features:

Database-Backed Property	Derived Property
`approvalRequired`	`derivedApprovalRequired`
`approvers`	`derivedApprovers`
`billingAddress`	`derivedBillingAddress`
`billingAddrs`	`derivedBillingAddrs`
`contacts`	
`contract`	`derivedContract`
`costCenters`	`derivedCostCenters`
`creditCards`	`derivedCreditCards`
`customerType`	`derivedCustomerType`
`defaultCostCenter`	`derivedDefault CostCenter`
`defaultCreditCard`	`derivedDefault CreditCard`
`dunsNumber`	
`orderPriceLimit`	`derivedOrderPriceLimit`
`preferredVendors`	`derivedPreferredVendors`
`shippingAddress`	`derivedShippingAddress`
`shippingAddrs`	`derivedShippingAddrs`
`type`	

Profile Form Handler Extensions

The Oracle ATG Web Commerce profile form handler (`atg.commerce.profile.CommerceProfileFormHandler`) is a subclass of `atg.userprofiling.ProfileFormHandler`. It performs operations specific to Commerce. For example, an anonymous user can accumulate promotions in their `activePromotions` attribute. During login, the anonymous user's active promotions are copied into the list of active promotions of the persistent user. During the registration and login process, any shopping carts (Orders) created before logging in are changed to be persistent.

If the user's persistent profile contains any old shopping carts (Orders in an `incomplete` state), these shopping carts are loaded into the user's session. After log in, the `PricingModelHolder` component is reinitialized to cache all the promotions that the persistent user has accumulated during previous sessions. These operations are performed by overriding the `addPropertiesOnLogin`, `postCreateUser` and `postLoginUser` methods from the `ProfileFormHandler`.

Profile Tools and Property Manager Extension

The Oracle ATG Web Commerce profile tools class (`atg.commerce.profile.CommerceProfileTools`) is a subclass of `atg.userprofiling.ProfileTools`. It provides additional methods to access commerce specific profile properties such as shipping and billing addresses and credit card information. The Commerce Property Manager (`atg.commerce.profile.CommercePropertyManager`) is a subclass of `atg.userprofiling.PropertyManager`. It provides additional access to profile property names specific to Commerce customers that are used by `CommerceProfileTools`.

For example, a registered Commerce customer may have multiple shipping addresses. These different shipping addresses are managed by the `CommerceProfileTools` component. This extension of the tools component provides methods to create, delete and update profile repository shipping addresses as well as to retrieve all shipping addresses. The shipping address property names are stored in `CommercePropertyManager`. This provides a convenient central location to store property names that may change depending on the site.

6 Configuring Commerce for Multisite

This chapter describes the Commerce-specific configuration tasks required in order to use Oracle ATG Web Commerce's multisite feature. The term *multisite* refers to running multiple Web sites from a single Oracle ATG Web Commerce instance. For example, a clothing manufacturer with two brands, a bargain brand and a luxury brand, could create and manage a separate Web site for each brand from one instance of the Oracle ATG Web Commerce platform. The sites could be configured to share items such as shopping carts.

This chapter includes the following sections:

Site Repository Extensions for Commerce

Oracle ATG Web Commerce extends the site repository with the following Commerce-specific properties:

- `catalog`
- `listPriceList`
- `salePriceList`

This allows you to associate catalogs and price lists with sites through Site Administration, as described in the following section. The `CommerceSitePropertyManager` component provides access to these properties through its own properties:

- `catalogPropertyName`
- `listPriceListPropertyName`
- `salePriceListPropertyName`

Configuring Commerce Options in Site Administration

Site Administration is the main tool for creating and configuring sites. For general information on Site Administration, see the *ATG Multisite Administration Guide*.

When you create sites in Site Administration, some of the configuration options you set are related to your Oracle ATG Web Commerce application. Note that if you have created your own site templates, or extended the provided template, you may have different options or additional options.

In Site Administration, configure the following:

1. When you select the site type, select the Commerce option.
2. On the Operations tab, provide the following default Commerce information for the site:
   - Default Catalog
   - Default List Price List
   - Default Sale Price List
   - Maximum Coupons per Profile

     Note: Site Administration settings, if used, override global settings for this property. See Tracking and Limiting Coupon Uses (page 112) in the *Configuring Commerce Services* (page 71) chapter for information.
3. Configure the shareable type, if necessary. Commerce by default includes a single shareable type, `atg.ShoppingCart`, which controls sharing for both the shopping cart and product comparison lists.

Assigning Price Lists and Catalogs in a Multisite Configuration

As described in Configuring Commerce Options in Site Administration (page 69), you select default catalog and (optionally) price lists for each site in your multisite environment during initial configuration. This section describes how Oracle ATG Web Commerce determines which catalog and price lists to assign to the user's profile when a user visits a given site, allowing the appropriate items to display.

For the profile's `catalog` property, the `/atg/userprofiling/CatalogProfilePropertySetter` component calls the `determineCatalog` method of the `/atg/commerce/catalog/CatalogTools` component. This method invokes the `/atg/commerce/util/ContextValueRetriever` component. If this component's `useProfile` property is `false` (the default), the following logic is applied:

- If there is a current site (the application is running in a multisite environment), use the value of the `defaultCatalog` property of the `siteConfiguration` item for the current site.
- Otherwise, use the value of the `defaultCatalog` property of the `CatalogTools` component.

Similar logic is used to set the profile's `priceList` and `salePriceList` properties. The `/atg/userprofiling/PriceListProfilePropertySetter` component calls the `/atg/commerce/pricing/priceLists/PriceListManager` component's `determinePriceList` method, which calls `ContextValueRetriever`.

For more information on the `ContextValueRetriever`, see ContextValueRetriever Class (page 49).

7 Configuring Commerce Services

This chapter includes the following information on Oracle ATG Web Commerce services:

Setting Up Gift Lists and Wish Lists

Gift lists are lists of items selected by a site visitor. Gift lists can be used in two ways:

- **As gift lists**
  Gift lists are lists of products that other site visitors can view. Customers can use gift lists to register for events, such as birthdays or weddings. Customers c an create an unlimited number of gift lists for themselves. Part of the purchase process allows for special handling instructions for gift purchasing, for example, address security, wrapping, and shipping.

- **As wish lists**
  Customers can use wish lists to save lists of products without placing the items in their shopping carts. A wish list is actually a gift list that is accessible to only the person who created it. Customers can access their wish lists every time they log into their accounts, and they can purchase items from their wish lists at any time. All customers have a single default wish list.

The gift list class package is `atg.commerce.gifts`. Refer to the *ATG Platform API Reference* for more information on the associated classes and programming interfaces. The following scenarios demonstrate how gift lists can be used on a commerce site.

Example of Using a Gift List

The following scenario describes how a site customer can use a gift list while shopping on a store web site. John Smith logs onto the site and creates a list of gifts he would like to receive for his birthday. Mike, John's friend, then visits the site and searches for "John Smith." Mike can use advanced search criteria to refine his search by including a state, event name, or event type.

The store returns a list of gift lists found. Mike selects the correct list and views the items that John selected. Mike sees that John needs two new inner tubes for his bike. Mike can see that someone has already purchased one of the inner tubes. Mike decides to purchase the other inner tube for John and adds it to his cart. During the checkout process, Mike can send the item to his own address or send it directly to John. The gift list now reflects that two inner tubes have been purchased for John.

Example of Using a Wish List

The following scenario describes how a site customer uses a wish list while shopping on a store web site. A customer, Sally, browses the store and finds items she would like to purchase, but can not afford to buy right now. Sally can add the items to her private wish list and save the list. The next time she visits the site, she can view her wish list. She can buy the items by moving them to her shopping cart. The remaining items are saved in the wish list for later visits. A wish list is not searchable or visible by other customers of the store. It serves only as a holder of items the customer has found.

Gift List Functionality

The gift list functionality is broken down into the following sub-areas. Each of these sub-areas is described in detail later in this section.

- Gift List Business Layer Classes (page 72): Business layer classes contain the logic for gift list creation and manipulation. These classes consist of a `GiftlistManager` and `GiftlistTools`.
- Gift List Repository (page 73): The repository definition maps the `Giftlists` repository to database tables. The business layer classes access the gift lists through the repository layer.
- Gift List Form Handlers (page 77) and Gift List Servlet Beans (page 87): The form handlers and servlet beans provide an interface to the `Giftlists` repository. They provide the interface between the UI and the business layer classes to create and edit gift lists.
- Purchase Process Extensions to Support Gift Lists (page 89): The purchase process has been extended to support purchasing gifts from published gift lists. This section describes the components that are used to support this functionality. For example, these components support adding an item from a gift list to a shopping cart and updating the `Giftlists` repository at checkout.
- Extending Gift List Functionality (page 95): Gift list functionality provided in Oracle ATG Web Commerce supports most requirements of commerce sites. However, you can extend this functionality if needed. This section describes what steps to take if extending the system is necessary.
- Disabling the Gift List Repository (page 96): Describes how to disable the `Giftlists` repository. Disable the repository if you are not going to use the gift list functionality.

Gift List Business Layer Classes

The business layer classes contain the logic for managing gift lists and items. The methods within these classes are used to create, update, and delete selected gift lists for a given customer. The business layer components are the interface to the `Giftlists` repository. All calls to modify a gift list are made through these classes.

Gift list business layer classes include the following:

- `GiftlistManager`: The majority of the functionality for gift list management exists in the `GiftlistManager` component. The class contains such methods as `createGiftlist`, `updateGiftlist`, `addItemToGiftlist` and `removeItemFromGiftlist`. These methods are higher level than those in `GiftlistTools` and mostly perform validation tasks before calling `GiftlistTools` to update the gift list repository.
- `GiftlistTools`: The `GiftlistTools` component is the low level interface and contains the logic for creating and editing gift lists in the repository. `GiftlistTools` is not usually called directly. Generally, it is called by the `GiftlistManager` class to perform tasks on gift lists.

Because these classes do not contain any state, it makes sense for them to exist as globally-scoped services in Oracle ATG Web Commerce. Rather than constructing a new object when required, Commerce places one instance of each component into the Nucleus hierarchy to be shared. They can be found in the hierarchy under:

- `/atg/commerce/gifts/GiftlistTools`
- `/atg/commerce/gifts/GiftlistManager`

Note: Gift lists use the `ClientLockManager` component to control locks on repository items. The `ClientLockManager` prevents a deadlock situation or data corruption that could occur when multiple customers update the same gift list. The `ClientLockManager` is located at `/atg/dynamo/service/ClientLockManager`. See the Using Locking in Fulfillment (page 424) section of the *Fulfillment Framework* chapter for more information on the `ClientLockManager`.

Gift List Repository

The `Giftlists` repository is the layer between Oracle ATG Web Commerce and the database itself. It provides an interface to the database layer to persist gift list information. The `Giftlists` repository uses the SQL Repository implementation. For more information on SQL repositories, see the *ATG Repository Guide*.

The `Giftlists` repository is defined in the `giftlists.xml` definition file, located in the Commerce configuration path at `/atg/commerce/gifts/`. This XML file defines item descriptors for gift lists and gift list items. In addition, properties in the `userProfile.xml` definition file allow you to link user profiles with gift lists in the gift list repository. The `userProfile.xml` file is located in the Commerce configuration path at `/atg/userprofiling`.

The following example shows the content of the `/atg/commerce/gifts/giftlists.xml` file located in `<ATG10dir>/DCS/src/config/config.jar`.

Note: The `siteId` properties defined for both the `gift-list` and `gift-item` item descriptors is required for multisite environments only. See Gift and Wish Lists in a Multisite Environment (page 91) for more details.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE taglib
        PUBLIC "-//Art Technology Group, Inc.//DTD General SQL Adapter//EN"
        "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">

<gsa-template>

  <header>
    <name>Commerce Giftlists</name>
    <author>DCS Team</author>
    <version>$Id: //product/DCS/main/templates/DCS/config/atg/
        commerce/gifts/giftlists.xml#6 $$Change: 546512 $</version>
  </header>

  <!--
  ************************************************************
    GiftList (also gift registries)
  ************************************************************
  -->

  <item-descriptor name="gift-list" item-cache-size="1000" query-cache-size="1000"
      display-name-resource="itemDescriptorGiftList">
    <attribute name="resourceBundle"
        value="atg.commerce.GiftListsTemplateResources"/>
```

```
<attribute name="categoryBasicsPriority" value="10"/>
<attribute name="categoryShippingPriority" value="20"/>
<attribute name="categoryInfoPriority" value="30"/>

<table name="dcs_giftlist" type="primary" id-column-name="id">
  <property name="id" writable="false" category-resource="categoryInfo"
      display-name-resource="id">
    <attribute name="propertySortPriority" value="-10"/>
  </property>
  <property name="owner" item-type="user"
      repository="/atg/userprofiling/ProfileAdapterRepository"
      column-name="owner_id" category-resource="categoryBasics"
      display-name-resource="owner">
    <attribute name="propertySortPriority" value="-11"/>
  </property>
  <property name="siteId" data-type="string" column-name="site_id"
      category-resource="categoryInfo" display-name-resource="siteId">
    <attribute name="propertySortPriority" value="-10"/>
  </property>
  <property name="public" data-type="boolean" column-name="is_public"
      default="false" category-resource="categoryInfo"
      display-name-resource="public" required="true">
    <attribute name="propertySortPriority" value="-8"/>
  </property>
  <property name="published" data-type="boolean" column-name="is_published"
      default="false" category-resource="categoryInfo"
      display-name-resource="published" required="true">
    <attribute name="propertySortPriority" value="-7"/>
  </property>
  <property name="eventName" data-type="string" column-name="event_name"
      category-resource="categoryBasics" display-name-resource="eventName">
    <attribute name="propertySortPriority" value="-10"/>
  </property>
  <property name="eventDate" data-type="timestamp" column-name="event_date"
      category-resource="categoryBasics" display-name-resource="eventDate">
    <attribute name="propertySortPriority" value="-9"/>
  </property>
  <property name="eventType" data-type="enumerated" default="other"
      column-name="event_type" category-resource="categoryBasics"
      display-name-resource="eventType">
    <attribute name="useCodeForValue" value="false"/>
    <option resource="valentinesDay" code="0"/>
    <option resource="wedding" code="1"/>
    <option resource="bridalShower" code="2"/>
    <option resource="babyShower" code="3"/>
    <option resource="birthday" code="4"/>
    <option resource="anniversary" code="5"/>
    <option resource="christmas" code="6"/>
    <option resource="chanukah" code="7"/>
    <option resource="otherHoliday" code="8"/>
    <option resource="iJustWantThisStuff" code="9"/>
    <option resource="other" code="10"/>
    <attribute name="propertySortPriority" value="-8"/>
  </property>
  <property name="comments" data-type="string" column-name="comments"
      category-resource="categoryBasics" display-name-resource="comments">
    <attribute name="propertySortPriority" value="-10"/>
  </property>
  <property name="description" data-type="string" column-name="description"
```

```
        category-resource="categoryBasics" display-name-resource="description">
      <attribute name="propertySortPriority" value="-10"/>
    </property>
    <property name="instructions" data-type="string" column-name="instructions"
        category-resource="categoryShipping"
        display-name-resource="instructions">
      <attribute name="propertySortPriority" value="-5"/>
    </property>
    <property name="lastModifiedDate" data-type="timestamp"
        column-name="last_modified_date" category-resource="categoryInfo"
        display-name-resource="lastModifiedDate">
      <attribute name="uiwritable" value="false"/>
      <attribute name="propertySortPriority" value="-5"/>
    </property>
    <property name="creationDate" data-type="timestamp"
        column-name="creation_date" category-resource="categoryInfo"
        display-name-resource="creationDate">
      <attribute name="uiwritable" value="false"/>
      <attribute name="useNowForDefault" value="true"/>
      <attribute name="propertySortPriority" value="-6"/>
    </property>
    <property name="shippingAddress" item-type="contactInfo"
        repository="/atg/userprofiling/ProfileAdapterRepository"
        column-name="shipping_addr_id" category-resource="categoryShipping"
        display-name-resource="shippingAddress">
      <attribute name="propertySortPriority" value="-6"/>
    </property>
   </table>
   <table name="dcs_giftinst" type="multi" id-column-name="giftlist_id"
       multi-column-name="tag">
     <property name="specialInstructions" column-name="special_inst"
        component-data-type="string" data-type="map"
        category-resource="categoryShipping"
        display-name-resource="specialInstructions">
      <attribute name="propertySortPriority" value="-4"/>
    </property>
   </table>
   <table name="dcs_giftlist_item" type="multi" id-column-name="giftlist_id"
       multi-column-name="sequence_num">
     <property name="giftlistItems" data-type="list"
        component-item-type="gift-item" column-name="giftitem_id"
        cascade="delete" category-resource="categoryBasics"
        display-name-resource="giftlistItems">
      <attribute name="propertySortPriority" value="-7"/>
    </property>
   </table>
</item-descriptor>

<item-descriptor name="gift-item" display-property="displayName"
    item-cache-size="1000" query-cache-size="1000"
    display-name-resource="itemDescriptorGiftItem">
  <attribute name="resourceBundle"
      value="atg.commerce.GiftListsTemplateResources"/>
  <attribute name="categoryBasicsPriority" value="10"/>
  <attribute name="categoryInfoPriority" value="20"/>

  <table name="dcs_giftitem" type="primary" id-column-name="id">

    <property name="id" column-name="id" writable="false"
        category-resource="categoryInfo" display-name-resource="id">
```

```
        <attribute name="propertySortPriority" value="-10"/>
      </property>
      <property name="catalogRefId" data-type="string"
          column-name="catalog_ref_id"
          editor-class="atg.ui.commerce.SkuItemStringEditor"
          category-resource="categoryInfo" display-name-resource="catalogRefId">
        <attribute name="propertySortPriority" value="-9"/>
      </property>
      <property name="productId" data-type="string" column-name="product_id"
          editor-class="atg.ui.commerce.ProductItemStringEditor"
          category-resource="categoryInfo" display-name-resource="productId">
        <attribute name="propertySortPriority" value="-8"/>
      </property>
      <property name="siteId" data-type="string" column-name="site_id"
          category-resource="categoryInfo" display-name-resource="siteId">
        <attribute name="propertySortPriority" value="-7"/>
      </property>
      <property name="displayName" data-type="string" column-name="display_name"
          category-resource="categoryBasics" display-name-resource="displayName">
        <attribute name="propertySortPriority" value="-10"/>
      </property>
      <property name="description" data-type="string" column-name="description"
          category-resource="categoryBasics" display-name-resource="description">
        <attribute name="propertySortPriority" value="-9"/>
      </property>
      <property name="quantityDesired" data-type="long"
          column-name="quantity_desired" category-resource="categoryBasics"
          display-name-resource="quantityDesired">
        <attribute name="propertySortPriority" value="-8"/>
      </property>
      <property name="quantityPurchased" data-type="long"
          column-name="quantity_purchased" category-resource="categoryBasics"
          display-name-resource="quantityPurchased">
        <attribute name="propertySortPriority" value="-7"/>
      </property>
    </table>
  </item-descriptor>

</gsa-template>
<!-- @version $Id:
//product/DCS/main/templates/DCS/config/atg/commerce/gifts/giftlists.xml#6
$$Change: 546512 $ -->
```

The following excerpt from the `/atg/userprofiling/userProfile.xml` file, located in `<ATG10dir>/DCS/config/config.jar` shows how gift lists are associated with user profiles.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE taglib
  PUBLIC "-//Art Technology Group, Inc.//DTD General SQL Adapter//EN"
  "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">

<gsa-template>

  <header>
    <name>Commerce Related Profile Changes</name>
    <author>DCS Team</author>
    <version>$Id: userProfile.xml,v 1.24 2000/05/03 03:51:19
       petere Exp $</version>
```

```
  </header>

  <item-descriptor name="user" default="true"
      sub-type-property="userType">

    <!-- key into private wishlist -->
    <table name="dcs_user_wishlist" type="auxiliary"
     id-column-name="user_id">
      <property name="wishlist" item-type="gift-list"
          repository="/atg/commerce/gifts/Giftlists" column-name="giftlist_id"
          cascade="insert,update,delete"/>
    </table>

    <!-- key into user created giftlists -->
    <table name="dcs_user_giftlist" type="multi" id-column-name="user_id"
          multi-column-name="sequence_num">
      <property name="giftlists" data-type="list"
          component-item-type="gift-list"
          repository="/atg/commerce/gifts/Giftlists"
          column-name="giftlist_id" cascade="delete"/>
    </table>

    <!-- key into giftlists found for other customers -->
    <table name="dcs_user_otherlist" type="multi" id-column-name="user_id"
          multi-column-name="sequence_num">
      <property name="otherGiftlists" data-type="list"
          component-item-type="gift-list"
          repository="/atg/commerce/gifts/Giftlists"
          column-name="giftlist_id"/>
    </table>

</gsa-template>
```

Gift List Form Handlers

Form handlers process forms and pages. They provide an interface between the customer and the business layer classes that have access to the `Giftlists` repository. Forms (or JSPs) use these handlers to take input from the user and call methods to perform actions on the `Giftlists` repository. For more information on form handlers, see the *Working with Forms and Form Handlers* chapter in the *ATG Platform Programming Guide* and the *Using Repository Form Handlers* chapter in the *ATG Page Developer's Guide*.

This section describes the following gift list form handlers:

- GiftlistFormHandler (page 77)
- GiftlistSearch (page 84)

GiftlistFormHandler

The `/atg/commerce/gifts/GiftlistFormHandler` accepts input from the customer to create, update and delete gift lists, as well as to add items to and remove items from gift lists. Properties in the handler are used to store user input for processing, as well as to store success and failure URLs for redirect after processing. Some handle methods have pre- and post- methods defined to make it easier to extend the methods.

Note: You can remove items from a gift list by using the `GiftlistFormHandler` to update the item's quantity to 0. Alternatively, you can use the `GiftitemDroplet` to remove items. See `GiftitemDroplet` in Gift List Servlet Beans (page 87) for more information .

The `GiftlistFormHandler` should be session-scoped because multiple pages usually gather the information needed for gift list management. The customer should be able to enter information on different pages to configure the same gift list.

GiftlistFormHandler Properties

The `GiftlistFormHandler` has the following properties that support the management of gift lists and items in the repository.

Property	Function
`giftlistManager`	The manager component that interfaces with the `Giftlists` repository.
`catalogTools`	The tools component that performs low-level operations on the catalog repository.
`CommerceProfileTools`	The tools component that performs low-level operations on the user profile repository.
Giftlist properties	Properties that store gift list attributes, as entered by the customer. For example, `eventName`, `eventDate`, `description`.
Success and failure URL properties	Properties that tell the Oracle ATG Web Commerce platform what pages to redirect the customer to after an action is performed. Both success and failure URL properties are provided for each handle method.
`siteId`	Used in multisite environments only. In multisite environments, if you don't want to use the current site as the `siteId` when creating a gift list or adding a gift item, you can set the `GiftlistFormHandler`'s `siteId` property and use it instead. Typically, this property would be set by the JSP page. See the Gift and Wish Lists in a Multisite Environment (page 91) for more information.

GiftlistFormHandler Handle Methods

The `GiftlistFormHandler` has a number of `handle` methods. Many of the `handle` methods have corresponding `pre`*`methodName`*, `post`*`methodName`*, and *`methodName`* methods. For example, the `handleCreateGiftlist` method has corresponding `preCreateGiftlist`, `postCreateGiftlist` and `createGiftlist` methods. A handle method calls its `pre`*`methodName`* method before executing its *`methodName`* method. Likewise, it calls its `post`*`methodName`* method after executing its *`methodName`* method. The `pre` and `post` methods, whose default implementations are empty, provide an easy way to extend the functionality of the handle methods. The exceptions to this rule are `handleDeleteGiftlist`, `handleSaveGiftlist`, and `handleUpdateGiftlist`. These three methods have `pre` and `post` methods but they must call the `GiftlistManager` class to accomplish their primary tasks of deleting, saving, and updating gift lists.

`GiftlistformHandler` also has a set of `successURL` and `errorURL` properties that map to its handle methods. For example, `handleCreateGiftlist` has corresponding `createGiftlistSuccessURL` and `createGiftlistErrorURL` properties. After a handle method executes, you can use these properties to redirect the customer to pages other than those specified by the form's `action` attribute. The redirected page's content depends on the type of operation and whether the operation succeeded or not. For example, if an attempt to create a gift list fails, you could redirect the customer to a page explaining what missing information

caused the failure. If the value for a particular success or failure condition is not set, no redirection takes place and the form is left on the page defined as the `action` page.

The value of the redirect properties is a URL relative to the `action` page of the form. You can either specify the values of these URL properties in the properties of the form handler or you can set them in the JSP itself using a `hidden` tag attribute. For example, you can set the `addItemToGiftlistSucessURL` property with this tag:

```
<dsp:input bean="GiftlistFormHandler.addItemToGiftlistSucessURL"
 value="../user/lists.jsp " type="hidden"/>
```

The following table lists the `GiftlistFormHandler` handle methods, along with each method's `pre` and `post` methods, and success/failure URLs.

Method	Function

`handleAddItemToGiftlist()`	Adds items to a gift list during the shopping process, using the following properties taken from the form: quantity, `catalogRefIds` (an array of SKU IDs), `giftlistId`, and `siteId` (multisite environments only).
	
	`handleAddItemToGiftlist()` calls `GiftlistFormHandler.addItemToGiftlist()`, whose primary responsibility is to call `GiftlistManager.addCatalogItemToGiftlist()`, where the actual work of adding an item to a gift list is done. `addCatalogItemToGiftlist()` performs several steps to create the item and then add it to the gift list. First, it determines whether an item already exists in the gift list with the same SKU ID, product ID, and, in multisite environments, site ID. If an item already exists, `addCatalogItemToGiftlist()` increments the quantity of the item. If a corresponding item doesn't already exist, `addCatalogItemToGiftlist()` creates the gift item. Next, `addCatalogItemToGiftlist()` calls `GiftlistManager.addItemToGiftlist()` to add the newly created gift item to the specified gift list. In multisite environments, `addItemToGiftlist()` also determines whether the gift item and the gift list have compatible site IDs before adding the item to the list (see Gift and Wish Lists in a Multisite Environment (page 91) for more details).
	
	Associated Methods:
	`addItemToGiftlist()`
	`preAddItemToGiftlist()`
	`postAddItemToGiftlist()`
	`GiftlistManager.addCatalogItemToGiftlist()`
	
	Success and Failure URL properties:
	`addItemToGiftlistSuccessURL`
	`addItemToGiftlistErrorURL`

Method	Function

`handleCreateGiftlist()`	Resets the properties in the `GiftlistFormHandler` in preparation for creating a new gift list.
	
	Associated Methods:
	`createGiftList()`
	`preCreateGiftList()`
	`postCreateGiftList()`
	
	Success and Failure URL properties:
	`createGiftlistSuccessURL`
	`createGiftlistErrorURL`

`handleDeleteGiftlist()`	Deletes a gift list from the user's profile and from the repository. This method calls `GiftlistManager.removeGiftlist()` with the `profileId` and `giftlistId` to remove the gift list from the repository.
	
	Associated Methods:
	`preDeleteGiftlist()`
	`postDeleteGiftlist()`
	`GiftlistManager.removeGiftlist()`
	
	Success and Failure URL properties:
	`deleteGiftlistSuccessURL`
	`deleteGiftlistErrorURL`

Method	Function
handleMoveItemsFromCart()	Takes items out of the shopping cart and adds them to the gift list whose ID is passed into the form handler. handleMoveItemsFromCart() calls GiftlistFormHandler.moveItemsFromCart(). This method performs several steps to create the gift item and then add it to the gift list. First, it determines whether an item already exists in the gift list with the same SKU ID, product ID, and, in multisite environments, site ID. If an item already exists, moveItemsFromCart() increments the quantity of the item, using the quantity specified. If no quantity is specified, moveItemsFromCart() moves the entire quantity to the gift list. If a corresponding item doesn't already exist, moveItemsFromCart() calls GiftlistManager.createGiftlistItem() to create the gift item, based on the properties in the original commerce item, then moveItemsFromCart() calls GiftlistManager.addItemToGiftlist() to add the item to the specified gift list. In multisite environments, addItemToGiftlist() also determines whether the gift item and the gift list have compatible site IDs before adding the item to the list (see Gift and Wish Lists in a Multisite Environment (page 91) for more details). Finally, moveItemsFromCart() calls GiftlistFormHandler.updateOrder(). This method is responsible for updating the quantity of the commerce item in the shopping cart, or removing the item altogether if the entire quantity has been transferred to the gift list. **Associated Methods:** moveItemsFromCart() updateOrder() preMoveItemsFromCart() postMoveItemsFromCart() GiftlistManager.createGiftlistItem() GiftlistMnager.addItemToGiftlist() **Success and Failure URL properties:** moveItemsFromCartSuccessURL moveItemsFromCartErrorURL
handleSaveGiftlist()	Creates and saves gift lists in the Giftlists repository. This method calls createGiftlist() in the GiftlistManager component with gift list properties to create a gift list in the repository, save properties and add the gift list to the customer's profile. **Associated Methods:** preSaveGiftlist() postSaveGiftlist() GiftlistManager.createGiftlist() **Success and Failure URL properties:** saveGiftlistSuccessURL saveGiftlistErrorURL

Method	Function
handleUpdateGiftlist()	Updates the current gift list. This method calls updateGiftlist() in the GiftlistManager component, passing in gift list properties, to update a particular gift list in the repository. **Associated Methods:** preUpdateGiftlist() postUpdateGiftlist() GiftlistManager.updateGiftlist() **Success and Failure URL properties:** updateGiftlistSuccessURL updateGiftlistErrorURL
handleUpdateGiftlistItems()	Changes the quantity of a gift list item or removes the item from the list. **Associated Methods:** updateGiftlistItems() preUpdateGiftlistItems() postUpdateGiftlistItems() **Success and Failure URL properties:** updateGiftlistItemsSuccessURL updateGiftlistItemsErrorURL

GiftlistFormHandler Example

The GiftlistFormHandler.properties file is used to configure the GiftlistFormHandler. This file is located at /atg/commerce/gifts/ in <ATG10dir>/DCS/config/config.jar.

Note: The GiftlistFormHandler.siteId property is typically set in the JSP page, not in the GiftlistFormHandler.properties file.

```
$class=atg.commerce.gifts.GiftlistFormHandler
$scope=session

# Profile properties
profile=/atg/userprofiling/Profile
defaultLocale^=/atg/commerce/pricing/PricingTools.defaultLocale

# Giftlist repository
giftlistRepository=Giftlists

# Business layer giftlist manager
giftlistManager=GiftlistManager

# Business layer order manager
orderManager=/atg/commerce/order/OrderManager
shoppingCart=/atg/commerce/ShoppingCart
pipelineManager=/atg/commerce/PipelineManager

# commerce tools
giftlistTools=GiftlistTools
```

```
catalogTools=/atg/commerce/catalog/CatalogTools
profileTools=/atg/userprofiling/ProfileTools

# giftlist properties
itemType=gift-list
```

The following code sample demonstrates how to use the `GiftlistFormHandler` in a template. This serves as an example of how to display error messages, set up input and URL properties and make calls to handle methods in the form handler.

Note: This code sample works for both multisite and non-multisite environments. In multisite environments, the sample will use the current site's ID when setting the `siteId` property on the newly created gift list. To set a gift list's `siteId` to something other than the current site, you should add another `dsp:input` tag (hidden or otherwise) that sets the `siteId` property on the `GiftlistFormHandler`. For more information on the `siteId` property, see Gift and Wish Lists in a Multisite Environment (page 91).

```
<!-Import statements for components-->
<dsp:importbean bean="/atg/commerce/gifts/GiftlistFormHandler"/>
<dsp:importbean bean="/atg/dynamo/droplet/ErrorMessageForEach"/>
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:importbean bean="/atg/dynamo/droplet/Switch"/>

<!-- Display any errors processing form -->
<dsp:droplet name="Switch">
<dsp:param bean="GiftlistFormHandler.formError" name="value"/>
<dsp:oparam name="true">
  <UL>
    <dsp:droplet name="ErrorMessageForEach">
      <dsp:param bean="GiftlistFormHandler.formExceptions"
           name="exceptions"/>
      <dsp:oparam name="output">
        <LI> <dsp:valueof param="message"/>
      </dsp:oparam>
    </dsp:droplet>
    </UL>
</dsp:oparam>
</dsp:droplet>

<!-Save giftlist -->
<dsp:form action="lists.jsp" method="POST">
  <!-Success and error URLs -->
  <dsp:input bean="GiftlistFormHandler.saveGiftlistSuccessURL"
     value="./lists.jsp" type="hidden"/>
  <dsp:input bean="GiftlistFormHandler.saveGiftlistErrorURL"
     value="./new_list.jsp" type="hidden"/>
  <b>Event Name</b><br>
  <dsp:input size="40" type="text" bean="GiftlistFormHandler.eventName"/>
  <p>
  <b>Event Type</b>
   <dsp:select bean="GiftlistFormHandler.eventType">
   <dsp:droplet name="ForEach">
     <dsp:param bean="GiftlistFormHandler.eventTypes" name="array"/>
     <dsp:oparam name="output">
       <dsp:option paramvalue="element"><dsp:valueof
            param="element">UNDEFINED</dsp:valueof>
     </dsp:oparam>
     </dsp:droplet>
     </dsp:select><br>
```

```
<p>
<b>Event Description</b><br>
<dsp:setvalue bean="GiftlistFormHandler.description" value=""/>
<dsp:textarea bean="GiftlistFormHandler.description" value="" cols="40"
   rows="4"></dsp:textarea>
<p>
<b>Where should people ship the gifts?</b><p>
  <dsp:select bean="GiftlistFormHandler.shippingAddressId">
  <!-display address nicknames for profile to select from -->
  <dsp:droplet name="ForEach">
    <dsp:param bean="GiftlistFormHandler.addresses" name="array"/>
    <dsp:oparam name="output">
      <dsp:option paramvalue="key"/>
      <dsp:valueof param="element">UNDEFINED</dsp:valueof>
    </dsp:oparam>
  </dsp:droplet>
  </dsp:select><br>
<b>Gift list public?</b>
<p>
<dsp:input bean="GiftlistFormHandler.isPublished" value="true"
   type="radio" name="published"/> Make my list public now<br>
<dsp:input bean="GiftlistFormHandler.isPublished" value="false"
   checked="<%=true%>" type="radio" name="published"/> Don't make my list
   public yet
<dsp:input bean="GiftlistFormHandler.saveGiftlist" value="Save gift
   list" type="submit"/>
```

GiftlistSearch

The `/atg/commerce/gifts/GiftlistSearch` form handler searches the repository for gift lists. The form handler uses input from the customer, such as owner name, event name, event type and state, to find gift lists published by other customers. It returns a list of gift lists that match the given criteria.

`GiftlistSearch` should be session-scoped because multiple pages are typically involved in gathering and displaying information for gift list searching (for example, you might want to maintain a list of results for paging purposes). This form handler uses supporting servlet beans to add the retrieved gift lists to the customer's profile and to display gift list contents.

`GiftlistSearch` is configurable to support all gift list searching requirements. Booleans specify what types of searching are done. The configurable searches include:

- **Name Search**: Searches by the name of the gift list owner.
- **Advanced Search**: Searches for matches to specific gift list properties (such as event name, event type, and owner's state).
- **Published List Search**: When `true`, searches for published lists only. When `false`, searches for both published and unpublished gift lists.

GiftlistSearch Properties

`GiftlistSearch` has the following properties to support gift list searching:

Property	Function
`doNameSearch`	Specifies whether to search gift lists by the owner's name.

Property	Function
nameSearchPropertyNames	Specifies the fields to use during a name search (typically, owner.firstName and owner.lastName).
doAdvancedSearch	Specifies whether to search gift lists using properties other than the owner's name.
advancedSearchPropertyNames	Specifies the fields to use during an advanced search (for example, eventType, eventName, and state).
doPublishedSearch	Specifies whether to search only published gift lists.
publishedSearchPropertyNames	When searching only published gift lists, lists must be both public and published in order to be included in the search results. Therefore, if doPublishedSearch is set to true, set this value to public,published.
giftlistRepository	The repository that stores your gift lists. Set this value to Giftlists.
itemTypes	The gift list item type. Set this value to gift-list.
giftlistManager siteGroupManager siteScope	These three properties are required for multisite environments only. See Gift and Wish Lists in a Multisite Environment (page 91) for more details.
siteIds	This property is used in multisite environments only. See Gift and Wish Lists in a Multisite Environment (page 91) for more details.
searchInput	Input text parsed for searching. This property should be set by the JSP page.
searchResults	Giftlist repository items found based on searching criteria. Your results page must use this property to render search results.
searchSuccessURL	URL of the page to which the user is redirected on a successful search.
searchErrorURL	URL of the page to which the user is redirected on an error.

GiftlistSearch Handle Methods

GiftlistSearch has the following handle method:

Method	Function
handleSearch	handleSearch provides the core functionality of this form. This method builds a search query based on the configuration specified in the GiftlistSearch form handler's properties file, along with any properties set on the JSP page itself. It then applies the query to the Giftlists repository to find a list of gift lists. The list is stored in the searchResults property for the form to display.

GiftlistSearch Example

The following properties file is an example of how you configure the `GiftlistSearch` form handler. Note that the last three properties, `giftlistManager`, `siteGroupManager`, and `siteScope`, are required for multisite environments only. This properties file is located at `/atg/commerce/gifts/GiftlistSearch.properties` in `<ATG10dir>/DCS/config/config.jar`.

```
$class=atg.commerce.gifts.SearchFormHandler
$scope=session

doNameSearch=true
nameSearchPropertyNames=owner.firstName,owner.lastName

doAdvancedSearch=true
advancedSearchPropertyNames=eventType,eventName,state

doPublishedSearch=true
publishedSearchPropertyNames=public,published

giftlistRepository=Giftlists
itemTypes=gift-list

# Multisite properties: required for multisite enviromments only
giftlistManager=/atg/commerce/gifts/GiftlistManager
siteGroupManager=/atg/multisite/SiteGroupManager
siteScope^=/atg/commerce/gifts/GiftlistManager.siteScope
```

The following code sample demonstrates one method for using `GiftlistSearch` in a template in non-multisite environments.

Note: This code sample works for both multisite and non-multisite environments. In multisite environments, the sample will use the current site's ID when determining which gift lists to return. To return gift lists from sites other than the current one, you should add another `dsp:input` tag (hidden or otherwise) that sets the `siteIds` property on the `GiftlistSearch` form handler. For more information on the `siteIds` property, see Gift and Wish Lists in a Multisite Environment (page 91).

```
<!-Import statements for form components>
<dsp:importbean bean="/atg/commerce/gifts/GiftlistSearch"/>
<dsp:importbean bean="/atg/dynamo/droplet/IsEmpty"/>
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:importbean bean="/atg/dynamo/droplet/Switch"/>

<TITLE>Giftlist Search</TITLE>

<dsp:form action="giftlist_search.jsp">
<p>
<b>Find someone's gift list</b>
<hr size=0>
Name: <dsp:input bean="GiftlistSearch.searchInput" size="30" type="text"/>
<p>
Optional criteria that may make it easier to find the right list:
<p>
<dsp:droplet name="ForEach">
  <!-- For each property specified in
   GiftlistSearch.advancedSearchPropertyNames, retrieve all possible
   property values. This allows the customer
   to pick one to search on for advanced searching. -->
```

```
  <dsp:param bean="GiftlistSearch.propertyValuesByType" name="array"/>
  <dsp:oparam name="output">
  <dsp:droplet name="Switch">
    <dsp:param param="key" name="value"/>
    <dsp:oparam name="eventType">
    Event Type
      <!-- property to store the customer's selection is
           propertyValues -->
      <dsp:select bean="GiftlistSearch.propertyValues.eventType">
      <dsp:option value=""/>Any
      <dsp:setvalue paramvalue="element" param="outerelem"/>
      <dsp:droplet name="ForEach">
        <dsp:param param="outerelem" name="array"/>
        <dsp:oparam name="output">
        <dsp:option/><dsp:valueof param="element">UNDEFINED</dsp:valueof>
        </dsp:oparam>
      </dsp:droplet>
      </dsp:select><br>
    </dsp:oparam>
    <dsp:oparam name="eventName">
      <b>Event Name
      <!-- property to store the customer's selection is
           propertyValues -->
      <dsp:input bean="GiftlistSearch.propertyValues.eventName" size="30"
           value="" type="text"/> <br>
    </dsp:oparam>
    <dsp:oparam name="state">
      <b>State
      <!-- property to store the customer's selection is
           propertyValues -->
      <dsp:input bean="GiftlistSearch.propertyValues.state" size="30"
           value="" type="text"/> <br>
    </dsp:oparam>
  </dsp:oparam>
</dsp:droplet>

</dsp:droplet>
<p>
<dsp:input bean="GiftlistSearch.search" value="Perform Search"
     type="hidden"/>
<dsp:input bean="GiftlistSearch.search" value="Perform Search"
     type="submit"/>
</dsp:form>
```

Gift List Servlet Beans

Several servlet beans are provided to support gift list and wish list functionality. These servlet beans can be used with forms to look up gift lists and gift items, as well as to perform actions, such as removing or purchasing items from a gift list, adding gift lists to a profile, and removing gift lists from a profile.

Lookup Servlet Beans

The `GiftlistLookupDroplet` and `GiftitemLookupDroplet` servlet beans, located in Nucleus at `/atg/commerce/gifts/`, are instances of class `atg.repository.servlet.ItemLookupDroplet`. These servlet beans provide a way to search for and display gift lists and gift items in the `Giftlists` repository based on ID. For information about the input, output, and open parameters of servlet beans instantiated from `ItemLookupDroplet`, refer to *Appendix B: ATG Servlet Beans* in the *ATG Page Developer's Guide*.

The following code example demonstrates how to use the `GiftlistLookupDroplet` to look up a gift list in the repository and check that the owner ID equals the ID of the current profile before displaying the gift list.

```
<dsp:droplet name="/atg/commerce/gifts/GiftlistLookupDroplet">
 <dsp:param param="giftlistId" name="id"/>
 <dsp:oparam name="output">
  <dsp:droplet name="IsEmpty">
   <dsp:param param="element" name="value"/>
   <dsp:oparam name="false">
    <dsp:setvalue paramvalue="element" param="giftlist"/>
    <dsp:droplet name="/atg/dynamo/droplet/Switch">
     <dsp:param bean="Profile.id" name="value"/>
     <dsp:getvalueof var="ownerId" param="giftlist.owner.id"/>
       <dsp:oparam name="${ownerId}">
         <%-- display gift list info here --%>
       </dsp:oparam>
    </dsp:droplet>
   </dsp:oparam>
  </dsp:droplet>
 </dsp:oparam>
</dsp:droplet>
```

Note: In multisite environments, the `GiftlistLookupDroplet` should be used in conjunction with the `GiftlistSiteFilterDroplet` to ensure that only gift lists that are appropriate for the site context are displayed. For more details, see *Filtering Multisite Gift and Wish Lists* in the *ATG Commerce Guide to Setting Up a Store*.

GiftlistDroplet

The `GiftlistDroplet` servlet bean (class `atg.commerce.gifts.GiftlistDroplet`), which is located in Nucleus at `/atg/commerce/gifts/`, adds or removes customer A's gift list from customer B's `otherGiftlists` Profile property, depending on the action supplied via the `action` input parameter. This enables the given customer to easily find those for whom the customer has shopped or is shopping.

`GiftlistDroplet` takes the following input parameters:

- `action`: The action to perform on the gift list ("add" or "remove"). (Required)
- `giftlistId`: The ID of the gift list. (Required)
- `profile`: The profile of the current customer. If not passed, the profile will be resolved by Nucleus.

`GiftlistDroplet` doesn't set any output parameters. It renders the following open parameters (`oparams`):

- `output`: The `oparam` rendered if the gift list is added or removed successfully from a profile.
- `error`: The `oparam` rendered if an error occurs while adding or removing the gift list.

The following code example demonstrates how to use the `GiftlistDroplet` to add a gift list that was retrieved in a search to a customer's profile.

```
<dsp:droplet name="/atg/dynamo/droplet/IsEmpty">
  <dsp:param param="giftlistId" name="value"/>
  <dsp:oparam name="false">
    <dsp:droplet name="/atg/commerce/gifts/GiftlistDroplet">
      <dsp:param param="giftlistId" name="giftlistId"/>
      <dsp:param value="add" name="action"/>
      <dsp:param bean="/atg/userprofiling/Profile" name="profile"/>
```

```
      <dsp:oparam name="output">Output</dsp:oparam>
      <dsp:oparam name="error">Error</dsp:oparam>
    </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>
```

GiftitemDroplet

Servlet beans instantiated from class `atg.commerce.gifts.GiftitemDroplet` enable customers either to buy or to remove items from their own personal gift lists, depending on the configuration of the servlet bean. (For information on how to add items to a personal gift list, see GiftlistFormHandler (page 77). For information on how to buy items from another's gift list, see CartModifierFormHandler (page 90).)

Two Oracle ATG Web Commerce servlet beans have been instantiated from `GiftitemDroplet`; they are `BuyItemFromGiftlist` and `RemoveItemFromGiftlist`. They take the following input parameters, both of which are required:

- `giftId`: The ID of the gift.
- `giftlistId`: The ID of the gift list.

They don't set any output parameters. They render the following open parameters:

- `output`: The `oparam` rendered if the item is bought or removed successfully from list.
- `error`: The `oparam` rendered if an error occurs during processing.

The following code example demonstrates how to use the `RemoveItemFromGiftlist` component to remove an item from a customer's personal gift list.

```
<dsp:droplet name="/atg/dynamo/droplet/IsEmpty">
<dsp:param param="giftId" name="value"/>
<dsp:oparam name="false">
  <dsp:droplet name="/atg/commerce/gifts/RemoveItemFromGiftlist">
    <dsp:param param="giftlistId" name="giftlistId"/>
    <dsp:param param="giftId" name="giftId"/>
  </dsp:droplet>
</dsp:oparam>
</dsp:droplet>
```

Purchase Process Extensions to Support Gift Lists

The Oracle ATG Web Commerce purchase process supports features such as browsing the catalog, adding items to a shopping cart, selecting payment methods, entering billing and shipping information, and the entire checkout process.

To also support purchasing gifts off a gift list, components in the order package (`atg.commerce.order`) have been extended. To understand how gift lists work in Commerce, it is important to know which components in the order package support this functionality. This section describes how order management and the purchasing process have been extended to support gift lists. (See the *Configuring Purchase Process Services* (page 263) chapter for more information on the purchase process.)

The following classes provide gift list functionality to the purchase process:

- CartModifierFormHandler (page 90)
- GiftlistHandlingInstruction (page 91)

- ShippingGroupDroplet and ShippingGroupFormHandler (page 91)
- ProcUpdateGiftRepository (page 91)
- ProcSendGiftPurchasedMessage (page 91)

CartModifierFormHandler

`atg.commerce.order.purchase.CartModifierFormHandler` is one class that provides the functionality to support shopping carts and the purchase process in Oracle ATG Web Commerce. More importantly, the `addItemToOrder` method of `CartModifierFormHandler` provides support for purchasing items that appear on gift lists. `CartModifierFormHandler` and the `addItemToOrder` method are described in Understanding the CartModifierFormHandler (page 269) in the *Configuring Purchase Process Services* chapter.

Adding a gift item to an order is virtually the same as adding any item to an order. A gift item is distinguished by two additional form input fields, `CartModifierFormHandler.giftlistId` and `CartModifierFormHandler.giftlistItemId`. When these two fields contain non-null values, `addItemToOrder` calls `addGiftToOrder` in the `GiftListManager` component, which does additional processing for the gift as required by the purchase process.

The following code sample demonstrates how to use the `CartModifierFormHandler` in a template for gifts. It is an example of how to set up input and URL properties and make calls to handle methods in the form handler. In this example, a customer is purchasing an item from another customer's gift list and adding it to the shopping cart.

```
<dsp:droplet name="/atg/commerce/gifts/GiftlistLookupDroplet">
 <dsp:param param="giftlistId" name="id"/>
 <dsp:param value="giftlist" name="elementName"/>

 <dsp:oparam name="output">

   Get this gift for
   <dsp:valueof param="giftlist.owner.firstName">someone</dsp:valueof>
    <dsp:form action="giftlists.jsp" method="post">
     <dsp:input bean="CartModifierFormHandler.addItemToOrderSuccessURL"
                     value="../checkout/cart.jsp" type="hidden"/>
     <dsp:input bean="CartModifierFormHandler.productId"
                     paramvalue="Product.repositoryId" type="hidden"/>
     <dsp:input bean="CartModifierFormHandler.giftlistId"
                     paramvalue="giftlist.id" type="hidden"/>
     <dsp:input bean="CartModifierFormHandler.giftlistItemId"
                     paramvalue="giftid" type="hidden"/>
     <dsp:droplet name="/atg/commerce/gifts/GiftitemLookupDroplet">
       <dsp:param param="giftId" name="id"/>
       <dsp:param value="giftitem" name="elementName"/>
       <dsp:oparam name="output">
         <dsp:input bean="CartModifierFormhandler.catalogRefIds"
                     paramvalue="giftitem.catalogRefId" type="hidden"/>

         <dsp:valueof param="giftlist.owner.firstName">firstname</dsp:valueof>
          wants
         <dsp:valueof param="giftitem.quantityDesired">?</dsp:valueof>
         <dsp:valueof param="giftitem.catalogRefId">sku</dsp:valueof><br>
           and so far people have bought
         <dsp:valueof param="giftitem.quantityPurchased">?</dsp:valueof>.
         <p>
       </dsp:oparam>
     </dsp:droplet>
```

```
      Buy <dsp:input bean="CartModifierFormHandler.quantity" size="4"
               value="1" type="text"/>
      <dsp:input bean="CartModifierFormHandler.addItemToOrder"
           value="Add to Cart" type="submit"/>
     </dsp:form>

  </dsp:oparam>
</dsp:droplet>
```

GiftlistHandlingInstruction

The `GiftlistHandlingInstruction` specifies what special handling needs to be done for a gift. For example, it could update gift list information to reflect that the item was purchased. A separate `GiftlistHandlingInstruction` could indicate that the gift should be wrapped.

When a person purchases an item off a gift list, `CartModifierFormHandler` calls `addGiftToOrder` in `GiftlistManager`. `addGiftToOrder` performs additional tasks to adding the item to an order. These tasks include:

- Creating a new shipping group with the recipient's address
- Adding that item(s) to the new shipping group
- Creating a `GiftlistHandlingInstruction` object for the item

For more information on Handling Instructions, see Setting Handling Instructions (page 252) in the *Configuring Purchase Process Services* chapter.

ShippingGroupDroplet and ShippingGroupFormHandler

Your sites may use `ShippingGroupDroplet` and `ShippingGroupFormHandler` to pull shipping information from the user's profile and to allow the user to assign shipping addresses to items in an order. Both of these components have been extended for gift lists so that shipping information for gift items is automatically preserved. See *Adding Shipping Information to Shopping Carts* in the *Implementing Shopping Carts* chapter of the *ATG Commerce Guide to Setting Up a Store* and Preparing a Complex Order for Checkout (page 282) in the *Configuring Purchase Process Services* chapter for descriptions of these components.

ProcUpdateGiftRepository

A pipeline is an execution mechanism that allows for modular code execution (see the *Processor Chains and the Pipeline Manager* (page 363) chapter for details). The `ProcUpdateGiftRepository` processor is added to the `processOrder` chain to support gift lists. It goes through the order and looks for a gift that has been added to the shopping cart. If one has been added, it updates the `Giftlists` repository to update the purchased count of items off the gift list.

ProcSendGiftPurchasedMessage

The `ProcSendGiftPurchasedMessage` processor is added to the `processOrder` chain. It goes through the order and looks for a gift that has been purchased. If one has been purchased, it sends a message that contains the order, gift, and recipient profile to the Scenarios module. This message can be used to trigger an event such as sending an e-mail message to the recipient. In multisite environments, the message also includes the ID of the site the gift was purchased from, so that the e-mail message may be properly branded.

Gift and Wish Lists in a Multisite Environment

This section provides information specific to working with gift and wish lists in a multisite environment.

Notes:

- As you read this section, keep in mind that, when a customer adds a SKU to a gift list, Oracle ATG Web Commerce creates a gift item based on that SKU, of item type `gift-item`, and then adds the gift item to the gift list.
- This section assumes you are familiar with sharing groups and shareable types, which are the foundation for shared data in a multisite environment. For more information on these concepts, refer to the *ATG Multisite Administration Guide*.

Using Gift and Wish Lists in a Multisite Environment

Gift list accessibility in a multisite environment typically falls into one of these categories:

- Gift lists are universal and accessible by all sites. Microsite environments, where a microsite shares most resources with its parent site, typically use universal gift lists.
- Gift list access is limited based on site context. This is a common approach for affiliated site environments where sites share profiles and shopping carts but not catalogs. For example, Oracle ATG Web Commerce Reference Store aligns gift list access with shopping cart access, so customers may only see gift lists that they are allowed to purchase items from.

  Note: See the *ATG Multisite Administration Guide* for more information on microsites and affiliated sites.

Wish lists, by contrast, are always universal because there is only one wish list per profile and all sites must be able to access it. As such, a wish list may contain items from multiple sites, making the ability to filter those items based on site context a necessity.

Gift Item and Gift List Site IDs

When the `GiftlistManager` creates either a `gift-list` or `gift-item` item in the gift lists repository, it includes a `siteId` property that represents the site the new item is affiliated with. Adding `siteId`'s to both gift lists and gift items allows Nucleus components to evaluate whether a gift list or a gift item should be included in a gift list operation, based on site context (for example, which items can be added to gift lists associated with site X, which gift lists should Nucleus return for a gift list search initiated from site X, and so on). Inclusion or exclusion is determined by the gift list site scope, described later in this section.

Gift list site IDs correspond to the site the customer was on when the gift list was created. A gift item, on the other hand, represents a `SKU` or `CommerceItem` that may be affiliated with multiple sites, so a gift item's `siteId` is not as straightforward. By default, the ID of the current site is used, however, you can override the default by doing the following:

- Use the `atg.droplet.multisite.SiteIdForItemDroplet` servlet bean to determine an appropriate site ID for the item.
- Set the `GiftlistFormHandler.siteId` property to the site ID returned by `SiteIdForItemDroplet` before calling any of the handle methods that save, update, or add a gift item to a gift list.

Notes:

- Gift lists and gift items with null site IDs are a special case; see Gift Lists and Gift Items with a Null Site ID (page 95) for more information.
- See *ATG Page Developer's Guide* for more information on the `SiteIdForItemDroplet`.

Gift List Site Scope

The `GiftlistManager` component has a `siteScope` property that controls whether gift list activity is limited to the current site, limited to sites in a sharing group, or not limited at all. Effectively, the `siteScope` property

determines what items can be added to a gift list, what gift lists should be returned by a search, and what gift items should be filtered out based on site context. There are three possible values:

- `all`—Gift list activity is not limited. This is the default.
- `current`—Gift list activity is limited to the specified sites or, if none have been specified, the current site.
- S*`hareableType-ID`*—Gift list activity is limited to the specified sites, or the current site if none are specified, and any sites that are in a sharing group with those sites, as defined by the `ShareableType` ID. For example, setting `GiftlistManager.siteScope` to the `ShoppingCartShareableType` component's ID will limit gift list activity to the specified sites, or the current site if none are specified, and any sites that share a shopping cart with those sites. For information on where to find `ShareableType` IDs, see the *ATG Multisite Administration Guide*.

Other components, described later in this section, use the `GiftlistManager.siteScope` property to do their work. These components accept an optional list of site IDs (the aforementioned "specified sites") and use that list, in conjunction with the `GiftlistManager.siteScope` property to determine which sites should included in the operation.

Continue with the following sections to see in detail how site scope affects adding gift items to, searching for, and filtering gift lists.

Adding Gift Items to Gift Lists in a Multisite Environment

Customers can add a gift item to a gift list either by adding the item from a product detail page, or by moving the item from their shopping cart to a gift list. In either case, the `GiftlistManager` must determine whether the gift item and the gift list are compatible, from a `siteId` perspective, before performing the add operation. To determine compatibility, the `GiftlistManager.addItemsToGiftlist()` method consults the `GiftlistManager.siteScope` property and then compares the `siteId` properties of the gift item and the gift list to determine if the sites are compatible, as shown in the following table:

Compatibility Test	All	Current	ShareableType ID
Gift item and gift list have the same `siteId`	Add gift item to gift list	Add gift item to gift list	Add gift item to gift list
Gift item and gift list have different `siteId`'s	Add gift item to gift list	Don't add gift item to gift list	Add gift item to gift list if both sites are in the same sharing group
Gift list has a null `siteId` (i.e., it is a wish list)	Add gift item to gift list	Add gift item to gift list	Add gift item to gift list

As previously mentioned, adding a gift item to a gift list is a two-step process:

1. Create a gift item, based on an existing `SKU` item or a `CommerceItem` in the shopping cart, including the item's `siteId`.
2. Add the gift item to the gift list.

It's during this second step that the `GiftlistManager` evaluates the gift list and gift item's `siteId` properties for compatibility. If they are determined to be compatible, the gift item is added to the gift list.

Searching for Gift Lists in a Multisite Environment

A key aspect of gift list functionality is the ability to allow customers to search for another customer's gift lists. Similar to the process for adding a gift list, when the `GiftlistSearch` form handler searches for gift lists, it must determine which gift lists not only match the search criteria but are also appropriate to include in the search results from a site context perspective. To support searching for gift lists in a multisite environment, the `GiftlistSearch` form handler includes the following multisite-specific properties:

- `giftlistManager`: Reference to the gift list manager component `/atg/commerce/gifts/GiftlistManager`.
- `siteGroupManager`: Reference to the site group manager component `/atg/multisite/SiteGroupManager`. This component determines which sites are part of the same sharing group and can share data such as gift lists.
- `siteScope`: Controls the scope of the gift list search, as described in Gift List Site Scope (page 92) above. By default, this property is set to the `GiftlistManager` component's `siteScope` property, however, you can override that setting by specifying a different scope (`all`, `current`, or *`shareableType-ID`*) in the `GiftlistSearch` form handler's `siteScope` property.
- `siteIds`: This property, typically set by the JSP page, may contain an array of `siteId`'s. This set of IDs, in conjunction with `siteScope` setting, defines the set of sites whose gift lists should be included in the search results. If no `siteId`'s are passed to the form handler, the current site's `siteId` is used. The following example shows how to set the `siteIds` array in a JSP page so that it contains three sites:

```
<dsp:input bean="GiftlistSearch.siteIds" type="hidden" value="siteId1"/>
<dsp:input bean="GiftlistSearch.siteIds" type="hidden" value="siteId2"/>
<dsp:input bean="GiftlistSearch.siteIds" type="hidden" value="siteId3"/>
```

When searching for gift lists, the `GiftlistSearch` form handler uses its `siteScope` and `siteIds` properties to apply constraints on the search query so that only gift lists that match the search terms and have compatible site IDs are returned, as shown in the following table:

Compatibility Test	All	Current	ShareableType ID
Gift list's `siteId` is in the `siteIds` array	Include gift list in search results	Include gift list in search results	Include gift list in search results
Gift list's `siteId` is not in the `siteIds` array	Include gift list in search results	Do not include gift list in search results	Include gift list in search results if the gift list's `siteId` is in the specified sharing group (for example, the shopping cart sharing group) with any of the sites in the `siteIds` array

Filtering Gift Lists

Oracle ATG Web Commerce includes functionality that allows you to filter collections of gift lists and gift items so that you display only those lists/items that are appropriate for the customer's site context. In a multisite environment, any time you retrieve a collection of gift lists or gift items by referring to a repository item's property, such as `Profile.giftlists` or `Profile.wishlist.giftlistItems`, you get back an unfiltered list that may contain items from multiple sites. For these situations, you should consider whether the collection should be filtered or not and, if so, implement gift list filtering functionality. For detailed information on this functionality, see *Filtering Multisite Gift and Wish Lists* in the *ATG Commerce Guide to Setting Up a Store*.

Gift Lists and Gift Items with a Null Site ID

In non-multisite environments, there is no site context, so the `GiftlistManager` sets the `siteId` properties for new gift lists and gift items to null. If a non-multisite environment is reconfigured to be multisite aware, gift lists and gift items with a null `siteId` are considered to be universal and all sites in the environment may manipulate them.

Wish lists must be accessible by all sites, so they always have a null site ID in both multisite and non-multisite environments.

Extending Gift List Functionality

The Oracle ATG Web Commerce implementation of gift lists supports most of the requirements for this feature for a typical commerce site. However, gift list functionality can be extended, if necessary.

This section describes how to extend gift list functionality by adding additional item properties to the `gift-list` item descriptor. The process includes the following basic steps:

- Updating the Database and Repository Definition (page 95)
- Extending GiftlistFormHandler (page 96)

Updating the Database and Repository Definition

You can extend your sites' gift list functionality by adding new `gift-list` item properties. To add new properties, do the following:

1. Modify your database schema and update the database script included with Oracle ATG Web Commerce. The script for gift lists is found in `<ATG10dir>/DCS/sql/db_components/`*`database_vendor/`*`user_giftlist_ddl.sql`.
2. Extend the repository definition for gift lists by layering on a `giftlists.xml` file at `/atg/commerce/gifts/` in your local configuration directory. This new file should add your new property to the `gift-list` item descriptor.

   As an example, the following XML example demonstrates how you might add the property `giftlistStatus` to the `gift-list` item descriptor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE taglib
  PUBLIC "-//Art Technology Group, Inc.//DTD General SQL Adapter//EN"
  "http://www.atg.com/dtds/gsa/gsa_1.0.dtd">

<gsa-template>
  <header>
    <name>Commerce Giftlists</name>
    <author>Company XYZ</author>
    <version>$Id$</version>
  </header>
  <item-descriptor name="gift-list">
    <table name="dcs_giftlist">
      <property name="giftlistStatus" data-type="timestamp"
                column-name="giftlist_status"/>
    </table>
  </item-descriptor>
</gsa-template>
```

Note: You should only need to add to the schema and repository definition. Removing anything that is already there requires substantially more work.

Extending GiftlistFormHandler

After Updating the Database and Repository Definition (page 95) to support additional `gift-list` item properties, you also need to extend the `GiftlistFormHandler` component to support them. To do so, do the following:

1. Extend the class `atg.commerce.gifts.GiftlistFormHandler` to support your new properties and override related methods. The source code for this form handler can be found in `<ATG10dir>/DCS/src/Java/atg/commerce/gifts`.

   Note that most handlers in the `GiftlistFormHandler` have preXXX and postXXX methods that can be overridden to support your requirements. To support your new property, you should override the `postSaveGiftlist` method to save your new property value in the repository.

2. Layer on a `GiftlistFormHandler.properties` file to configure an instance of the new class you created in step 1. The contents of the configuration file would look similar to the following:

   ```
   # MyNewGiftlistFormHandler
   #
   $class=xyz.commerce.gifts.MyNewGiftlistFormHandler
   ```

Disabling the Gift List Repository

If you are not going to use the `Giftlists` repository, you can disable it. This repository is represented by the `/atg/commerce/gifts/Giftlist` component. Disabling this repository prevents you from having to create the associated tables in your database.

Perform the following steps to disable the `Giftlists` repository:

1. Edit the `/atg/registry/ContentRepositories` component and remove the value in the `initialRepositories` property that references the `/atg/commerce/gifts/Giftlist` component. For more information, see the *Configuring the SQL Repository Component* section of the *SQL Repository Reference* chapter in the *ATG Repository Guide*.

2. Edit the `/atg/commerce/gifts/GiftlistTools` component and set the `giftlistRepository` property to null.

3. Optionally, edit the `userProfile.xml` file and remove properties that reference items in the `Giftlists` repository. The `userProfile.xml` file is located in the Commerce configuration layer. Remove the following lines:

   ```
   <!-- key into private wishlist -->
   <table name="dcs_user_wishlist" type="auxiliary"
   id-column-name="user_id">
   <property category="Commerce - Lists" name="wishlist"
   item-type="gift-list"
   repository="/atg/commerce/gifts/Giftlists"
   column-name="giftlist_id"cascade="insert,update,delete"
   display-name="Wish list"/>
   </table>
   <!-- key into user created giftlists -->
   <table name="dcs_user_giftlist" type="multi"
   id-column-name="user_id" multi-column-name="sequence_num">
   ```

```
<property category="Commerce - Lists" name="giftlists"
data-type="list" component-item-type="gift-list"
repository="/atg/commerce/gifts/Giftlists"
column-name="giftlist_id" display-name="Gift list"/>
</table>
<!-- key into giftlists found for other customers -->
<table name="dcs_user_otherlist" type="multi"
id-column-name="user_id" multi-column-name="sequence_num">
<property category="Commerce - Lists" name="otherGiftlists"
data-type="list" component-item-type="gift-list"
repository="/atg/commerce/gifts/Giftlists"
column-name="giftlist_id"
display-name="Other gift lists"/>
</table>
```

4. Remove the `updateGiftRepository` and `sendGiftPurchased` processors from the commerce pipeline. These processors are found in the `processOrder` pipeline chain. After these entries have been removed from the chain definition, create a link from the `authorizePayment` processor to the `addOrderToRepository` processor.

Setting Up Product Comparison Lists

Commerce sites often require the ability for a user to compare items in the product catalog. A simple site may offer the user a single product comparison list. A more complex site may offer the user multiple comparison lists to compare different types of items. A multisite configuration may provide the ability to compare items across sites.

The default implementation of the product comparison system enables the user to compare any number of products and to do so using the products' category, product, SKU, and inventory information. (You can extend the system to include additional information.) Additionally, it enables the page developer to display product comparison information as a sortable table, which the user can manipulate to change the sort criteria for the displayed information.

This section describes the default implementation of the product comparison system and includes the following subsections:

- Understanding the Product Comparison System (page 97)
- Extending the Product Comparison System (page 105)
- Using TableInfo to Display a Product Comparison List (page 106)

Understanding the Product Comparison System

The product comparison system consists of the following four classes in the `atg.commerce.catalog.comparison` package:

- ComparisonList (page 98)

  A class that provides a generic data structure to maintain an ordered list of objects and an associated set of sort directives to apply when displaying the items.

- ProductComparisonList (page 99)

  A subclass of `ComparisonList` that provides an extended API for comparing product information. Oracle ATG Web Commerce includes a session-scoped instance of `ProductComparisonList`, located in Nucleus at `/atg/commerce/catalog/comparison/ProductList`.

- ProductListContains (page 103)

  A droplet that queries whether a product comparison list contains an entry for a specific product. Oracle ATG Web Commerce includes a globally-scoped instance of `ProductListContains`, located in Nucleus at `/atg/commerce/catalog/comparison/ProductListContains`.

- ProductListHandler (page 103)

  A form handler that manages product comparison lists. Commerce includes a session-scoped instance of `ProductListHandler`, located in Nucleus at `/atg/commerce/catalog/comparison/ProductListHandler`.

ComparisonList

`atg.commerce.catalog.comparison.ComparisonList` provides a generic data structure to maintain an ordered list of items that the user may want to compare, as well as an associated set of sort directives to apply when displaying the items in the list. The objects in the list may be of any Java class. Like Java `List` classes, `ComparisonList` maintains the insertion order of items in the list. Unlike `List` classes, it prohibits duplicate entries in the list by ignoring requests to add items that compare equal to items already in the list.

The following table describes the `ComparisonList` methods used to maintain a list of items to compare. For additional methods and details, refer to the *ATG Platform API Reference*.

Method	Description
`addItem`	Adds an item to the end of the comparison list if the item isn't already present in the list.
`clear`	Removes all items from the comparison list.
`containsItem`	Returns true if the comparison list contains the specified item.
`getItem(n)`	Returns the item at the specified index in the comparison list.
`getItems`	Returns the list of items being compared.
`indexOf`	Returns the index of the specified item in the comparison list; returns -1 if the item does not appear in the list.
`removeItem`	Removes an item from the comparison list if it was present in the list.
`Size`	Returns the number of items in the comparison list.

`ComparisonList` internally synchronizes operations on the list. This makes it possible for multiple request-scoped servlet beans and form handlers to operate safely on a shared session-scoped `ComparisonList`, as long as all changes to the list are made through the `ComparisonList` API. Note that if your application calls the `getItems` method to obtain a reference to the list, you should synchronize all operations on the list or call the

`java.util.Collections.synchronizedList` method to obtain a thread-safe version of the list upon which to operate.

`ComparisonList` maintains a property of type `atg.service.util.TableInfo` as a convenience to the developer. In cases where the comparison information will be displayed as a table, this provides an easy way to associate default table display properties with a comparison list. Oracle ATG Web Commerce includes a session-scoped instance of `TableInfo`, located in Nucleus at `/atg/commerce/catalog/comparison/TableInfo`. For more information about the `TableInfo` component and how to use it to display sortable tables, see the *Implementing Sortable Tables* chapter in the *ATG Page Developer's Guide*.

Use `ComparisonList` when you want to compare sets of simple Java beans, repository items, user profiles, or other self-contained objects. If you want to compare more complex objects, or sets of objects against each other, you'll want to subclass `ComparisonList` to be able to manage application-specific objects.

ProductComparisonList

`ProductComparisonList` extends `ComparisonList`, providing an API designed to manage and compare products and SKUs. `ProductComparisonList` uses the `items` property to store a list of `Entry` objects, each of which represents a product or SKU that the user has added to her product comparison list. `Entry` is an inner class defined by `ProductComparisonList`; it combines category, product, SKU, and inventory information about a product into a single object.

You can configure additional instances of `ProductComparisonList` in Nucleus to provide multiple comparison lists.

The API for ProductComparisonList

The public API for `ProductComparisonList` can be divided broadly into the following four categories:

- **add methods**, which add entries to the list.

  When you call `ProductComparisonList`'s `add` method, a new `Entry` object is automatically constructed and added to the item list if it is not already present. When you call `ProductComparisonList`'s `addAllSkus` method, a new `Entry` object for each SKU associated with the given product is automatically constructed and added to the item list.

  When the `add` method or `addAllSkus` method is called, if no category ID for the given product is specified, then the product's default parent category is used. If no default parent category for the given product is set, then the `category` property of the new `Entry` object is null. Similarly, if no SKU is specified in the method call, then the given product's first child SKU is used. If the product has no child SKUs, then the `sku` property of the new `Entry` object is null.

- **remove methods**, which remove entries from the list.

- **contains methods**, which query whether the list contains an entry matching specified product, category, SKU, or site information.

- **set and get methods**, which set and get various properties of the `ProductComparisonList` itself.

  When you call `getItems` on a `ProductComparisonList`, you get back a List of `Entry` objects. When working with these objects in Java, you can either cast the objects to `ProductComparisonList.Entry` or use the `DynamicBeans` system to retrieve the product, category, SKU, site, and inventory information from the `Entry`. When working with these objects in JSPs, you can refer to their properties in the same way you refer to the properties of any other Java bean.

- **refresh methods**, which refresh the inventory information for the items in the `ProductComparisonList`.

  The `refreshInventoryData()` method iterates over the items in the `ProductComparisonList` and loads updated inventory information into them.

The `setRefreshInventoryData(String unused)` method calls the `refreshInventoryData()` method. This method enables you to update the inventory information for the items when you render the page that displays the `ProductComparisonList`. To do so, you could use the following `setvalue` tag at the top of the page:

```
<setvalue bean="ProductList.refreshInventoryData">
```

For related form handler methods, see ProductListHandler (page 103).

There are several different variations on the `add`, `remove`, and `contains` methods. The various methods take different sets of arguments to support a wide range of application behaviors. For example, there are `remove` methods to remove all entries for a specific product, to remove all entries for all products in a specified category, and to remove the entry for a particular category/product/SKU combination.

Additionally, several methods of `ProductComparisonList` take an optional `catalogKey` parameter. This `String` parameter is useful for applications using catalog localization because it enables you to specify the product catalog to use when operating on a product comparison list. Through the `catalogKey` parameter, you pass a key to `CatalogTools`, which then uses the given key and its key-to-catalog mapping to select a product catalog repository.

Refer to the *ATG Platform API Reference* for additional information on the public API for `ProductComparisonList`. Also note that there is one important protected method:

```
protected Entry createListEntry(RepositoryItem pCategory,
RepositoryItem pProduct, RepositoryItem pSku)
```

The `createListEntry` method is called to create a new list entry with a given category, product, and SKU. By subclassing `ProductComparisonList` and overriding `createListEntry`, you can extend or replace the properties of the `Entry` object. See Extending the Product Comparison System (page 105) for more information.

The Entry Inner Class

The public API for the `Entry` class exposes properties that the page developer can display in a product comparison list or table. The default implementation includes the following properties:

Property Name	Property Type	Description
`product`	`RepositoryItem`	The product being compared.
`category`	`RepositoryItem`	The category of the product being compared. If the category is not set explicitly when the product is added to the list, then the product's default parent category is used. If the product's default parent category is unset, the `category` property is null.
`sku`	`RepositoryItem`	The product's SKU. If the SKU is not set explicitly when the product is added to the list, then the first SKU in the product's `childSkus` list is used. If the product has no child SKUs, then the `sku` property is null.

Property Name	Property Type	Description
`inventoryInfo`	`InventoryData`	The `InventoryData` object that describes the inventory status for the given product and SKU. If the `sku` property is null or the inventory information isn't available, then the `inventoryInfo` property is null. (See the next section for more information on the `InventoryData` object.)

`productLink`	`String`	An HTML fragment that specifies an anchor tag that links to the product's page in the catalog. The default format for the link is `<a href="product.template.url?id=product.repositoryId">product.displayName</a>`.
		
		If you are using Oracle ATG Web Commerce's multisite feature, the `ProductComparisonList` automatically uses the `SiteURLManager` to find the base production URL for the site from which the entry was added.
		
		You can change the link format by setting the `ProductComparisonList.productLinkFormat` property.
		
		Note: If you display the product comparison information in a table, you can use the `productLink` property in the configuration of the `TableInfo` object that maintains the table information, as in the following example:
		
		`columns=\`
		`Product Name=productLink,\`
		`Price=sku.listPrice,\`
		…
		
		Or, similarly, to display the product link in a table column but sort the column on the product's display name, you could modify the example in the following manner:
		
		`columns=\`
		`Product Name=productLink;`
		`product.displayName,\`
		`Price=sku.listPrice,\`
		…
		
		For more information on the `TableInfo` component, see the *Implementing Sortable Tables* chapter in the *ATG Page Developer's Guide*.

Property Name	Property Type	Description
`categoryLink`	`String`	An HTML fragment that specifies an anchor tag that links to the category's page in the catalog. The default format for the link is `<a href=category.template.url?id=category.repositoryId>category.displayName</a>`. However, you can change the format by setting the `ProductComparisonList.categoryLinkFormat` property. **Note:** Like the `productLink` property, the `categoryLink` property can be used in the configuration of a `TableInfo` component. See the description of the `productLink` property in this table for more information. Also like `productLink`, if you are using Oracle ATG Web Commerce's multisite feature, the `ProductComparisonList` automatically uses the `SiteURLManager` to find the base production URL for the site from which the entry was added.
`id`	`Int`	A unique ID that names the list entry. You can use this property to retrieve individual entries by calling `ProductComparisonList.getItems(id)` in Java code or by using `<dsp:valueof bean="ProductList.entries[id]"/>` in a `.jsp` page. You can also use this property to delete specific entries, for example, with a form handler. For a JSP example, refer to *Examples of Product Comparison Pages* in the *Implementing Product Comparison* chapter of the *ATG Commerce Guide to Setting Up a Store*.
`siteId`	`String`	If you are using Oracle ATG Web Commerce's multisite feature, this property holds the ID of the site with which the item is associated.

A page developer can refer to the properties of the `Entry` objects using familiar JSP syntax, as in the following example:

```
<dsp:droplet name="ForEach">
  <dsp:param bean="ProductComparisonList.items" name="array"/>

  <dsp:oparam name="output">
    <p>Product Name: <dsp:valueof
param="element.product.displayName"/><br>
       Category: <dsp:valueof param="element.category.displayName"/><br>
       Inventory: <dsp:valueof
param="element.inventoryInfo.inventoryAvailabilityMsg"/><br>
  </dsp:oparam>

</dsp:droplet>
```

The InventoryData Inner Class

The `getInventoryInfo()` method of the `Entry` inner class (class `atg.commerce.catalog.comparison.ProductComparisonList.Entry`) returns an instance of the `InventoryData` inner class (class `atg.commerce.catalog.comparison.ProductComparisonList.Entry.InventoryData`).

The `InventoryData` object stores the inventory data for a given item in the product comparison list. It returns a subset of the readable properties of an `InventoryInfo` object (class `atg.commerce.inventory.InventoryInfo`). However, unlike an `InventoryInfo` object, an `InventoryData` object is serializable, which enables it to participate in session failover. For a list of `InventoryData` properties, refer to the *ATG Platform API Reference*.

ProductListContains

When given a category, product, and SKU, the `ProductListContains` droplet queries whether a product comparison list includes the given item.

The behavior of `ProductListContains` parallels that of `ProductComparisonList`. Namely, you can specify a product ID with or without an accompanying category or SKU. In the latter situation, `ProductListContains` behaves as follows:

- If you don't specify a category ID for the given product, then `ProductListContains` looks for a list entry whose `category` property matches either the given product's default category or null if there is no default category for the given product.
- If you don't specify a SKU for the given product, then `ProductListContains` looks for a list entry whose `sku` property matches either the given product's first child SKU or null if there are no SKUs for the given product.

For a list of the input, output, and open parameters for `ProductListContains`, and for JSP examples of how page developers can use `ProductListContains`, refer to the *Implementing Product Comparison* chapter of the *ATG Commerce Guide to Setting Up a Store*.

ProductListHandler

The `ProductListHandler` form handler manages product comparison lists. By default, Oracle ATG Web Commerce includes a session-scoped instance of `ProductListHandler`, located in Nucleus at `/atg/commerce/catalog/comparison/ProductListHandler`. It is configured to operate on the product comparison list located at `/atg/commerce/catalog/comparison/ProductList`.

If your application uses multiple instances of `ProductComparisonList` to manage multiple product comparison lists, then you may want to configure multiple instances of `ProductListHandler` to manage the contents of each list.

If your application uses alternate product catalogs for different locales, you can specify the product catalog to use when operating on a product comparison list. To do so, set the `ProductListHandler.repositoryKey` property to the key to pass to `CatalogTools`. `CatalogTools` uses the given key and its key-to-catalog mapping to select a product catalog repository. Typically, you would set the `ProductListHandler.repositoryKey` property via a hidden input field in a form. If the `repositoryKey` property is unset, then the default product catalog repository is used.

The following table describes `ProductListHandler`'s handle methods for managing a product comparison list:

Handle Method	Description
handleAddProduct	Adds the product specified by productID to the product comparison list, applying optional category, SKU, and site information if supplied in categoryID, skuID, and siteID.
handleAddProductAllSkus	Adds all of the SKUs for the product specified by productID to the product comparison list, applying optional category and site information if supplied in categoryID and siteID.
handleAddProductList	Adds all of the products specified by productIDList to the product comparison list, applying optional category and site information if supplied in categoryID, siteID, and the default SKU for each product, if any.
handleAddProductListAllSkus	Adds all of the SKUs for all of the products specified by productIDList to the product comparison list, applying optional category and site information if supplied in categoryID and siteID.
handleCancel	Resets the form handler by setting productID, categoryID, skuID, and siteID to null.
handleClearList	Clears the product comparison list.
handleRefreshInventoryData	Calls the ProductComparisonList.refreshInventoryData() method. The handleRefreshInventoryData method also calls pre and post methods. If necessary, your subclasses can override these methods to provide additional application-specific processing. Also note that ProductListHandler has two optional properties, refreshInventoryDataSuccessURL and refreshInventoryDataErrorURL, which you can set to redirect the user when the handle method succeeds or fails, respectively.
handleRemoveCategory	Removes all entries for the category specified by categoryID and siteID from the product comparison list.
handleRemoveEntries	Removes the list entries whose ids are specified in entryIds from the product comparison list.
handleRemoveProduct	Removes the product specified by productID from the product comparison list, applying the optional category, SKU, and site information if supplied in categoryID, skuID, and siteID respectively.
handleRemoveProductAllSkus	Removes all entries for the product specified by productID and siteID from the product comparison list.

Handle Method	Description
handleSetProductList	Sets the product comparison list to the products specified by productIDList, applying optional category information if supplied in categoryID and the default SKU for each product, if any.
handleSetProductListAllSkus	Sets the product comparison list to contain all the SKUs for all the products specified by productIDList, applying optional category information if supplied in categoryID.

The behavior of ProductListHander's handle methods parallels that of ProductComparisonList. Namely, optional category and SKU information is managed in the same way. If a product's category information isn't specified in categoryID, then the form handler looks for the default category of the product. If no default value exists, then the property is set to null. Similarly, if a product's SKU information isn't specified in skuID, then the form handler looks for the product's first child SKU (that is, the default SKU). If no default value exists, then the property is set to null.

For additional information on ProductListHandler's methods, refer to the *ATG Platform API Reference*. For examples of how page developers can use ProductListHandler in JSPs to manage product comparison lists, refer to the *Implementing Product Comparison* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Using Product Comparison Lists in a Multisite Environment

If you are using Oracle ATG Web Commerce's multisite feature, you may want to provide users with the ability to compare products across multiple sites. You do not need to do any additional configuration to use this feature; the ProductComparisonList is registered as a shareable component by default and works the same way in a multisite environment as in a single site. The ProductComparisonList, ProductComparisonList.Entry, and ProductListContains classes are all site-aware by default.

Note: The product comparison list does not prevent users from adding the same product to a list from different sites.

The shareable Nucleus component that refers to the ProductComparisonList is located at /atg/commerce/ShoppingCartShareableType. By default, the ProductComparisonList is registered as a shareable component:

```
id=atg.ShoppingCart
paths=/atg/commerce/ShoppingCart,\
    /atg/commerce/catalog/comparison/ProductList
```

See the *ATG Multisite Administration Guide* for information on shareable components and how to use sharing groups in your multisite configuration.

Extending the Product Comparison System

As previously described, the Entry object maintains category, product, SKU, site, and inventory information in a single object. ProductComparisonList maintains a list of Entry objects, with each Entry object representing a product that the user has added to her product comparison list.

You can subclass `ProductComparisonList` and override its `createListEntry` method to extend the information stored in an `Entry` object. Although the `Entry` class contains convenience methods for setting and getting properties like `product`, `category`, `sku`, `siteId`, and `inventoryInfo`, your subclasses don't need to provide similar methods for their own properties. Because `Entry` is a subclass of `java.util.HashMap`, you can call `Entry.put(name, value)` to add a new property value to the `Entry` object. The following code example illustrates this point; it stores a hypothetical value called "popularity," which indicates how popular a given product is.

```
public class MyProductComparisonList extends ProductComparisonList
  {
    protected Entry createListEntry(RepositoryItem pCategory,
                                    RepositoryItem pProduct,
                                    RepositoryItem pSku),
                                    RepositoryItem pSiteId)
{
      Entry e = super.createListEntry(pCategory, pProduct, pSku, pSiteId);
      int score = computePopularity(pProduct);
      e.put("popularity", score);
      return e;
    }

    public int computePopularity(RepositoryItem pProduct) {
      ...
    }
  }
```

Note that by the time the `createListEntry` method is called, `pCategory` and `pSku` will have been populated with the product's default parent category and first child SKU, if necessary and available. Consequently, `createListEntry` is called with the same category and SKU values that the user ultimately sees on the page in the product comparison list.

Page developers can refer to the `popularity` property of a `ProductComparisonList` entry to display the corresponding value in a JSP.

Using TableInfo to Display a Product Comparison List

The `ProductList` component, which maintains the list of `Entry` objects in its `items` property, also includes a reference to a `TableInfo` object in its `tableInfo` property. The `TableInfo` component maintains the display information to compare the products in table form, such as the properties to display in the table, the column headings for the table, and the sorting instructions for the table.

Depending on the complexity of your commerce application, you may require multiple instances of `ProductComparisonList` and `TableInfo`. In general, however, an application will maintain one instance of `ProductComparisonList` and one instance of `TableInfo` for each comparison table desired.

For detailed information on `TableInfo`, refer to the *Implementing Sortable Tables* chapter in the *ATG Page Developer's Guide*. For JSP examples of how to use the `ProductList` and `TableInfo` components to manage product comparisons, refer to the *Implementing Product Comparison* chapter of the *ATG Commerce Guide to Setting Up a Store*. For an example of using multiple instances of `TableInfo` to manage a single product comparison table, refer to the *Viewing Compare Results* section of the *Displaying and Accessing the Product Catalog* chapter in the *ATG Business Commerce Reference Application Guide*.

Setting Up Gift Certificates and Coupons

Oracle ATG Web Commerce provides the ability to create and manage gift certificates and coupons. (Collectively, these are sometimes referred to as "claimable items" in Commerce.) By providing gift certificates as an option for your customers, you can increase sales and attract new business. Coupons can also help increase your customer base (for example, by including them in a cold call e-mail campaign), but more importantly they provide an alternative way to deliver promotions to your existing customers.

If you are using Oracle ATG Web Commerce's multisite feature, you can specify at the time of coupon creation whether users can claim the coupon on any site, or only sites to which the associated promotion is limited. See the *ATG Merchandising Guide for Business Users* for information.

The claimable item system in Commerce is made up of three components:

- The `Claimable` repository
- The `ClaimableTools` component
- The `ClaimableManager` component

The Claimable Repository

The `Claimable` repository holds claimable items, namely, gift certificates and coupons . The repository itself is made up of two parts: the database schema and the XML repository definition file. The definition file represents an item that can be claimed (a sub-type of type `Claimable`) and then defines specific implementations of this item.

The following example shows the pertinent code from the `Claimable` repository definition file:

```
<item-descriptor name="claimable" sub-type-property="type"
                 version-property="version">
    ...
    <property name="type" data-type="enumerated">
      <option value="GiftCertificateClaimable"/>
      <option value="PromotionClaimable"/>
      <option value="CouponBatch"/>
      <option value="BatchPromotionClaimable"/>
    </property>
   ...
</item-descriptor>
```

Each item in the `Claimable` repository has a `repositoryId` property. The system uses the value in this property as the key for claiming the item (for example, as the claim code for a gift certificate). The value is created by the `ObfuscatedIdGenerator` service. The `ObfuscatedIdGenerator` service creates non-sequential `repositoryId` values. This is important to prevent users from guessing claim codes. The standard `IdGenerator` generates sequential `repositoryId` values. See the *ID Generators* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming Guide*.

The ClaimableTools Component

The `ClaimableTools` component (`/atg/commerce/claimable/ClaimableTools`) provides two pieces of functionality:

- Low-level access to the `Claimable` repository.
- Naming for various claimable item properties.

The first use of the `ClaimableTools` component is simply to create and claim any type of claimable item. When claiming items, it can obtain any super-type of type `claimable` since they all share the same common base type of `claimable`. Additionally, it takes the item-descriptor type as an argument and then creates and adds the claimable item to the repository.

The `ClaimableTools.caseInsensitiveClaimableIds` property determines whether coupon IDs are case-sensitive or not. Having case-sensitive IDs allows for more possible errors when entering coupon codes. Therefore, the `ClaimableTools` component can return multiple items for a given claimable. The component attempts to match the case, but if there is no match, selects the next best match and logs a debug message.

The second use of the `ClaimableTools` component is to provide configurable values for the various properties of claimable items. For example, if someone changes the name of a field in the XML file, you can reflect that change in the code by adjusting the property values of the `ClaimableTools` component.

The ClaimableManager Component

The `ClaimableManager` component (`/atg/commerce/claimable/ClaimableManager`) provides higher level access to the repository and is the mechanism that most applications use to interact with the claimable item functionality. It provides the ability to claim items, initialize them, and then perform functions that are specific to one type of claimable item, thus combining several smaller functions from the `ClaimableTools` package.

An example of how the `ClaimableManager` component can layer together several pieces of functionality from the `ClaimableTools` package is as follows: when a user claims an item by entering a code for a gift certificate, the `ClaimableManager` component attempts to claim the item through the `ClaimableTools` package. Then it attempts to adjust the status of the item to a claimed status, also through the `ClaimableTools` package.

Other examples of functions that the `ClaimableManager` component provides are as follows:

- `createClaimableGiftCertificate`—Creates a gift certificate in the system.
- `debitClaimableGiftCertificate`—Debits a specific amount from the gift certificate.
- `creditClaimableGiftCertificate`—Credits a specific amount to a gift certificate.

The ClaimableManager also handles coupon functionality. For example, if you are using Oracle ATG Web Commerce's multisite feature, the `ClaimableManager` can check the user's site against the list of sites for which the coupon is valid before allowing it to be claimed.

Using Serialized Coupons

Serialized coupons allow merchandisers to create large numbers of coupons with usage limits and randomly generated coupon codes in order to closely target and monitor coupon use.

The merchandiser creates a `CouponBatch` repository item that stores all of the information for the batch of serialized coupons, and uses the batch to generate the desired number of coupon codes. The merchandiser supplies a prefix for the entire batch, and the remainder of the code is randomly generated, as shown in the following examples.

Coupon Code	Prefix	Coupon Number	Validation Characters
HOL1HV5432	HOL	1HV5	432
SHIP6LKC4GC	SHIP	6LKC	4GC

CouponBatch Repository Item

The `CouponBatch` item descriptor holds all of the information related to generating and validating a set of serialized coupon codes, including the promotions to use for the coupon. The `CouponBatch` item descriptor implements the following properties, which are used to generate and validate coupon codes:

Property	Definition	Datatype	Required
`prefix`	String of characters that appears at the start of every coupon code generated by the BatchCoupon.	String	Y
`numberOfCoupons`	Indicates how many coupon codes to generate for a batch.	Int	Y
`seedValue`	A binary array that is used to create a static seed for creating the coupon codes. This value is set when a BatchCoupon is created, and stored to allow the codes to be validated or regenerated.	Binary	N
percentClaimed (derived)	The percentage of coupon batches that have been claimed. This derivation queries for all `BatchPromotionClaimable` items where the `couponBatch` property is equal to the given `couponBatch`. Note that this shows the percentage of unique coupon codes that have been claimed at least once; it is not affected by the `uses` property.	Float	N/A
numberClaimed (derived)	The number of coupons that have been claimed at least once.	Int	N/A

Two related repository items are the `BatchPromotionClaimable` and the `PromotionClaimable` item. A `BatchPromotionClaimable` claimable item type is created when a coupon is claimed from a coupon batch. The item descriptor extends the `PromotionClaimable` item descriptor and adds an additional `couponBatch` property. The `couponBatch` property is a reference to the `CouponBatch` that created the `BatchPromotionClaimable`.

The `uses` and `maxUses` properties of the `PromotionClaimable` item descriptor are used to set usage limits for any `PromotionClaimable` item. The `maxUses` property is configurable; the `uses` property increments when a customer tries to claim a coupon. If `uses` is greater than `maxUses`, the coupon has no more remaining uses and cannot be claimed anymore. If `maxUses` is -1, then the coupon can be claimed an unlimited number of times.

How Batch Coupon Codes Are Generated

In order to generate coupon codes for a coupon batch, the `generateCouponBatchCodes` method in the `CouponBatchTools` class checks to see if the `seedValue` is populated for the batch. The `seedValue` is typically set when the `CouponBatch` item is created; if it has not been set, a new random array of bytes is generated (via `SecureRandomService`) and saved as the batch's `seedValue`. This `seedValue` is used to ensure that each `CouponBatch` has a different set of validation codes; storing it ensures that the codes can be regenerated or validated at a later time.

The `generateCouponBatchCodes()` method uses the `seedValue` as a seed to a random number generator to obtain an array of random bytes, which are then parsed into an encoded base-32 number. The length of the encoded character segment is determined by the `couponBatchTools` component's `couponBatchNumberLength` property. The encoded characters form the coupon number, which is the first portion of the coupon code. This process ensures that the coupon numbers are non-sequential. Duplicates are discarded.

Next, the `generateCouponBatchValidationSeed()` method generates a second seed specific to the coupon number. The new seed value is used to generate an additional array of bytes encoded in base-32, which is used as the validation characters. The number of characters in the resulting code depends on the `CouponBatchTools` component's configured `couponBatchValidationLength` value.

The complete coupon code consists of the prefix + coupon number + verification code, and is stored in the `generateCouponBatchCodes()` method's HashMap.

Note that you can adjust the length of the coupon code (which is supplied by the merchandiser when the coupon batch is created), and that this affects the length of the other sections, the number of coupons that can be generated, and the odds of a user guessing a code. The number of coupons that can be generated for a batch is 32^n, where n is the number of characters used in the coupon number. In practice, the number of coupons generated per batch will be less than the maximum.

A user has a 1/(32^n) chance of brute force guessing the validation characters correctly, where n is the length of the validation code. For example if there are 3 validation characters then the user has a 1/32,786 chance of guessing the proper set of validation characters for a given code if they use a proper prefix and correct coupon code length.

How Serialized Coupons Are Claimed

Just as it does for ordinary coupons, the `ClaimableManager` handles the claiming process for coupon codes from a `BatchCoupon`. If a coupon item is not found for the given code, the `ClaimableManager` checks the `CouponBatch` item descriptors to see if any batches match the coupon prefix. If a match is found, the remaining parts of the coupon code are decoded and validated. If the coupon code is valid, the coupon is active, no claimable item yet exists for the coupon (for coupons that can be used multiple times), and the coupon is valid for the current site, then a `BatchPromotionClaimable` item is created and granted to the user.

Note that if the coupon can be used multiple times, the claimable item is created on the first use; on subsequent uses, the existing coupon is found and no item is created.

The `BatchPromotionClaimable` uses the passed coupon code, including the prefix, as the ID. The `startDate`, `endDate`, `promotions`, and `displayName` properties are copied over from the `CouponBatch` to the new `BatchPromotionClaimable` item. The `BatchPromotionClaimable` item's `uses` property is set to the `CouponBatch` item's uses value minus 1 (the first claim is also counted),, and the `couponBatch` property on the claimable item is set to the `CouponBatch` ID.

The `claimCoupon()` method checks the uses property to determine if the coupon can still be claimed If the uses property equals 0 then an exception is thrown. Otherwise the coupon is claimed and if uses does not equal -1 the uses property is decremented.

CouponBatchTools

The `CouponBatchTools` class handles the lower-level logic in manipulating `CouponBatch` objects, such as generating the different parts of the coupon codes.

You can configure the following properties to change the characters allowed in coupon codes and how many coupons can be generated for a single batch:

Property Name	Type	Default Value	Description
`couponBatchNumberLength`	int	4	Determines the number of characters used as a unique identifier for the coupon number.
`couponBatchValidationLength`	int	3	Determines the number of characters used as validation for the passed in coupon code.
`encodedCharacters`	array	Character set of 0-9, &, *, A-Z excluding A,E, I, L, O and U	The set of characters that will appear in generated coupon codes. Must contain exactly 32 unique values. Default characters are 0-9, and A-Z excluding a, e, i, o, and u.
`maxBatchRatio`	double	0.96	Restricts the number of coupons that can be generated for a batch to a ratio of the total possible unique codes that can be generated for a batch. 0.96 was chosen as a default because it allows for slightly more than 1 million coupons given the default `couponBatchNumberLength`.

The following additional properties refer to other components and configure the random number generator used by the `CouponBatchTools`.

Property Name	Default Value	Purpose
`secureRandomService`	`/atg/dynamo/service/random/SecureRandom`	Component used to generate random numbers
`secureRandomConfiguration`	`/atg/dynamo/service/random/SecureRandomConfiguration`	Contains configuration information for using SecureRandom

Property Name	Default Value	Purpose
`transactionManager`	`/atg/dynamo/transaction/TransactionManager`	Used to begin and end the transactions necessary for obtaining the # of claimed coupons or finding a coupon batch based on prefix.

Tracking and Limiting Coupon Uses

The `ClaimableManager` component's `maxCouponsPerProfile` property sets the maximum number of coupons a customer can use per order. The default is -1, which allows unlimited coupon uses.

As part of the coupon claiming process, the `ClaimableManager.claimCoupon()` method checks to see if the coupon claim limit has been reached, how many coupons are on the customer's profile, and whether those coupons are valid for the current site. If the limit has not been reached, claiming proceeds as normal.

If the limit has been reached, two behaviors are available. The `ClaimableManager.validMaxCouponLimitReachedActions` property can be set to either of the following values:

- `throwException`
- `removeLRU`

It is up to your Web application to handle results from these behaviors.

Tracking Coupon-Adjusted Promotions

The pricing process creates a `pricingAdjustment` object, which stores the promotion that caused the adjustment. The `coupon` property of the `pricingAdjustment` stores the coupon that resulted in the adjustment.

The `ProcUpdateAdjustmentsWithCoupon` pipeline processor iterates through the pricing adjustments on the order and pulls the first `coupon` related to the `promotion` from the `promotionStatus` matching on the promotion. This processor runs before the `ProcRemoveUnusedPromotions` pipeline processor in the `processOrder` pipeline chain.

Removing Unused Coupon-Based Promotions

You can configure Commerce to remove unused coupon-based promotions from the user's profile during processing. This feature tidies up promotions that customers have qualified for, but did not use on an order. The process is as follows:

1. Customer creates an order.
2. Customer enters a coupon code.
3. Promotion is added to the customer's profile.
4. Customer does not qualify for the promotion when the order is placed.

If unused promotions are not removed, those promotions remain on the customer's profile and can be used on a later order (if the customer qualifies) without re-entering the coupon code.

To enable unused coupon removal, set the `removeUnusedCouponBasedPromotions` property of the `RemoveUnusedPromotions` component to `true`. The default is false. If set to true, the `ProcRemoveUnusedPromotions` pipeline processor in the `processOrder` pipeline chain looks at each promotion's status and removes promotions that meet all of the following criteria:

- Created as a result of a coupon claim
- Single-use
- Has not been used
- Start date is before the current date

If the customer wants to enable this feature, the `RemoveUnusedPromotions.enabled` property must be set to `true`. The default is `false`.

Setting Up Gift Certificates

Gift certificates allow a customer to pay for all or part of a purchase using a prepaid amount. Processing a gift certificate involves the following steps:

1. Customer A purchases the gift certificate on behalf of Customer B.
2. Oracle ATG Web Commerce fulfills the purchase for Customer A (and sends a notification e-mail to Customer B as part of the fulfillment process).
3. Customer B uses (claims) the gift certificate to pay for all or part of an order.

The Purchase and Fulfillment Process for Gift Certificates

Set up gift certificates as items in your product catalog, adding separate SKUs for different amounts. For example, you could have two SKUs, one for a $50 dollar certificate and another for a $100 certificate. The value of the gift certificate is stored in the `listPrice` property of each gift certificate SKU in the product catalog.

Each SKU has a `fulfiller` property that defines an `ElectronicFulfiller`. When Customer A adds a gift certificate to his or her shopping cart, the `fulfiller` property tells the `OrderFulfiller` to route the purchase to the specified `ElectronicFulfiller`. The `ElectronicFulfiller` then creates the gift certificate in the `Claimable` repository (by means of the `ClaimableManager` component) and sends an e-mail to the recipient notifying him or her that a gift certificate has been purchased on his or her behalf and including the code to use for claiming it. Refer to the *Configuring the Order Fulfillment Framework* (page 413) chapter for more information on the `ElectronicFulfiller`.

The following properties can be set when the gift certificate is created. The gift certificate is not functional unless the three required properties (identified below) are set.

`amount`	(Required) The original dollar value of the gift certificate.
`amountAuthorized`	(Required) The total amount of money on the gift certificate that has been authorized for debit.

amountRemaining	(Required) The current amount of money remaining on the gift certificate for debit.
purchaserId	The profileId of the person who bought the gift certificate.
purchaseDate	The date the gift certificate was purchased.
lastUsed	The latest date on which fulfillment updated the amount.

Note: By default, the stock level of a gift certificate is set to -1 indicating that an infinite number of the item is available. Setting the stock level of a gift certificate to 0 (out of stock) does not affect the gift certificate's availability because the ElectronicFulfiller does not check a gift certificate's stock level.

Sending an e-mail to the recipient of the gift certificate requires knowing his or her e-mail address. Oracle ATG Web Commerce does this through an ElectronicShippingGroup, which designates the necessary information for delivering electronic goods, in this case an e-mail address. So, when Customer A adds the gift certificate to his or her cart, you add an ElectronicShippingGroup to the order as well, and define the relationship between the two. Use the handleAddSoftGoodToOrder() method in the /atg/commerce/order/ShoppingCartModifier component to do this. The following example shows the JSP code you would add to the page on which the gift certificate product is displayed. It adds the gift certificate to the customer's order and prompts him or her to specify the e-mail address of the recipient:

```
<%@ taglib uri="http://www.atg.com/dsp.tld" prefix="dsp" %>
<dsp:page>

<dsp:form action="<%=request.getServletPath()%>" method="post">
<!-- Store this Product's ID in the Form Handler: -->
 <dsp:input bean="ShoppingCartModifier.ProductId"
paramvalue="Product.repositoryId" type="hidden"/>
<!--set id param so that the Navigator won't get messed up
     in case of an error that makes us return to this page.-->
 <input value='<dsp:valueof param="Product.repositoryId"/>' type="hidden"
name="id">
    Give
    <dsp:input bean="ShoppingCartModifier.quantity" size="4" value="1"
type="text"/>

<!-----------DropDownList format (default) ----------- -->
<!--Create a dropdown list will all Sku in the Product.
           Store the selected SKU's id in the Form Handler: -->
       <dsp:select bean="ShoppingCartModifier.catalogRefIds">
<!--For each of the SKUs in this Product, add the SKU to the
            dropdown list:-->
  <dsp:droplet name="/atg/dynamo/droplet/ForEach">
         <dsp:param param="Product.childSKUs" name="array"/>
         <dsp:param value="Sku" name="elementName"/>
         <dsp:param value="skuIndex" name="indexName"/>
         <dsp:oparam name="output">
<!--This is the ID to store, if this SKU is selected in
             dropdown:-->
          <dsp:getvalueof id="option48" param="Sku.repositoryID"
idtype="java.lang.String">
<dsp:option value="<%=option48%>"/>
</dsp:getvalueof>
<!--Display the SKU's display name in the dropdown
```

```
              list:-->
          <dsp:valueof param="Sku.displayName"/>
         </dsp:oparam>
        </dsp:droplet>
<!--ForEach SKU droplet-->
       </dsp:select> to
       <br>
    Recipient's e-mail address
    <dsp:input bean="ShoppingCartModifier.softGoodRecipientEmailAddress"
size="30"/>
    <p>
 <!-- ADD TO CART BUTTON: Adds this SKU to the Order-->
     <dsp:input bean="ShoppingCartModifier.addSoftGoodToOrder"
value=" Add Gift Certificate to Cart " type="submit"/>
 <!-- Goto this URL if NO errors are found during the ADD TO CART
          button processing:-->
     <dsp:input bean="ShoppingCartModifier.addSoftGoodToOrderSuccessURL"
value="../checkout/cart.jsp" type="hidden"/>
     <dsp:input bean="ShoppingCartModifier.addSoftGoodToOrderErrorURL"
value="product_gift_certificate.jsp" type="hidden"/>
   </td>
  </tr>
 </table>
</dsp:form>

</dsp:page>
```

In this example, note the code that handles the recipient's e-mail address. The customer enters the recipient's e-mail address into a form field set to the property `ShoppingCartModifier.softGoodRecipientEmailAddress`. Thus, the e-mail address is set in the `ElectronicShippingGroup` through the `handleAddSoftGoodToOrder` method, and then the `ElectronicFulfiller` examines it to determine where to send the message.

Note that electronic goods require special processing, so the submit method of the form is set to `addSoftGoodOrder` and not `addItemToOrder`.

For more information on the `ShoppingCartModifier` component, please refer to the Setting Up Gift Lists and Wish Lists (page 71) section.

Using a Gift Certificate

On your Checkout page (or an equivalent page on your sites), include a text field where customers can enter the claim codes for any gift certificates they may have (you send the codes in the gift certificate notification e-mail). Hook the text field up to the `giftCertificateNumbers` property of the `ShoppingCartModifier` component. During the `handleMoveToConfirmation` process, the `ShoppingCartModifier` parses any values it finds in this field into tokens, using white space as the delimiter. This behavior allows customers to enter multiple gift certificate codes into a single text field.

After the system tokenizes the `giftCertificateNumbers` property, it hands each token string to the `ClaimableManager` component. The `ClaimableManager` attempts to match each string to a corresponding gift certificate in the `Claimable` repository. If it cannot find a corresponding item, it adds an error to the `FormHandler`. If the system does find an item that corresponds to the token string, it creates a new `GiftCertificate` payment group and adds it to the order.

You must initialize the properties of the payment group to the correct values. The properties to initialize include the claim code ID, the user profile ID, and the amount of the gift certificate. Finally, create a relationship between the order and the new payment group indicating that the gift certificate payment group can account for any amount up to the value of the gift certificate.

Please refer to the *Configuring Purchase Process Services* (page 263) chapter for more information on payment groups.

Settling a Gift Certificate

Gift certificate settlement is handled similar to the way that settlement for credit cards is handled. (For more information, see the Processing Payment of Orders (page 300) section.)

When an order is submitted, each `PaymentGroup` in the order is authorized using the `PaymentManager`. The `PaymentManager` has different processors for gift certificates and credit cards. The processor used for gift certificates handles the authorization, debit, and credit of gift certificates. When a gift certificate is authorized, the `amountRemaining` is reduced by the amount being authorized. This prevents the same gift certificate from being used for more than it is worth.

During fulfillment, the `PaymentGroup` containing the gift certificate is debited through the `PaymentManager` (and ultimately the `GiftCertificateProcessor`). When this happens, the amount that was authorized is checked against the amount that is being debited. If the amount authorized was the same as the amount being debited, the `PaymentManager.debit` returns successfully. If there are any differences, they are corrected now. To make sure that the gift certificate is valid if an order is removed before the debit occurs (and after authorization), any amount that was authorized will be credited back to the gift certificate before the `PaymentGroup` is removed.

Oracle ATG Web Commerce treats coupons as a type of promotion that customers can claim. The customer types in a specific code, and the system adds the corresponding promotion or promotions to the customer's profile. Most of the code for handling coupons is included in the `ClaimableManager` component and the standard promotion architecture, with a `FormHandler` to connect these two systems. The process for handling a coupon is as follows:

1. Obtain a coupon code.
2. Try to claim the coupon from the `ClaimableRepository`.

   Coupon codes are case-sensitive. For example, `COUP100` and `coup100` are two different coupon codes.

   Important: If you are using an MSSQL database, be aware that unlike some other databases, MSSQL uses a case-insensitive character set by default. If you want to have case-sensitive coupon codes, set the case-sensitive character set at the database level.
3. Add the resulting promotion or promotions to the `activePromotions` list in the user profile.

Use the `/atg/commerce/promotion/CouponFormHandler` component to obtain a coupon code and add it to a customer's list of promotions in the customer's `activePromotions` profile property. Create an input field on an appropriate site page (for example, a Checkout page) and hook it up to the `couponClaimCode` property of the `CouponFormHandler` component. Then, have the form submit to the `handleClaimCoupon` method of the form handler. This method uses the `ClaimableManager` component to attempt to get a coupon from the Claimable repository. Then it extracts the promotion from the coupon and uses the `PromotionTools` component to place the promotion into the customer's user profile.

The following example shows the JSP code for using the `CouponFormHandler` component:

```
<%@ taglib uri="http://www.atg.com/dsp.tld" prefix="dsp" %>
<dsp:page>

<dsp:form method="post">
<!-- Where to go to on success or failure -->
<dsp:input bean="CouponFormHandler.claimCouponSuccessURL"
value="CouponClaim.jsp" type="hidden"/>
```

```
<dsp:input bean="CouponFormHandler.claimCouponSuccessURL"
value="CouponClaim.jsp" type="hidden"/>

<!-- Get the coupon claim code -->
Coupon code: <dsp:input bean="CouponFormHandler.couponClaimCode"
type="text"/>
<dsp:input bean="CouponFormHandler.claimCoupon" type="submit"/>
</dsp:form>

</dsp:page>
```

When a user enters a coupon code on a Checkout page, ATG Commerce checks that:

- The coupon is active
- The coupon has not expired
- The coupon is valid on the current site
- Any promotions associated with the coupon are not expired or otherwise invalid

If these conditions are met, the user "claims" the coupon, which means that the promotions are added to the user's `activePromotions` profile property. Note that promotions cannot be granted independently while claiming a coupon; either all are granted or none.

During order pricing, Oracle ATG Web Commerce determines whether the order qualifies for the coupon's promotions by checking that:

- The order meets the requirements of the promotions
- The promotions have not expired

This double-checking ensures that if a user claims a coupon as part of one order, discontinues that order, then creates a second one, any promotions apply to the second order, as long as the promotions are active and applicable.

Promotions given by a coupon persist on the user profile, not as part of the order. A claimed coupon is automatically applied to any order to which it qualifies; there's no need to claim a coupon twice. You can persist a coupon code with an order by adding a new property to the `Order` object and storing the coupon code in the new property. For information on how to extend the commerce object hierarchy to include a new property, refer to Extending the Purchase Process (page 341) in the *Customizing Purchase Process Externals* chapter.

The other step to consider when you set up coupons is to make sure that there are coupons in the Claimable repository for a customer to claim. The following example shows the default item-descriptor for coupons:

```
<!-- Promotion Claimable object -->
<item-descriptor name="PromotionClaimable" super-type="claimable"
                 sub-type-value="PromotionClaimable">
 <table name="dcspp_coupon" type="auxiliary" id-column-name="coupon_id">
    <property name="promotion" column-name="promotion_id"
item-type="promotion" repository="/atg/commerce/pricing/Promotions"/>
 </table>
</item-descriptor>
```

The coupon item-descriptor is defined in `/atg/commerce/claimable/claimableRepository.xml`, which is located in a `.jar` file at `<ATG10dir>/DCS/config/config.jar`. The item-descriptor defines a claim code

ID and a link to a promotion object in the Promotions repository. If your Web sites require multiple types of coupons, you can define additional item-descriptor types by editing the `claimableRepository.xml` file and then specifying the valid coupon types in the `validCouponItemTypes` property in the `CouponFormHandler` properties file. When a coupon is claimed, the `CouponFormHandler.checkPromotionType` method checks the item type of the coupon corresponding to the given claim code against the array of acceptable item types in the `CouponFormHandler.validCouponItemTypes` property.

You can populate the Claimable repository through the following methods:

- Oracle ATG Web Commerce Merchandising (see the *ATG Merchandising Guide for Business Users*)
- ATG Control Center (see the *ATG Commerce Guide to Setting Up a Store*)
- the `atg.commerce.promotion.CouponDroplet` (see *Appendix: ATG Commerce Servlet Beans* in the *ATG Commerce Guide to Setting Up a Store*)

8 Commerce Pricing Services Overview

The next several chapters discuss the Oracle ATG Web Commerce pricing services. This chapter provides an introduction to concepts and terms, while those that follow provide details on the default services and the ways you can extend them.

Pricing services in Commerce provide a flexible system for personalizing the prices of items in your product catalog. Just as you can personalize content for every shopper who visits your sites, you can tailor prices and generate them dynamically on demand. You can also set up coupons, sales, and other promotions, target them for appropriate visitors, associate them with sites in a multisite environment, and use them in dynamic pricing situations.

Commerce pricing services include a set of standard features that are designed to handle the pricing demands of most Web sites. If your sites require additional functionality, you can write a new implementation of the many public pricing APIs that the product supplies.

This chapter provides an overview of the provided classes and components that make up the pricing system. It includes the following sections:

Common Terms in Pricing Services

The following table describes common terms used in pricing. Each of these terms is described in detail in the chapters that follow.

Pricing Term	Definition
Calculator	An object that looks at all or part of an `Order` and returns a price. See the *Commerce Pricing Calculators* (page 139) chapter for information.
Pricing Model Definition Language (PMDL)	Describes promotions. This includes the discount rules for *when* a promotion may apply (the condition), the rules for *what* may be discounted (the offer), and how to apply the discount (for example, 10% off). The ACC includes an interface for creating rules, as does Oracle ATG Web Commerce Merchandising; see the *ATG Commerce Guide to Setting Up a Store* and the *ATG Merchandising Guide for Business Users*
`PriceInfo`	An object that contains the price of part of an order. There are four main kinds of `priceInfo` objects: `OrderPriceInfo`, `ItemPriceInfo`, `ShippingPriceInfo`, and `TaxPriceInfo`. There is also a `DetailedItemPriceInfo`, which is described with the `ItemPriceInfo` object. See the Price Holding Classes section of the Base Commerce Pricing Engines chapter.
`PricingAdjustment`	Describes why and how a particular price was changed. It includes a description of the change and the amount. It also contains the promotion (if any) that triggered the change. See the DetailedItemPriceInfo (page 132) object for information.
`PricingContext`	Provides the items being priced, the order, the site, the promotion, the locale, and the profile used when a price was calculated. It can also include secondary information not applicable in all cases, such as the shipping group.
`PricingModel`	A repository item that describes a discount. It includes a PMDL rule and the discount type and amount. It also contains information about when the pricing model may be used, including upsell information if applicable.
Promotion	Allows you to offer periodic discounts on specific products or groups of products. See PMDL and `PricingModel` in this table, and the *Understanding Promotions* (page 169) chapter.
Qualifier	A service that interprets a PMDL rule and decides what, if anything, may be discounted.
Condition	The first part of a PMDL rule, which defines when something can receive a discount.
Offer	The second part of a PMDL rule, which defines which part of an order receives a discount.

Using Dynamic vs Static Product Pricing

Oracle ATG Web Commerce supports two pricing methods for use on product pages:

- Static pricing—Each item in the catalog has a list price stored in the `listPrice` property of the catalog repository (optionally, each item can also have a `salePrice`). You display the price on the appropriate site pages, and the Commerce pricing services use that price as a base for calculating order totals, shipping costs, and sales tax.

- Dynamic pricing—Programmatically determines the price. Dynamic pricing is always used on the shopping cart and purchase process pages, but can also be used on product pages if necessary.

These are both forms of SKU-based pricing, as opposed to pricing based on price lists. See the *Using Price Lists* (page 211) chapter for information on price lists.

Using dynamic pricing on a product page can cause a significant decrease in performance compared to using static pricing. Many sites do not need to show dynamic pricing on product pages; it may be sufficient to show dynamic prices only when a customer places items in the shopping cart. For example, suppose you send a specific group of customers a coupon for 20% off all blue items. On the product pages of the site, blue items appear with their static list or sale price, which is the same for all customers. However, when a customer with the "20% off" coupon adds a blue item to his or her shopping cart, dynamic pricing takes effect, and the item price appears on the Shopping Cart or Checkout page (for example) with a discount of 20%.

If you do decide to use dynamic pricing on product pages, you can still ensure a high level of performance by using the pricing features selectively. You can use dynamic pricing on certain product templates and static pricing on others. You can also choose to restrict dynamic pricing to certain types of customers (for example, registered visitors only).

How Static Pricing Works

With static pricing, each item in the catalog has a list price stored in the `listPrice` property of the catalog repository. You display the price on the appropriate site pages, and the Commerce pricing services can then use that price as a base for calculating order totals, shipping costs, and sales tax.

Optionally, you can maintain more than one price per item. For example, you can give each item a fixed sale price in addition to its list price by specifying a value for the `salePrice` property in the catalog repository. When you want the alternate price to take effect, use the `Switch` servlet bean with the `onSale` property from the Catalog repository. The following example uses the default Commerce product catalog:

```
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param param="sku.onSale" name="value"/>
  <dsp:oparam name="false">
    List price of <dsp:valueof param="sku.listPrice" converter="currency"/>
  </dsp:oparam>
  <dsp:oparam name="true">
    List price of <dsp:valueof param="sku.listPrice" converter="currency"/>
    on sale for <dsp:valueof param="sku.salePrice" converter="currency"/>!
  </dsp:oparam>
</dsp:droplet>
```

For information about the `Switch` servlet bean, see *Appendix B: ATG Servlet Beans* in the *ATG Page Developer's Guide*.

How Dynamic Pricing Works

You can use dynamic pricing for the following types of pricing object:

- **Items.** Each item has a list price that can be specified in the `listPrice` property of the Product Catalog repository. Oracle ATG Web Commerce pricing services adapt the list price by applying any discounts or other promotions that you have set up. For example, you might set up a "2-for-1" sale for a limited period on a specific type of item. (Note that an "item" is a `CommerceItem`, which represents a quantity of a SKU or a product).
- **Orders.** Commerce pricing services calculate the total cost of an order and can apply any discounts that are applicable (for example, a customer might have a coupon offering a 10% discount on a total order).
- **Shipping price.** Commerce pricing services can calculate the price of shipping for an order and apply discounts if applicable.
- **Tax.** Commerce pricing services can calculate the sales tax for an order.

Commerce uses the same basic structure for pricing each type of object. The structure includes the following:

- A pricing engine
- One or more calculators
- A helper method in the qualifier service
- An item-descriptor in the Promotions repository

For example, the structure for pricing and discounting catalog items includes the following:

- An Item Pricing Engine
- An Item List Price Calculator, an Item Sale Price Calculator, and an Item Discount Calculator
- The `findQualifyingItems` call in `atg.commerce.pricing.Qualifier`
- The `Item Discount` item-descriptor in the repository `/atg/commerce/pricing/Promotions`.

Note: The structure for determining sales tax is slightly different because Oracle ATG Web Commerce does not support offering discounts on tax. For this reason, there is no item-descriptor for tax discounts in the default Promotions repository. You could add such an item-descriptor if necessary.

The *ATG Business Commerce Reference Application Guide* contains extensive descriptions of how dynamic pricing was implemented in the Motorprise reference application. Also see the sections on the *PriceItem* and the *PriceEachItem* servlet beans in the *ATG Commerce Guide to Setting Up a Store*.

How Pricing Services Generate Prices

Oracle ATG Web Commerce pricing services are based on two complementary elements: pricing engines and pricing calculators. These components work together to determine prices for orders, sales tax, and shipping.

Pricing engines are responsible for these tasks:

- Retrieving any promotions that are available to site visitors
- Determining which pricing calculators generate prices
- Invoking the calculators in the correct order

Pricing calculators are responsible for the following:

- Looking up the price.
- Using information they receive from the pricing engines, promotions, and the qualifier service to determine prices.

The following overview describes the way Commerce calculates prices.

Note: If your promotions were generated using Oracle ATG Web Commerce 9.1 or earlier versions, see the documentation for that version; the pricing process treats promotions differently depending on the version of Commerce used to create them.

Before pricing happens, the following steps take place:

1. On a scheduled basis, the pricing engines load global promotions (those defined as applying automatically to all customers). An engine builds its list by using its `globalPromotionsQuery` property to query the Promotions repository, searching for any promotion where the Automatically Give to All Customers (`global`) property is set to `true`. Each engine loads the global promotions that are specific to that pricing engine; for example, the `ItemPricingEngine` loads the global Item Discount promotions.

2. At the start of the customer session, a `PricingModelHolder` instance is created. `PricingModelHolder` calls each pricing engine's `getPricingModels()` method.

3. The pricing engine `getPricingModels()` method gets any promotions listed in the `activePromotions` property of the current customer's profile and merges them with the global promotions list it previously created. The pricing engine can also veto promotions from being returned to the `PricingModelHolder` at this point.

4. `PricingModelHolder` periodically updates both the global and active promotions.

The result is that `PricingModelHolder` has a merged list of global and active promotions for each pricing engine. When the customer performs an action that prompts a pricing operation, such as adding an item to their cart, the following steps are performed:

1. The business layer logic (such as a `PriceItem` servlet bean in a page) invokes a pricing engine (see The Base Pricing Engine (page 127)).

2. The pricing operation invokes `PricingTools` which then gets the `PricingModelHolder` for that customer.

3. `PricingTools` gets the promotions from the `PricingModelHolder` and calls the pricing engine's `priceItems` method, passing in the promotions as a list.

4. The pricing engine applies its configured precalculators in the order in which they appear in its `preCalculators` property. A precalculator modifies a price without using any associated promotions. For example, a site could use a list price precalculator or an order subtotal precalculator to establish a base price to which subsequent calculators then apply a discount.

   Each type of engine calls its corresponding type of precalculator – for example, the `OrderPricingEngineImpl` calls `OrderPricingCalculators` and the `TaxPricingEngineImpl` calls `TaxPricingCalculators`.

5. The pricing engine then takes the promotions list that was passed in and can again veto promotions from that list. The remaining promotions are sorted by priority and then evaluated.

   Each promotion contains a PMDL rule that describes the discount. For example, a rule might define a discount in which a customer buys one item and receives a second for free. The PMDL specifies the

calculator type to use to determine the amount of the discount in the `calculator-type` attribute of the `discount-structure` element. The calculator type maps to a calculator component in the `calculatorTypeCalculators` map.

For each available promotion, Commerce does the following:

- The pricing engine calls the appropriate helper method (`findQualifyingItems`, `findQualifyingOrder`, or `findQualifyingShipping`) in the `Qualifier` class to determine which items should be discounted. The pricing engine passes the current pricing context into the helper method's input parameters.
- The `findQualifyingItems()` method calls `evaluateTarget()`, which returns a Collection of `QualfiedItem` objects, representing `CommerceItem` objects. The `findQualifyingOrder()` and `findQualifyingShipping()` methods return a single `MatchingObject`.
- The `QualifiedItem` and `MatchingObject` include discount information such as the PMDL discount structure. See the QualifiedItem Class (page 164) for more information.
- The `QualifiedItem` or `MatchingObject` information is returned to the pricing engine, which uses the discount information to determine which calculator to use.
- The pricing engine calls the calculator and passes in the items to be discounted and the discount information.
- The calculator marks items that have received a discount, which might not be eligible for further promotions.
- The qualifier marks items that have already been used as qualifiers for the promotion. This prevents the qualifier items from being used again during the same price calculation.
- The calculator applies the discount to the list of objects.

6. The pricing engine applies its configured `PostCalculators`, which make any necessary modifications to the price after discounts have been applied. Each pricing engine calls postcalculators of its own type.
7. The pricing engine returns an updated `PriceInfo` object.

This process is repeated every time a price is requested. Price requests are resource-intensive, and should be performed only when necessary.

PricingTools Class

The `atg.commerce.pricing.PricingTools` class is the main way in which business-layer logic interacts with the pricing engines and the other classes in the `atg.commerce.pricing` package. In Oracle ATG Web Commerce, the classes that extend the `ItemPricingDroplet` class (`PriceItemDroplet` and `PriceEachItemDroplet`) use `PricingTools` to interface with all the Commerce pricing engines.

When a store using Commerce needs a price for items, orders, shipping, or tax, `PricingTools` can be consulted to return the price. In addition, `PricingTools` contains translation functions that identify which `currencyCode` goes with which locale.

`PricingTools` includes methods that can be called to produce prices. These methods consult the configured pricing engines, which are described in detail in the *Commerce Pricing Engines* (page 127) chapter.

PricingTools also determines the pricing locale. If you are using SKU-based pricing (instead of price lists; see *Using Price Lists* (page 211) for information), PricingTools returns the configured defaultLocale property. The pricing engines use this to determine the currency code. If you are using Oracle ATG Web Commerce's multisite feature and want to use different currency codes for different sites, you can take the following steps:

1. Add a pricing locale to the Site repository items.
2. Override the PricingTools.getDefaultLocale method to retrieve the currency code for the site.

See the *ATG Platform API Reference* for detailed information on the properties and methods of this class.

PricingModelHolder

The atg.commerce.pricing.PricingModelHolder class is a session-scoped component that holds the merged list of a customer's active and global promotions while he or she is using the Web application. The pricing engine APIs define a method for collecting a customer's pricing models. Because it can be resource intensive to perform this operation, the PricingModelHolder is essentially a session cache of promotions.

If a new promotion becomes available during a user's session, the user's promotions must be reloaded for the user to see this new discount. The reinitializeTime property in the pricingModelHolder is set to reload a user's promotions every 10 minutes by default. You can change this time if desired. Setting the reinitializeTime property to a smaller value (two minutes) will affect performance, but will minimize the risk of a user missing a promotion that is added during his or her session. You should consider changing the reinitializeTime value if new promotions are added frequently. You should also decrease this value if promotions are being delivered by scenarios with short delays in them, for example, giving a promotion two minutes after a user logs in.

Before adding a promotion to the list, the PricingModelHolder runs a series of checks to eliminate as many promotions as possible from consideration (for example, a "buy one rain hat, get a free umbrella" promotion where the customer's cart does not contain a rain hat). This saves considerable processing effort, as fewer promotions have to be evaluated at pricing time.

The PricingTools class uses PricingModelHolder to perform order pricing. Each pricing engine takes only the collection of pricing models related to its own type as a parameter. For example, order pricing engines take only the pricing models related to order pricing. The PricingTools method, which accepts a PricingModelHolder, extracts individual collections and passes the collections into the appropriate pricing engines.

Developers should not create an instance of this class and call into the PricingTools class. Instantiate an instance of this class only as a session-scoped component that can be resolved through the request. This is the pattern that the item pricing servlet beans use. If no pricing models are supplied as explicit parameters, the PricingModelHolder is resolved from the request, and the collection is retrieved.

PricingAdjustment

The atg.commerce.pricing.PricingAdjustment class represents an element of a price's audit trail. A chain of these objects represents all changes made to the price. These objects appear in the adjustments list of AmountInfo. A PricingAdjustment is created by a pricing calculator when it modifies an AmountInfo object.

The `PricingAdjustment` class contains the following properties:

- `adjustmentDescription`: A short description of the adjustment that this object records.
- `pricingModel`: The ID of the pricing model, if any, that adjusted the price.
- `manualPricingAdjustment`: ID of the manual adjustment that was applied to the order, if any. Manual adjustments are applied by agents using CSC; see the *ATG Commerce Service Center User Guide*.
- `totalAdjustment`: The total adjustment amount is calculated by multiplying the `adjustment` property by the `quantityAdjusted`.
- `quantityAdjusted`: The quantity of the object whose price was adjusted.
- `adjustment`: The price adjustment per quantity of one object. This value is calculated by dividing the value of the `totalAdjustment` property by the value of the `quantityAdjusted` property.

PricingCommerceItem

The `atg.commerce.pricing.PricingCommerceItem` is a simple `CommerceItem` used as a placeholder while pricing items. This `CommerceItem` cannot be added to an order.

The pricing engines can compute prices only within the context of a `CommerceItem`. There may be times, however, when you want to price an entity for a customer and no `CommerceItem` is available. This problem is most evident when prices are shown for products in the catalog. Products and SKUs are usually represented by `RepositoryItems`, which the pricing engines do not handle. `PricingCommerceItem` is a `CommerceItem` class into which you can plug the product and SKU objects. The item pricing servlet beans deal with input, which are plain `RepositoryItems`, and "convert" them to `CommerceItems`.

These `CommerceItems` cannot be used in the default order management system. Use the `OrderManager` APIs to add a `CommerceItem` to an order.

PricingModelProperties

The `atg.commerce.pricing.PricingModelProperties` class contains the names of properties of the `ItemDescriptor` in the Promotions repository that represents pricing models.

The `PricingModelProperties` class stores these names so that they may be internationalized or otherwise changed to suit a particular installation. For more information, refer to the *ATG Platform API Reference*. If the name of a property descriptor is changed in the `ItemDescriptor` that defines the pricing models in the Promotions repository, you must change the corresponding value here as well. For example, the `PricingModelRepository` holds all pricing models. It contains an item descriptor called `pricingModel`. The properties of this item descriptor need to appear in the `PricingModelProperties` class.

9 Commerce Pricing Engines

Pricing engines are responsible for the following tasks:

- Retrieving any promotions that are available to the site visitor.
- Determining which pricing calculators generate the price.
- Invoking the calculators in the correct order.

This chapter describes the base Commerce pricing engine classes and interfaces. It includes the following sections:

For more information on the default pricing interfaces and classes, see the *ATG Platform API Reference*.

Pricing Engine Interfaces

This section describes the Oracle ATG Web Commerce engine interfaces:

The Base Pricing Engine

`atg.commerce.pricing.PricingEngine` is the main interface for interacting with the `atg.commerce.pricing` package. Extensions of this interface describe objects that calculate prices

for different types of Commerce object. The `PricingEngine` interface itself contains only one method, `getPricingModels`, which extracts a collection of promotions from an input profile. Different `PricingEngine` implementations take a range of information to calculate prices for their specific classes of object.

Oracle ATG Web Commerce provides four extensions of the main `PricingEngine` interface. Each extension provides a price for a different object type:

- `atg.commerce.pricing.ItemPricingEngine`
  Provides a price for `atg.commerce.order.CommerceItem` objects.
- `atg.commerce.pricing.OrderPricingEngine`
  Provides a price for `atg.commerce.order.Order` objects.
- `atg.commerce.pricing.ShippingPricingEngine`
  Provides a price for `atg.commerce.order.ShippingGroup` objects.
- `atg.commerce.pricing.TaxPricingEngine`
  Determines tax for `atg.commerce.order.Order` objects.

The pricing context is defined by the method's input parameters, which typically include the following:

- The objects to be priced.
- The promotions to apply to the prices.
- The customer's profile.
- The locale for which the price is being generated.
- A hash table of extra parameters, which exists so that new APIs do not have to be created to accommodate additional pricing context information. One important use for extra parameters is to pass in the site ID, if you want to override the current site context.

These engine interfaces are described in the sections that follow; the calculators are described in the *Commerce Pricing Calculators* (page 139) chapter.

ItemPricingEngine Interface

`atg.commerce.pricing.ItemPricingEngine` is an extension of the `PricingEngine` interface. It describes an object that determines prices for `atg.commerce.order.CommerceItem` objects. The prices that are generated are `ItemPriceInfo` objects.

An `ItemPricingEngine` can determine prices in three ways:

- The `priceItem` method prices an item as a single item. It reviews any promotions that are passed in through a `pPricingModels` parameter.

  The single item that is passed in is the only one available to satisfy the requirements of any promotion. For example, if the item passed in is "5 blue women's shorts," and there is a promotion for "Buy 7 or more blue shorts, get one pair free," the promotion would not take effect. This call is mainly used for displaying item prices when a customer is browsing the catalog.
- The `priceEachItem` method batch processes all the items that are passed in, but they are priced as if they were passed into `PriceItem` one at a time.
- The `priceItems` method prices all input items all in the same context. To continue with the same example, the customer now puts "6 blue men's shorts" in the shopping cart in addition to the "5 blue women's shorts."

While the customer is just browsing the catalog, the "Buy 7 or more blue shorts, get one pair free" promotion is not factored in when displaying prices. Therefore, when the customer makes the decision to add the shorts to the shopping cart, the price shown is still full price for all the shorts. However, when the customer subsequently displays the contents of his or her cart, the promotion takes effect and shows that one pair of shorts is free.

The pricing context is defined by the method's input parameters. In the case of `priceItem`, the context is as follows: the items to be priced, the promotions to factor, the profile of the user, the locale, and any additional parameters.

The context can be important because some promotions take effect only if the item appears in a pricing context with other items. For example, a certain promotion might give 10 percent off an order if the pricing context includes one shirt and one pair of pants. The items would only receive the discount if priced in the same context.

Oracle ATG Web Commerce provides the `ItemPricingEngineImpl` class as a default implementation of the `ItemPricingEngine` interface. It computes the price of items both individually and in a larger context. It invokes a series of `ItemPricingCalculators`.

OrderPricingEngine Interface

`atg.commerce.pricing.OrderPricingEngine` is an extension of the `PricingEngine` interface. (See The Base Pricing Engine (page 127).) It describes an object that determines prices for `Order` objects, which equal the total price of all items in a customer's shopping cart. An `OrderPricingEngine` uses the `priceOrder` method to determine the price of an order.

The pricing context is defined by the `priceOrder` method's input parameters. Implementations of this interface create an `OrderPriceInfo` object that accurately represents the price of an input `order`. The way in which they do this depends on the implementation. The specific way in which the engine creates the order object varies according to individual implementations.

Oracle ATG Web Commerce provides the `OrderPricingEngineImpl` class as a default implementation of the `OrderPricingEngine` interface. It computes the price for an order, invoking a series of `OrderPricingCalculators`.

ShippingPricingEngine Interface

`atg.commerce.pricing.ShippingPricingEngine` is an extension of the `PricingEngine` interface. It describes an object that determines prices for `ShippingGroup` objects.

Implementations of this interface determine the cost of shipping the contents of a shipping group. The `priceShippingGroup` method asks this object to determine a price for the specified shipping group. The `getAvailableMethods` call returns the methods available for shipping the specified group.

Oracle ATG Web Commerce provides the `ShippingPricingEngineImpl` class as a default implementation of the `ShippingPricingEngine` interface. It computes the shipping cost for an order by invoking a series of `ShippingPricingCalculators`.

TaxPricingEngine Interface

`atg.commerce.pricing.TaxPricingEngine` is an extension of the `PricingEngine` interface. It describes an object that determines taxes for `Order` objects.

Implementations of this interface determine the price for the tax associated with a specified order object. The interface provides one way to ask for a tax price. The calling code provides the pricing context by inputting the order, any pricing models (promotions), a locale in which the pricing should occur, the current profile, and any additional parameters.

Oracle ATG Web Commerce provides the `TaxPricingEngineImpl` class as a default implementation of the `TaxPricingEngine` interface. It computes the tax for an order, invoking a series of `TaxPricingCalculators`.

PricingConstants Interface

The `atg.commerce.pricing.PricingConstants` interface contains constant values for static reference by other classes in the `atg.commerce.pricing` package.

Default Pricing Engines

Oracle ATG Web Commerce includes four preconfigured implementations of its pricing engine classes. You can use these implementations as they are or adapt them for your own site development needs.

- Default Item Pricing Engine (page 131)
- Default Order Pricing Engine (page 131)
- Default Tax Pricing Engine (page 131)
- Default Shipping Pricing Engine (page 131)

PricingEngineService

The `PricingEngineService` class is a `GenericService` implementation of a `PricingEngine`. `PricingEngine` implementations can extend this class to leverage scheduling, global promotions, locale, and other configuration functionality.

The `PricingEngineService` includes the `getCalculatorForCalculatorType()` method, which determines the calculator component to use for promotions (note that it is not used for pre- or post-calculators). To make this determination, it consults a Map in which the key is a `CalculatorType`. The calculator type specified in the PMDL is used to identify the correct calculator component. The calculator identified by `getCalculatorForCalculatorType()` is then used by the pricing calculator (such as `ItemPriceCalculator`). You can use the map to extend the pricing engine and add your own calculators to the pricing system.

Note: Only use the calculator type Map for promotions created in Commerce 10 or later. Earlier versions use a different mechanism for retrieving calculator details.

The `QualifiedItems` and `MatchingObjects` are passed to the calculator using the "extra parameters" Map. For the `ItemPricingCalculator`, the List of `QualifiedItem` objects is stored in `Constants.QUALIFIED_ITEMS`. For `Order`, `Shipping`, and `TaxPricingCalculators`, the `MatchingObject` is stored in `Constants.MATCHING_OBJECT`. For both, the `DiscountStructure` is stored in `Constants.DISCOUNT_STRUCTURE`.

Default Item Pricing Engine

The `ItemPricingEngine` component is a preconfigured implementation of the `ItemPricingEngineImpl` class. It determines the price of one or more items by retrieving applicable promotions from the customer's profile and invoking one or more `ItemPricingCalculators`.

You can view and modify this component in the ATG Control Center. Its location is `/atg/commerce/pricing/ItemPricingEngine`.

Default Order Pricing Engine

The `OrderPricingEngine` component is a preconfigured implementation of the `OrderPricingEngineImpl` class. It determines the price of an entire order by invoking a series of `OrderPricingCalculators`. It uses the same mechanisms as the `ItemPricingEngine` component for determining which promotions to apply.

You can view and modify this component in the ATG Control Center. Its location is `/atg/commerce/pricing/OrderPricingEngine`.

Default Tax Pricing Engine

The `TaxPricingEngine` component is a preconfigured implementation of the `TaxPricingEngineImpl` class. It determines the price of tax for an order by invoking a series of `TaxPricingCalculators`. It uses the same mechanisms as the `ItemPricingEngine` component for determining which promotions to apply to taxes.

The `TaxPricingCalculator` determines if an item is taxable using `pricingTools.isTaxable()` method. If an item is taxable, `pricingTools.isTaxable()` returns true. By default, all items are taxable. `pricingTools.calculateTaxableAmount()` then determines the tax by returning 0 if `isTaxable` returns `false`, otherwise it returns the price of the item minus its `orderDiscountShare`.

You can view and modify this component in the ATG Control Center. Its location is `/atg/commerce/pricing/TaxPricingEngine`.

Default Shipping Pricing Engine

The `/atg/commerce/pricing/ShippingPricingEngine` component is a preconfigured implementation of the `ShippingPricingEngineImpl` class. It determines the price of shipping for an order by invoking a series of `ShippingPricingCalculators`. It uses the same mechanisms as the `ItemPricingEngine` component for determining which promotions to apply.

Price Holding Classes

This section describes the Oracle ATG Web Commerce price holding classes. These classes are used to store price information about different object types.

- DetailedItemPriceInfo (page 132)
- OrderPriceInfo (page 135)
- ShippingPriceInfo (page 135)
- TaxPriceInfo (page 135)

AmountInfo

The `atg.commerce.pricing.AmountInfo` parent class represents price information about an object. In addition to the actual amount, it also contains information about how to interpret that amount. This class is the base for the ItemPriceInfo (page 132), OrderPriceInfo (page 135), TaxPriceInfo (page 135), and ShippingPriceInfo (page 135) classes.

For information about `AmountInfo` properties, see the *ATG Platform API Reference*.

ItemPriceInfo

The `atg.commerce.pricing.ItemPriceInfo` class contains information about the price of an item (a `CommerceItem`). It also contains detailed price information about how individual quantities of the `CommerceItem` were priced. For example, if an item's quantity is 5, and 3 items received Discount A and the other two received Discount B, there would be two `DetailedItemPriceInfo` entries in `currentPriceDetails`:

- One `DetailedItemPriceInfo` entry for the quantity of 3, describing the changes discount A made to the price.
- One `DetailedItemPriceInfo` entry for the quantity of 2, describing the changes discount B made to the price.

The `amount` property of `ItemPriceInfo` (inherited from `AmountInfo`) should always equal the sum of the amounts of the `DetailedItemPriceInfo` entries in `currentPriceDetails`.

For information about `ItemPriceInfo` properties, see the *ATG Platform API Reference*.

DetailedItemPriceInfo

This section describes `DetailedItemPriceInfo` objects, including how `DetailedItemPriceInfo` objects relate to each other and to their `ItemPriceInfo`.

It is easiest to understand `DetailedItemPriceInfo` by comparing it to `ItemPriceInfo`. The `ItemPriceInfo` object describes the price for an entire `CommerceItem` in an order. The `amount` property of the `ItemPriceInfo` is the final price for the given `CommerceItem`. If part of the quantity of the `CommerceItem` was discounted due to a promotion, this information is reflected in the `ItemPriceInfo`. For example, if the order contains 10 shirts at $10.00 each, and there was a promotion "Buy 9 shirts, get 1 free" then the `amount` property of the `ItemPriceInfo` would be $90.00. The `PricingAdjustments` in the `ItemPriceInfo` would contain one `PricingAdjustment` for the list price, and one for the promotion. For more information on the `ItemPriceInfo` object, see the ItemPriceInfo (page 132) section.

The `DetailedItemPriceInfo` objects provide a much more detailed view of the price of an item. In the example above, there would be two `DetailedItemPriceInfo` objects:

- `DetailedItemPriceInfo` object #1:
  - `quantity`: 9
  - `amount`: $90.00
  - `PricingAdjustment`: set to the list price.
- `DetailedItemPriceInfo` object #2:
  - `quantity`: 1
  - `amount`: $0.00
  - `PricingAdjustment`: There would be two `PricingAdjustments`. One with the list price, and one with the promotion that caused the item to be free.

Another feature of `DetailedItemPriceInfo` is the `range` property. Instead of a detail referring to "5 items" it is more specific and refers to "the first 5 items" or "items 2-6". This is used for two reasons:

- To split the details of items in different shipping groups. There is a `range` property in `ShippingGroupCommerceItemRelationship`. `DetailedItemPriceInfo` objects cannot refer to items in more than one shipping group.
- During qualification of a promotion (the process of determining if a particular promotion applies to the order) we need to know which items have been looked at and which items have already qualified a promotion. For more information, see the *Qualifier Class* (page 159) section.

The following statements are true for all `CommerceItem` objects:

- Each `CommerceItem` has exactly one `ItemPriceInfo`.
- The price of a particular item (if there are 10 shirts, the first shirt, or the second shirt, etc.) is described by exactly one detail.
- Each `DetailedItemPriceInfo` in the `ItemPriceInfo` describes the price of a quantity of items that are all priced in exactly the same way. (The same list price, the same sale price, and the same promotions.)
- All the items described by a `DetailedItemPriceInfo` are in the same shipping group.

To make sure all of the above statements are true, each `ItemPricingCalculator` has the responsibility of manipulating the `DetailedItemPriceInfos` correctly.

The following sections describe how the three different types of item calculators interact with `DetailedItemPriceInfos`:

- Using List Price Calculators with DetailedItemPriceInfo Objects (page 133)
- Using Sale Price Calculators with DetailedItemPriceInfo Objects (page 134)
- Using Item Discount Calculators with DetailedItemPriceInfo Objects (page 135)

Using List Price Calculators with DetailedItemPriceInfo Objects

The list price calculators are responsible for calculating the price based on the list price. They can usually look up a list price and multiply it by the quantity. If the entire quantity of the item is being shipped to the same place, there will only need to be one `DetailedItemPriceInfo`. The logic for making sure that each detail only refers to items in one shipping group is contained in this method: `pricingTools.detailedItemPriceTools.createInitialDetailedItemPriceInfos`.

This method takes the following parameters:

- `TotalPrice` - The total price for which the new `DetailedItemPriceInfo` objects must account. `TotalPrice` is the price of the entire `CommerceItem` (`listPrice` * `quantity`).
- `PriceQuote` - The current working price of the `Item`. `PriceQuote` is the `ItemPriceInfo` to which the newly created details will be added.
- Item - The `CommerceItem` being priced. This is needed to get to the `ShippingGroupCommerceItemRelationships`.
- `PricingModel` - The discount that will be set in the `PricingAdjustment` (usually null).
- `Profile` - The person for whom the items are to be discounted (currently ignored).
- `Locale` - The locale in which the items are to be discounted (currently ignored).
- `ExtraParameters` - Any extra information that this method might need to set the number of the qualifying item (currently ignored).
- `AdjustmentDescription` - The adjustment description used when creating all new `PricingAdjustments` that is added to each new detail.

These parameters are the only parameters required to perform list pricing and bulk pricing. The entire quantity gets the same price. To facilitate tiered pricing, there is another version of this method that also takes a `Range` argument. This means that new details will only be created for the given range. In this case, the `TotalPrice` is the price for the range as opposed to the entire `CommerceItem`. The `ItemTieredPriceCalculator` will call this method once per tier.

This method will only modify the `DetailedItemPriceInfo` objects. It is the responsibility of the caller to update the `ItemPriceInfo`.

Using Sale Price Calculators with DetailedItemPriceInfo Objects

The calculators responsible for calculating the sale price usually change the price of the entire quantity. The calculator retrieves the sale price, subtracts the list price, and then modifies the amount accordingly.

This functionality is contained in a centralized method: `pricingTools.detailedItemPriceTools.assignSalePriceToDetails`. The `assignSalePriceToDetails` method takes the following parameters:

- `DetailedItemPriceInfos` - The list of `DetailedItemPriceInfo` objects that should be adjusted. This will usually be the entire list of details.
- `UnitSalePrice` - The sale price for a single given `CommerceItem`. The total adjustment for each detail is this amount times the quantity of the detail.
- `PriceQuote` - The current working price of `Item`. This is taken from the `ItemPriceInfo` to which the newly created details will be added. This is also ignored.
- `Item` - `CommerceItem` being priced. This is ignored in the default implementation.
- `PricingModel` - The discount that will be set in the `PricingAdjustment` (usually null).
- `Profile` - The person for whom the items are to be discounted (currently ignored).
- `Locale` - The locale in which the items are to be discounted (currently ignored).

- `ExtraParameters` - Any extra information that this method might need to set the prices of a number of the qualifying item(currently ignored).
- `AdjustmentDescription` - This is the adjustment description used when creating all new `PricingAdjustments`.

The `assignSalePriceToDetails` method walks through each detail and adjusts the amount accordingly. There is no reason to create any new details. This method will only modify the `DetailedItemPriceInfo` objects. It is the responsibility of the caller to update the `ItemPriceInfo` objects.

One sale calculator that does not use this method is the `ItemSalesTieredPriceCalculator`. This calculator splits the details since each quantity of the item will not necessarily get the same unit sale price. Therefore it usually splits each detail to maintain the property that each item in a detail was priced in exactly the same way (this includes having the same unit sale price).

Using Item Discount Calculators with DetailedItemPriceInfo Objects

The calculators that are responsible for discounting the price based on a promotion usually only adjust the price for some subset of the quantity. Therefore, the `ItemDiscountCalculator` frequently splits details. The `ItemDiscountCalculator` determines the range that is undiscounted vs. the range that is discounted. It creates a new `DetailedItemPriceInfo` for the undiscounted portion and set its quantity to the number of undiscounted items. It modifies the current detail to be discounted and changes its price. Then, with each detail, it calls `pricingTools.detailedItemPriceTools.splitDetailsAccordingtoRanges`.

This method takes the following parameters:

- `Detail`—The `DetailedItemPriceInfo` to split apart.
- `Ranges`—The list of Ranges that should have a `DetailedItemPriceInfo`. The size of these must equal the quantity of `Detail`.

This method takes the current detail and creates enough new details so that there is one per Range that is passed in. All the details are returned.

OrderPriceInfo

The `atg.commerce.pricing.OrderPriceInfo` class contains information about the price of an order. Its properties are modified by order pricing calculators and returned by order pricing engines.

For information about `OrerPriceInfo` properties, see the *ATG Platform API Reference*.

ShippingPriceInfo

The `atg.commerce.pricing.ShippingPriceInfo` class contains information about the price of a `ShippingGroup`.

For information about `ShippingPriceInfo` properties, see the *ATG Platform API Reference*.

TaxPriceInfo

The `atg.commerce.pricing.TaxPriceInfo` class represents tax price information.

For information about `TaxPriceInfo` properties, see the *ATG Platform API Reference*.

Extending Pricing Engines

Oracle ATG Web Commerce provides several preconfigured pricing engines (see Default Pricing Engines (page 130)). You can extend these engines to fit your sites' requirements, and you can also create new pricing engines if necessary.

Extending a Pricing Engine

You can extend one or more of the pricing engine implementations to provide new pricing functionality. For example, you can extend `ItemPricingEngineImpl` to create an algorithm that prices a set of items differently from the current implementation of `priceEachItem`. You could create an algorithm that applies promotions in a random order rather than in order of ascending precedence.

Because each implementation of the `PricingEngine` interface extends the `PricingEngineService` class, you can extend one or all of the implementations to alter the behavior of a method of `PricingEngineService`. For example, you could implement the `expirePromotion` method to send a JMS event enabling the creation of scenarios related to unused and expired promotions. After you complete your extensions, configure the corresponding pricing engine component to use the class. (For more information, see the description of PricingEngineService (page 130).)

Each engine can also be extended to leverage existing code. For example, you can extend the pricing engine to determine global promotions using a Personify or NetPerceptions integration. The `ItemPricingEngine` could be extended to get its global promotions from the integration.

The relevant interfaces are:

- `atg.commerce.pricing.ItemPricingEngine`
- `atg.commerce.pricing.TaxPricingEngine`
- `atg.commerce.pricing.ShippingPricingEngine`
- `atg.commerce.pricing.OrderPricingEngine`
- `atg.commerce.pricing.ItemDiscountCalculator`
- `atg.commerce.pricing.OrderDiscountCalculator`
- `atg.commerce.pricing.ShippingDiscountCalculator`

The properties of a promotion repository item are in `atg.commerce.pricing.PricingModelProperties`.

The `Qualifier` class that holds helper methods is `atg.commerce.pricing.Qualifier`.

Creating a New Pricing Engine

In the following example, you want to create a new pricing engine that prices handling costs separately from shipping. You create a `HandlingPricingEngine` that acts independently of the `ShippingPricingEngine`.

1. Create an interface called `HandlingPricingEngine` that extends `PricingEngine`.
2. Create an implementation called `HandlingPricingEngineImpl` that extends `PricingEngineService`.
3. Create a `HandlingPricingInfo` that extends the `AmountInfo` price holding class. (For more information, see AmountInfo (page 132).)

4. Create a calculator called `HandlingPricingCalculator` and implementations of it that calculate and discount handling as your business requires.

5. Modify the Promotions repository definition file (by default, `pricingModels.xml`). Add an item-descriptor for the Handling discount type and sub-descriptors for the various implementations of the `HandlingPricingCalculator` that you created.

6. Create properties files for the `HandlingPricingEngine` and each of the calculators.

Your engine is ready for use. You may also want to add `preCalculators` that calculate the base cost of handling.

10 Commerce Pricing Calculators

Pricing calculators are responsible for the following:

- Looking up the price in the catalog.
- Using information they receive from the pricing engines and from the qualifier service to determine prices.

This chapter describes the base Oracle ATG Web Commerce pricing engine classes and interfaces. It includes the following sections:

For more information on the default calculator interfaces and classes, see the *ATG Platform API Reference*.

Pricing Calculator Interfaces

This section describes the interfaces that are used as part of the Oracle ATG Web Commerce calculators. These interfaces are:

ItemPricingCalculator Interface

`atg.commerce.pricing.ItemPricingCalculator` modifies the price of a `CommerceItem`.

A calculator's `priceItem`, `priceEachItem`, or `priceItems` method is invoked by the corresponding method of the same name on the pricing engine. The calculator's `priceItem` method modifies the input `priceObjects` according to the current pricing context. The specific way in which the calculator modifies an item price varies according to individual implementations.

OrderPricingCalculator Interface

The `OrderPricingCalculator` interface, `atg.commerce.pricing.OrderPricingCalculator`, modifies the price of an order.

The `priceOrder` method of the `OrderPricingCalculator` (or calculators) is invoked by the order pricing engine that is configured to use it. The `priceOrder` method modifies the input `pPriceQuote` according to the current pricing context. The specific way in which the calculator modifies an order price varies according to individual implementations.

ShippingPricingCalculator Interface

The `ShippingPricingCalculator` interface, `atg.commerce.pricing.ShippingPricingCalculator`, modifies a price object that represents the cost of shipping for an order.

The shipping pricing engine invokes the `priceShippingGroup` method of the `ShippingPricingCalculator` (or calculators) that it is configured to use. The `priceShippingGroup` method modifies the input `pPriceQuote` according to the current pricing context. The specific way in which the calculator modifies a shipping price varies according to individual implementations.

Oracle ATG Web Commerce includes several classes that are implementations of the `ShippingPricingCalculator` interface. For example, it includes the `atg.commerce.pricing.ShippingDiscountCalculator` class, which you can use to apply a promotional discount to the shipping price of an order.

TaxPricingCalculator Interface

The `atg.commerce.pricing.TaxPricingCalculator` interface modifies the price of tax for an order.

The tax pricing engine invokes the `priceTax` method of the `TaxPricingCalculator` (or calculators). The calculator's `priceTax` method modifies the input `pPriceQuote` according to the current pricing context. The specific way in which the calculator modifies a tax price varies according to individual implementations.

Oracle ATG Web Commerce includes several classes that are implementations of the `TaxPricingCalculator` interface. For example, it includes the `atg.commerce.pricing.NoTaxCalculator` class, which you can use for situations in which a sales tax of zero is applicable for an order.

CalculatorInfoProvider Interface

The `atg.commerce.pricing.CalculatorInfoProvider` interface allows a calculator to provide information in the form of a `CalculatorInfo` object. This interface is used by default in promotions templates to dynamically update the user interface and PMDL for calculators.

If a calculator does not implement this interface, the pricing engine returns a default `CalculatorInfo` object.

Pricing Calculator Classes

These classes are used by the Oracle ATG Web Commerce pricing engines to calculate prices. They can be extended according to your needs (see Extending Pricing Calculators (page 155)).

DiscountCalculatorService

The `atg.commerce.pricing.DiscountCalculatorService` class is an extension of `GenericService`. `DiscountCalculatorService` computes a price based on the type of discount, the discount amount, and the current price. It holds information common to all the discount calculators, which can extend this class to eliminate redundant configuration code.

The `adjust` method can be used as a quick way to apply a discount. It calculates a price based on an existing price, the discount type, and the discount amount. This functionality is used by all discount calculators.

The `DiscountCalculatorService` also includes `getAdjuster()` and `getDiscountType()` methods, which make it easy to override the default means of determining the adjuster and discount types.

The following list describes important properties in the `DiscountCalculatorService` class:

- `pricingModelProperties`: Points to a configuration bean that holds the names of all the properties of a pricing model repository item.
- `negativeAmountException`: Oracle ATG Web Commerce never discounts the price of an object to less than zero. This property determines what happens when a discount would cause the amount to be negative. `True`: Throw an exception when a discount causes an amount to be negative. `False`: (default) Log a warning message and set the amount to 0.0 when a discount causes an amount to be negative.

ItemPriceCalculator

The abstract class `atg.commerce.pricing.ItemPriceCalculator` is the parent class for many item pricing calculators (for example, `ItemListPriceCalculator` and `ItemSalePriceCalculator`), consolidating the functionality that these calculators have in common.

This class determines a price for an object based on a `pricesource` object and the properties of that `pricesource`. It contains a single abstract method, `priceItems`. Extending classes implement this method to leverage the other item pricing functionality that this class provides.

The `ItemPriceCalculator` class also contains the following properties:

- `loggingIdentifier`: The ID that this class uses to identify itself in logs.
- `pricePropertyName`: The name of the property of the `priceSource` that represents an item's price. The `priceSource` is the value returned from the `getPriceSource` property.
- `requirePriceValue`: If this property is `true`, an exception is thrown if the `priceSource` of the `CommerceItem` does not have its `pricePropertyName` property set.
- `priceFromCatalogRef`: If this property is `true`, `getPriceSource` returns the `catalogRef` property of the input `CommerceItem`. If this property is `false`, `getPriceSource` returns the `productRef` property.

`ItemPriceCalculator` determines an item's price based on a property value. The property value comes from a `priceSource` object. The `priceSource` object can be one of two objects, depending on the boolean value of the `priceFromCatalogRef` property.

- If `true`, the `priceSource` object is the `catalogRef` property of the item to be priced.
- If `false`, the `priceSource` object is the `productRef` property of the item to be priced.

The `catalogRef` property of items to be priced points to a SKU in the SKU catalog. The `productRef` property points to a product in the product catalog. For more information, see the `atg.commerce.order.CommerceItem` entry in the *ATG Platform API Reference*.

The `pricePropertyName` in the `ItemPriceCalculator` (and therefore the `ItemListPriceCalculator` and `ItemSalePriceCalculator`, which extend `ItemPriceCalculator`) identifies the property on the `priceSource` object (the SKU or product) that contains the price for the current item. The `ItemListPriceCalculator` and `ItemSalePriceCalculator` read this value and return a price object that contains the price.

How Classes that Extend ItemPriceCalculator Determine Prices

The `ItemListPriceCalculator` extends `ItemPriceCalculator`. In this example, the `ItemListPriceCalculator` is set with the following properties:

- `priceFromCatalogRef=true` (indicating it will get its property from a SKU)
- `pricePropertyName=listPrice`.

In this example, the SKU catalog contains a SKU that identifies blue shorts. That SKU has the following properties:

- `color` = blue
- `itemType` = shorts
- `listPrice` = 10.00

When a `CommerceItem` is passed to the `ItemListPriceCalculator`, this particular `Itemlistpricecalculator` looks at the item's `catalogRef` property (SKU) and retrieves the value of that object's `listPrice` property (10.00). The `ItemListPriceCalculator` then modifies the input price to be 10.00 and returns it.

The `ItemSalePriceCalculator` works in almost the same way. The `ItemSalePriceCalculator` has an additional property called `onSalePropertyName`, which is a boolean property on the `priceSource` object that indicates whether the item is on sale.

In this example, the `priceFromCatalogRef` property of `ItemSalePriceCalculator` is set to `true`. The `pricePropertyName` property is set to `salePrice`. A SKU in the SKU catalog has a property called `onSale`. If a SKU were on sale, the `onSale` property would be set to `true`. The `onSalePropertyName` property of `ItemSalePriceCalculator` is set to `onSale`.

When an item is passed to the `ItemSalePriceCalculator`, it has a `catalogRef` property that points to a SKU from the SKU catalog.

- `color=blue`
- `itemType=shorts`
- `listPrice=10.00`
- `onSale=true`
- `salePrice=7.00`

When the `ItemSalePriceCalculator` receives this item, it uses the value of the SKU's `onSale` property to determine if it is on sale. In this example, the value is `true`, indicating that the SKU is on sale. The calculator then gets the sale price using the SKU's `salePrice` property. The price in this case is 7.00. The calculator then modifies the input price for the item to be 7.00 and registers that a change has been made to the price. This registering is done by creating a new `PricingAdjustment` and adding it to the adjustments property of the price object, which in this case is an `ItemPriceInfo`. The adjustment would be -3.00, and it would show that the `ItemSalePriceCalculator` was responsible for the change.

ItemDiscountCalculator

The `atg.commerce.pricing.ItemDiscountCalculator` class calculates the new price for an item or items based on a given pricing model. It applies the discount that the pricing model describes to the price in the `ItemPriceInfo` corresponding to each passed `CommerceItem`. The discount can be a fixed amount, a percentage, or a fixed price.

The `ItemDiscountCalculator` class inherits all the properties of DiscountCalculatorService (page 141). See the *ATG Platform API Reference* for detailed information on `ItemDiscountCalculator` and its related classes.

The `ItemDiscountCalculator` component is a preconfigured instance of the class `atg.commerce.pricing.ItemDiscountCalculator` class. It is the default discount calculator for the item discount promotions.

The following table describes the properties of the `ItemDiscountCalculator` component.

Property	Description
`pricingModelProperties`	Specifies a bean that hosts the names of all of the properties of a pricing model repository item.
`qualifierService`	Specifies a Qualifier that performs the actual evaluation of a `pmdlRule` of the `PricingModel` against the running environment. See the Qualifier Properties (page 160) section for additional information.
`negativeAmountException`	Oracle ATG Web Commerce never discounts the price of an item to less than zero. This property determines what happens when a discount would cause the amount of an item to be negative. `True`: Throw an exception when a discount causes an amount to be negative. `False`: (default) Log a warning message and set the amount to 0.0 when a discount causes an amount to be negative.

You can view and modify this component in the ATG Control Center. Its location is `/atg/commerce/pricing/calculators/ItemDiscountCalculator`.

BulkItemDiscountCalculator

The `atg.commerce.pricing.BulkItemDiscountCalculator` class is a calculator that supports bulk item discounts. This class is based on the `ItemDiscountCalculator`. The unique behavior of `BulkItemDiscountCalculator` is to determine the adjuster for the discount; other functionality is inherited.

The calculator has two default properties for banding attributes, in case those are not provided in the PMDL:

```
defaultBandingProperty = null
defaultBandingPropertyScope = "DetailedItemPriceInfo"
```

The `defaultBandingPropertyScope` provides access to the collection of qualified items for the calculator to process.

The calculator's `bandedDiscountCalculatorHelper` points to a helper class, `BandedDiscountCalculatorHelper`, that holds the banded discount logic. See the BandedDiscountCalculatorHelper (page 153) section for details.

ItemListPriceCalculator

The `atg.commerce.pricing.ItemListPriceCalculator` class is a calculator that determines the list price of an item and sets the `itemPriceInfo` to that amount. This class extends the ItemPriceCalculator (page 142). The list price is determined based on the value returned from the `getPriceSource` property. This is typically the first in a series of calculations, with this calculator providing a starting price for other calculators. The `ItemListPriceCalculator` sets the `listPrice` property of the input price object to the input `Price`.

The ItemPriceCalculator (page 142) section includes an example of how the `ItemListPriceCalculator` determines a price.

ItemSalePriceCalculator

The `atg.commerce.pricing.ItemSalePriceCalculator` class extends the `ItemPriceCalculator`. It determines the sale price of an item and discounts the `itemPriceInfo` to that amount. This class also maintains the audit trail of the `ItemPriceInfo`. There is no rule associated with this calculator. If one of the pricing methods of `ItemSalePriceCalculator` is invoked, all input items are discounted to the sale price.

The `ItemSalePriceCalculator` class also contains the following property:

- `onSalePropertyName`: The boolean property of the price source that determines if the price source is on sale. A price source is the `catalogRef` or `productRef` of a `CommerceItem`.

The ItemPriceCalculator (page 142) section includes an example of how the `ItemSalePriceCalculator` determines a price.

ConfigurableItemPriceCalculator

The `atg.commerce.pricing.ConfigurableItemPriceCalculator` class extends the `ItemPriceCalculator`. It calculates the list price of a configurable item and sets the `itemPriceInfo` to that amount. It computes the price of the configurable item by adding up the price of all the individual subSKUs and the price of the configurable SKU. This sets the list price with the prices of the subSKUs that are configured in the configurable item. If the configurable item is on sale then the sale price will also be modified.

OrderDiscountCalculator

The `atg.commerce.pricing.OrderDiscountCalculator` class implements the `OrderPricingCalculator`. It calculates `OrderPriceInfo` values for orders when the calculator is invoked. This calculator receives a `MatchingObject` and `DiscountStructure` from the pricing engine, then computes an `OrderPriceInfo` based on the input `PricingModel` (`RepositoryItem`).

See the *ATG Platform API Reference* for detailed information on `OrderDiscountCalculator` and its related classes.

The `OrderDiscountCalculator` component is a preconfigured instance of the class `atg.commerce.pricing.OrderDiscountCalculator`. It is the default discount calculator for order promotions.

The following table describes the properties of the `OrderDiscountCalculator` component.

Property	Description
`pricingModelProperties`	Specifies a bean that hosts the names of all of the properties of a pricing model repository item. `pricingModelProperties` are used so you do not have to hard code the properties into a pricing model.
`qualifierService`	Specifies a Qualifier that performs the actual evaluation of a `pmdlRule` of the `PricingModel` against the running environment.

Property	Description
negativeAmountException	Determines what happens when discounts cause the amount of an item to be negative. True: Throw an exception when a discount causes an amount to be negative. False: (default) Log a warning message and set the amount to 0.0 when a discount causes an amount to be negative.

You can view and modify the OrderDiscountCalculator component in the ATG Control Center. The component is located in /atg/commerce/pricing/calculators/OrderDiscountCalculator.

BulkOrderDiscountCalculator

The atg.commerce.pricing.BulkOrderDiscountCalculator class is a calculator that supports bulk item discounts. This class is based on the OrderDiscountCalculator. The unique behavior of BulkOrderDiscountCalculator is to determine the adjuster for the discount; other functionality is inherited.

The calculator has a default property for banding attributes, in case those are not provided in the PMDL:

```
defaultBandingProperty = "OrderPriceInfo.amount"
```

The defaultBandingProperty provides access to the price for the calculator to process.

The calculator's bandedDiscountCalculatorHelper points to a helper class, BandedDiscountCalculatorHelper, that holds the banded discount logic. See the BandedDiscountCalculatorHelper (page 153) section for information.

OrderSubtotalCalculator

The atg.commerce.pricing.OrderSubtotalCalculator class computes the rawSubtotal and amount of an OrderPriceInfo that corresponds to the input order. Unlike in the case of discount calculators, there is no rule that determines whether the subtotal should be calculated. The order's subtotal is always calculated by summing the prices of the items in the order. If a pricing model is passed in, it is ignored.

ShippingCalculatorImpl

The atg.commerce.pricing.ShippingCalculatorImpl class is an abstract class that acts as a starting point for general price calculation in shipping groups. The implementation of priceShippingGroup checks that there are items in the shipping group. If there are no items, the price quote is reset to zero. If there are items to price for shipping, the performPricing method confirms that the items in the group should be priced. For example, soft goods, such as gift certificates, should not be priced for shipping.

When extending this class, implement the getAmount method as the base shipping cost in this calculator.

The amount returned is set into the ShippingPriceInfo. If the addAmount property is true, the amount returned is added to the current ShippingPriceInfo amount. This behavior allows for the addition of surcharges.

Set the `shippingMethod` property to the name of a particular delivery process that your sites offer, for example ground, 2-day or next day. If the `ignoreShippingMethod` property is `true`, then the calculator does not expose a shipping method name (through `getAvailableMethods`). In addition, this calculator always attempts to perform pricing. This option is useful for situations in which you do not want to give customers a choice of different shipping methods.

ShippingDiscountCalculator

The `atg.commerce.pricing.ShippingDiscountCalculator` class calculates `ShippingPriceInfos` for specified `ShippingGroups`. This calculator receives a `MatchingObject` and `DiscountStructure` from the pricing engine, then computes a `ShippingPriceInfo` based on the input `PricingModel` (`RepositoryItem`).

See the *ATG Platform API Reference* for detailed information on `ShippingDiscountCalculator` and its related classes.

The `ShippingDiscountCalculator` component is a preconfigured instance of the class `atg.commerce.pricing.ShippingDiscountCalculator`. It is the default discount calculator for shipping promotions.

The following table describes the properties of the `ShippingDiscountCalculator` component.

Property	Description
`pricingModelProperties`	Specifies a bean that hosts the names of all of the properties of a pricing model repository item.
`qualifierService`	Specifies a Qualifier that performs the actual evaluation of a `pmdlRule` of the `PricingModel` against the running environment.
`negativeAmountException`	Determines what happens when discounts cause the amount of an item to be negative. `True`: Throw an exception when a discount causes an amount to be negative. `False`: (default) Log a warning message and set the amount to 0.0 when a discount causes an amount to be negative.

You can view and modify the `ShippingDiscountCalculator` component in the ATG Control Center. The component is located in `/atg/commerce/pricing/calculators/ShippingDiscountCalculator`.

BulkShippingDiscountCalculator

The `atg.commerce.pricing.BulkShippingDiscountCalculator` class is a calculator that supports bulk item discounts. This class is based on the `ShippingDiscountCalculator`. The unique behavior of `BulkShippingDiscountCalculator` is to determine the adjuster for the discount; other functionality is inherited.

The calculator has a default property for banding attributes, in case those are not provided in the PMDL:

```
defaultBandingProperty = "OrderPriceInfo.amount"
```

The `defaultBandingProperty` provides access to the shipping price for the calculator to process.

The calculator's `bandedDiscountCalculatorHelper` method points to a helper class, `BandedDiscountCalculatorHelper`, that holds the banded discount logic. See BandedDiscountCalculatorHelper (page 153) in this chapter.

PriceRangeShippingCalculator

The `atg.commerce.pricing.PriceRangeShippingCalculator` class determines the shipping price based on the subtotal of all the items in the shipping group. The service is configured through the `ranges` property. With the given array of price range configurations (format: `low:high:price`) the service parses the values into their double format for calculating shipping costs. For example:

```
ranges=00.00:15.99:4.50,\
        16.00:30.99:6.00,\
        31.00:40.99:7.25,\
        41.00:MAX_VALUE:10.00
```

Note: The keyword `MAX_VALUE` indicates the maximum possible value in the range.

The `PriceRangeShippingCalculator` also contains the following properties:

- `addAmount`: If the property `addAmount` is `true`, instead of setting the price quote amount to the value of the amount property, the calculator adds the amount to the current amount in the price quote. This can be used to configure a "surcharge" calculator, which increases the shipping price.
- `shippingMethod`: The `shippingMethod` property is set to the name of a particular delivery process. For example: UPS Ground, UPS 2-day or UPS Next Day.
- `ignoreShippingMethod`: Setting the `ignoreShippingMethod` property to `true` prevents this calculator from exposing the shipping method name (through `getAvailableMethods`). In addition, this calculator always attempts to perform pricing. This option is useful for situations in which you do not want to give customers a choice of different shipping methods.

DoubleRangeShippingCalculator

This `atg.commerce.pricing.DoubleRangeShippingCalculator` class is an abstract shipping calculator that determines the shipping price by comparing a value from the `ShippingGroup` to a series of ranges. The service is configured through the `ranges` property. It is extended by the `PriceRangeShippingCalculator`, `PropertyRangeShippingCalculator`, and `WeightRangeShippngCalculator` classes.

With the given array of price range configurations (format: `low:high:price`), the service parses the values into their double format for calculating shipping costs. For example:

```
ranges=00.00:15.99:4.50,\
        16.00:30.99:6.00,\
        31.00:40.99:7.25,\
        41.00:MAX_VALUE:10.00
```

Note: The keyword `MAX_VALUE` indicates the maximum possible value in the range.

The `DoubleRangeShippingCalculator` also contains the following properties:

- `addAmount`: If the property `addAmount` is `true`, instead of setting the price quote amount to the value of the `amount` property, the calculator adds the amount to the current amount in the price quote. This behavior can be used to configure a "surcharge" calculator, which increases the shipping price.
- `shippingMethod`: The `shippingMethod` property is set to the name of a particular delivery process, for example ground, 2-day or next day.
- `ignoreShippingMethod`: Setting the `ignoreShippingMethod` property to `true` prevents this calculator from exposing the shipping method name (through `getAvailableMethods`). In addition, this calculator always attempts to perform pricing. This option is useful for situations in which you do not want to give customers a choice of different shipping methods.

FixedPriceShippingCalculator

The `atg.commerce.pricing.FixedPriceShippingCalculator` class is a shipping calculator that sets the shipping amount to a fixed price.

The `FixedPriceShippingCalculator` also contains the following properties:

- `addAmount`: If the property `addAmount` is `true`, instead of setting the price quote amount to the value of the `amount` property, the calculator adds the amount to the current amount in the price quote. This behavior can be used to configure a "surcharge" calculator, which increases the shipping price.
- `shippingMethod`: The `shippingMethod` property is set to the name of a particular delivery process. For example: ground, 2-day or next day.
- `ignoreShippingMethod`: Setting the `ignoreShippingMethod` property to `true` prevents this calculator from exposing the shipping method name (through `getAvailableMethods`). In addition, this calculator always attempts to perform pricing. This option is useful for situations in which you do not want to give customers a choice of different shipping methods.

PropertyRangeShippingCalculator

The `atg.commerce.pricing.PropertyRangeShippingCalculator` class is a highly flexible shipping calculator that identifies an item property and adds the value provided to each item in the shipping group together to create a shipping group total. The total falls into one of the ranges specified in the `ranges` property, which provides a shipping cost for each range.

For example, all items may have a property called `weight` that correlates to the weight of the item in pounds. To base shipping cost on the cumulative weight total, you set the `PropertyRangeShippingCalculator` `propertyName` property to `weight`. If your shipping group has three items, each of which has a weight of 1, Oracle ATG Web Commerce calculates a total of three and uses the `ranges` property to determine how much to charge. The range property takes the format of `low:high:price and holds these options:`

```
ranges=00.00:15.99:4.50,\
        16.00:30.99:6.00,\
        31.00:40.99:7.25,\
        41.00:MAX_VALUE:10.00
```

In this example, shipping charges total $4.50. Note that keyword `MAX_VALUE` indicates the maximum possible value in the range.

The `PropertyRangeShippingCalculator` also contains the following properties:

- `addAmount`: If the property `addAmount` is `true`, instead of setting the price quote amount to the value of the amount property, the calculator adds the amount to the current amount in the price quote. This can be used to configure a "surcharge" calculator, which increases the shipping price.
- `shippingMethod`: The `shippingMethod` property is set to the name of a particular delivery process. For example: ground, 2-day or next day.
- `ignoreShippingMethod`: Setting the `ignoreShippingMethod` property to `true` prevents this calculator from exposing the shipping method name (through `getAvailableMethods`). In addition, this calculator always attempts to perform pricing. This option is useful for situations in which you do not want to give customers a choice of different shipping methods.
- `propertyName`: Set the `propertyName` property to the name of the property that you want to add across all items. For example, "weight" would calculate the total weight of an order by adding together the `weight` property values of all the items.
- `useCatalogRef`: If the `useCatalogRef` property is set to `true`, the property value is extracted from the `catalogRef` of the `CommerceItem` (usually the SKU). If `useCatalogRef` property is set to `false`, the `product` is used as the source.

WeightRangeShippingCalculator

The `atg.commerce.pricing.WeightRangeShippingCalculator` class is a shipping calculator that determines the shipping price based on the sum of the weights of each item in a shipping group.

The service is configured through the `ranges` property. With the given array of price range configurations (format: `low:high:price`), the service parses the values into their double format for calculating shipping costs. For example:

```
ranges=00.00:15.99:4.50,\
       16.00:30.99:6.00,\
       31.00:40.99:7.25,\
       41.00:MAX_VALUE:10.00
```

Note: The keyword `MAX_VALUE` indicates the maximum possible value in the range.

The `WeightRangeShippingCalculator` also contains the following properties:

- `addAmount`: If the property `addAmount` is `true`, instead of setting the price quote amount to the value of the `amount` property, the calculator adds the amount to the current amount in the price quote. This can be used to configure a "surcharge" calculator, which increases the shipping price.
- `shippingMethod`: The `shippingMethod` property is set to the name of a particular delivery process. For example: ground, 2-day or next day.
- `ignoreShippingMethod`: Setting the `ignoreShippingMethod` property to `true` prevents this calculator from exposing the shipping method name (through `getAvailableMethods`). In addition, this calculator always attempts to perform pricing. This option is useful for situations in which you do not want to give customers a choice of different shipping methods.

NoTaxCalculator

The `atg.commerce.pricing.NoTaxCalculator` class creates a new `TaxPriceInfo` object that specifies a tax price of zero for an order.

TaxDiscountCalculator

Use the `atg.commerce.pricing.TaxDiscountCalculator` to calculate `TaxPriceInfo` objects for `Orders`. This calculator receives a `MatchingObject` and `DiscountStructure` from the pricing engine, then computes a `TaxPriceInfo` based on the input `PricingModel`.

See the *ATG Platform API Reference* for detailed information on `TaxDiscountCalculator` and its related classes.

BulkTaxDiscountCalculator

The `atg.commerce.pricing.BulkTaxDiscountCalculator` class is a calculator that supports bulk item discounts. This class is based on the `TaxDiscountCalculator`. The unique behavior of `BulkTaxDiscountCalculator` is to determine the adjuster for the discount; other functionality is inherited.

The calculator has a default property for banding attributes, in case those are not provided in the PMDL:

```
defaultBandingProperty = "OrderPriceInfo.amount"
```

The `defaultBandingProperty` provides access to the shipping price for the calculator to process.

The calculator's `bandedDiscountCalculatorHelper` method points to a helper class, `BandedDiscountCalculatorHelper`, that holds the banded discount logic. See the BandedDiscountCalculatorHelper (page 153) section in this chapter.

TaxProcessorTaxCalculator

You use the `atg.commerce.pricing.TaxProcessorTaxCalculator` class if you are setting up a site that uses third-party software to handle tax calculation. This class consults a `TaxProcessor` (an implementation of the `atg.payment.tax.TaxProcessor` interface) to determine how much tax to charge for an order.

The `TaxProcessorTaxCalculator` component is located in the ATG Control Center at `atg/commerce/pricing/calculators`. `TaxProcessorTaxCalculator` class has the following properties:

- `taxStatusProperty`: The property in the SKU repository that indicates whether each SKU is taxable or not.
- `taxProcessor`: Your custom tax calculator component. **Note:** The default `TaxProcessor` for Oracle ATG Web Commerce is `/atg/commerce/payment/DummyTaxProcessor`, which always returns "no tax."
- `orderManager`: The location of the `OrderManager` class instance. The default is `/atg/commerce/order/OrderManager`.
- `pricingTools`: The location of the `PricingTools` class instance. The default is `/atg/commerce/pricing/PricingTools`.
- `verifyAddresses`: If `true`, the `TaxProcessor` verifies the addresses passed in before attempting to calculate tax.

- calculateTaxByShipping: If true, the calculator calculates tax by shipping group. If false, tax is calculated by total order.

Price List ConfigurableItemPriceListCalculator

The ConfigurableItemPriceListCalculator calculator assumes the ItemListPriceCalculator has already run. ItemListPriceCalculator calculator sets the list price and the amount of the ItemPriceInfo based on the price of the ConfigurableSku. The ConfigurableItemPriceListCalculator calculator then iterates through the subSKUs and modifies the list price and amount accordingly.

For example, consider a situation when a parentSKU is $5, subSKU A is $2 and subSKU B is $1. If we buy two of this configurable SKU, then coming into this calculator the listPrice will be $5 and the amount will be $10. After the ConfigurableItemPriceListCalculator calculator runs the listPrice will be $8 and the amount will be $16.

The ConfigurableItemPriceListCalculator component is located in the ACC at atg/commerce/pricing/calculators/ConfigurableItemPriceListCalculator. For more information on this calculator, see the Price List Calculators (page 215) section of the *Using Price Lists* (page 211) chapter.

Price List ItemListPriceCalculator

This calculator determines the list price of an item and sets the itemPriceInfo to that amount. The pricing scheme for that item is list pricing.

The ItemListPriceCalculator component is located in the ACC at /atg/commerce/pricing/calculators/ItemListPriceCalculator. For more information on this calculator. See the Price List Calculators (page 215) section of the *Using Price Lists* (page 211) chapter.

Price List ItemPriceCalculator

The ItemPriceCalculator class can either price a single commerce item or price a list of commerce items. It first selects the priceList to use based on the profilePriceListPropertyName property. The ItemPriceCalculator then delegates the pricing to different ItemSchemePriceCalculators based on the item's pricing scheme by the pricingSchemePropertyName property.

The ItemPriceListCalculator component is located in the ACC at /atg/commerce/pricing/calculators/ItemPriceListCalculator. It has the following properties:

- loggingIdentifier: the ID that this class uses to identify itself in logs.
- pricingSchemePropertyName: the property name in the repository for the pricing scheme.
- profilePriceListPropertyName: the property name in the repository for the user's price list.
- useDefaultPriceList: If true and ProfilePriceListPropertyName is null, then the value of the automaticallyUseDefaultPriceList property of the PriceListManager determines if the default price list is used. If false, then the default price list is never used.
- noPriceIsError: If true, and the price list is null or there is no price in the price list, then an error is thrown. If false, and the price list is null, then nothing happens.
- priceListManager: points to the location of the PriceListManager.

- `pricingSchemeNames`: the Map whose key is the allowed scheme names and whose value is the corresponding calculators.

For more information on this calculator. See the Price List Calculators (page 215) section of the *Using Price Lists* (page 211) chapter.

Price List ItemSalesPriceCalculator

The `ItemSalesPriceCalculator` sets the sales price for a commerce item. The `ItemSalesPriceCalculator` implements the `SalePriceListsListCalculator` component, which is located in the ACC at `atg/commerce/pricing/calculators/SalePriceListsListCalculator`.

For more information on this calculator. See the Price List Calculators (page 215) section of the *Using Price Lists* (page 211) chapter.

Price List ItemSalesTieredPriceCalculator

A calculator which determines the sales tiered price of an item and sets the `itemPriceInfo` to be that amount. The definition of tiered pricing can be referenced in `ItemTieredPriceCalculator`.

The `ItemSalesTieredPriceCalculator` implements the `SalePriceListsTieredCalculator` component, which is located in the ACC at `atg/commerce/pricing/calculators/SalePriceListsTieredCalculator`.

For more information on this calculator, see the Price List Calculators (page 215) section of the *Using Price Lists* (page 211) chapter.

Price List ItemTieredPriceCalculator

The `ItemTieredPriceCalculator` determines the tiered price of an item and sets the `itemPriceInfo` to that amount. The pricing scheme for that item is tier pricing. For more information on tiered pricing, see the *Using Price Lists* (page 211) section of this chapter.

The `ItemTieredPriceCalculator` implements the `PriceListsTieredCalculator` component, which is located in the ACC at `atg/commerce/pricing/calculators/PriceListsTieredCalculator`.

For more information on this calculator, see the Price List Calculators (page 215) section of the *Using Price Lists* (page 211) chapter.

BandedDiscountCalculatorHelper

The `BandedDiscountCalculatorHelper` class is used by `BulkItemDiscountCalculator`, `BulkOrderDiscountCalculator`, `BulkShippingDiscountCalculator`, and `BulkTaxDiscountCalculator`.

```
public class BandedDiscountCalculatorHelper extends ApplicationLoggingImpl
{
  public double getAdjuster(
     RepositoryItem pPricingModel,
     Collection pQualifiedItems,
     Map pExtraParameters,
```

```
    String pDefaultBandingProperty,
    String pDefaultBandingPropertyScope) throws PricingException
}
```

The function of the helper class is to get adjuster information for bulk discounts. The `getAdjuster()` method does the following:

1. Gets the discount structure from the extra parameters map using the key `Constants.DISCOUNT_STRUCTURE`.
2. Returns the adjuster value if it has already been calculated.
3. Get all the discount details from the discount structure, which allows the class to identify and sort the bands to use.
4. Identify which property to use for placing items within the bands.
5. Identify the `QualifiedItems` for the promotion and their quantities (if applicable) to determine the banding value.
6. Set the `adjuster` value for that band on the discount structure object before returning the adjuster value.

GWPPriceCalculator

The `atg.commerce.pricing.GWPPriceCalculator` is located at `/atg/commerce/pricing/calculators/GWPPriceCalculator` in Nucleus. The calculator checks for commerce items that have Gift with Purchase commerce item markers (see the Gift with Purchase Promotions (page 181) section of the *Understanding Promotions* (page 169) chapter in this guide).

By default this calculator is used as a pre-calculator. For any commerce items that include a Gift with Purchase marker, the calculator retrieves the promotion ID and the free quantities from the marker. The calculator then adjusts the item's pricing to free.

A single commerce item can be used for multiple promotions; therefore, there can be multiple markers on the commerce item. Each marker is processed separately and results in an additional quantity being marked free.

The GWP calculator can target existing items in the cart to be free but otherwise does not update the order at all; it just updates a `Map` of `GWPInfo` objects. All order updates to add/remove commerce item quantity and markers are handled by the `GWPManager` component.

GWPDiscountCalculator

The `atg.commerce.pricing.GWPDiscountCalculator` is located at `/atg/commerce/pricing/calculators/GWPDiscountCalculator`. The calculator is responsible for discounting existing targeted items in the cart, and creating and updating `GWPInfo` objects for the current pricing operation.

To use this calculator, set the `calculatorTypeCalculators` property of the `ItemPricingEngine` to `gwp`.

CalculatorInfo

The `CalculatorInfo` object is used by the `Bulk`*`Type`*`DiscountCalculator` classes. It provides access to the following information for a calculator:

- `CalculatorType`—Corresponds to the key of the `calculatorTypeCalculators` map in the pricing engine.
- `DiscountStructureAttributeInfos`—An array of attributes specified under the `discount-structure` in the PMDL, such as `bandingProperty` or `bandingPropertyScope`.
- `DiscountDetailAttributeinfos`—An array of attributes that can be specified per discount-detail in the PMDL, such as `band` or `adjuster`.
- `DiscountTypes`—Discount types valid for this calculator, such as `fixedPrice` or `percentOff`.

For information on the `discount-structure` in PMDL, see the Understanding PMDL Discount Rules (page 177) section.

Extending Pricing Calculators

Oracle ATG Web Commerce provides several preconfigured pricing calculators (see Pricing Calculator Classes (page 140) for more information). You can extend these calculators to fit your sites' requirements, and you can also create new pricing calculators if necessary.

Adding a New Pricing Calculator

This section explains how to create a new implementation of a pricing calculator interface and how to use the new calculator.

Creating a New Pricing Calculator

Use the following interfaces to create a new pricing calculator that fits into the existing Oracle ATG Web Commerce pricing architecture:

- `atg.commerce.pricing.ItemPricingCalculator`
- `atg.commerce.pricing.OrderPricingCalculator`
- `atg.commerce.pricing.ShippingPricingCalculator`
- `atg.commerce.pricing.TaxPricingCalculator`

Implement the interface that corresponds to the type of price you want to calculate. For example, if you want to make calculations on order prices, implement `OrderPricingCalculator`.

In the following example, assume you have identified a need for a calculator that sets an item's price to half its current price plus one. The existing Oracle ATG Web Commerce tools include an `ItemDiscountCalculator` that discounts items. It can give a percent off a price or an amount off, or it can set an item's price to a fixed amount. None of these three options, however, easily gives a "half off plus one" discount. To achieve that result, you would have to use two different discounts: one to give 50 percent off, and another to add 1 to that total. A better alternative would be to create a new calculator that discounts an item's price to half its current price plus one.

To create the new calculator, you create a class called `HalfPlusOneItemDiscountCalculator` that implements the `ItemPricingCalculator` interface. The `HalfPlusOneDiscountCalculator` is an example

of a discount calculator that leverages existing Oracle ATG Web Commerce functionality to perform its own unique task.

In addition to modifying an item's price, the Commerce item discount calculators also do the following:

- Maintain an audit trail of the changes made to the price using the `PricingAdjustments` property in the `AmountInfo` price object.
- Mark items that have been discounted and therefore may not be eligible for additional discounts.
- Maintain `DetailedItemPriceInfo` objects.

The `HalfPlusOneDiscountCalculator` leverages all the above functionality from the `ItemDiscountCalculator`. The only method it overrides is the `findAdjustedPrice` method, which modifies an input `DetailedItemPriceInfo` to be the right amount. In this case, the class modifies the price of the detail to half its current price plus one.

Using a New Pricing Calculator

After you have created a new calculator, you must associate it with its corresponding pricing engine. You can do this in one of the following ways:

- Add the calculator to an engine's list of pre- or post-calculators. The configuration invokes the calculator on the price of every item that passes through the engine.
- Add the calculator to the `PricingEngineService CalculatorType` map, along with the type of calculation it provides.

Extending Calculators

You can extend any of the pricing calculators to add functionality, if that suits your needs more than implementing one of the provided interfaces. As an example this section describes the order of calls in `ItemDiscountCalculator`; other calculators may vary:

1. The `priceItems` method changes the price of input items. It first calls `findQualifyingItems` to get items whose prices need changing. It then calls `priceQualifyingItems` to change their price.
2. The `findQualifyingItems` method selects items to be discounted. It bases item selection on attributes of the input environment as represented by the method's parameters.
3. The `priceQualifyingItems` method modifies the prices of an input collection of items. It also verifies that the items' audit trail is maintained. `priceQualifyingItems` calls `priceQualifyingItem` once for each input item to be priced.
4. The `priceQualifyingItem` method modifies the price of the input item. It also verifies that the item's audit trail is maintained. `priceQualifyingItem` calls each `priceDetailedItemPriceInfo`, discounting details until the total number of items to discount has been reached.
5. The `priceDetailedItemPriceInfo` method modifies the price of a `detailedItemPriceInfo`. It maintains the audit trail and marks the details that have acted as qualifiers. It calls `findAdjustedPrice` to find the new price of the details.
6. The `findAdjustedPrice` method produces a number that is the new price of a `DetailedItemPriceInfo`. It examines the existing price and the input parameters, and it returns the new price.

You can override any of these methods to provide new functionality while leveraging the existing code.

- Override `findQualifyingItems` to change the way the calculator finds the items to discount.
- Override `priceQualifyingItems` to change how a group of `ItemPriceInfos` are adjusted.
- Override `priceQualifyingItem` to change how an individual `ItemPriceInfo` is adjusted.
- Override `priceDetailedItemPriceInfo` to change how a `DetailedItemPriceInfo` within an `ItemPriceInfo` is adjusted.
- Override `findAdjustedPrice` to change how the calculator determines new prices.

11 Qualifier Class

The `atg.commerce.pricing.Qualifier` class is a helper class for discount calculators. An instance of the `Qualifier` class is sometimes referred to as a qualifier service, because `Qualifier` extends `atg.nucleus.GenericService`.

This chapter contains the following sections:

Qualifier Class Overview

Each pricing engine calls a corresponding helper method in the `Qualifier` class to determine the objects to which it should apply a discount. The `Qualifier` determines whether anything qualifies for the discount and then figures out which pieces should receive the discount.

The `Qualifier` determines which things should change and how they should be discounted; it does not modify the actual prices. The pricing engine passes the parameters that make up its pricing context to the qualifier, and goes through the result set returned by the qualifier, calling the correct calculator to discount each element as appropriate.

- `ItemPricingEngine` uses the qualifier's `findQualifyingItems` method, which returns a Collection of `QualifiedItems`. Each `QualifiedItem` maps a `DetailedItemPriceInfo` to the list of Ranges that qualified for the discount. The `CommerceItem` containing these `DetailedItemPriceInfos` is also stored. For more information, see the QualifiedItem Class (page 164).
- `OrderPricingEngine` uses the qualifier's `findQualifyingOrder`method, which returns a `MatchingObject` identifying the `Order` to discount.
- `ShippingPricingEngine` uses the qualifier's `findQualifyingShipping` method, which returns a `MatchingObject` object identifying the `ShippingGroup` to discount.

Each method determines its result set by comparing the PMDL rule of a given promotion to the input item or items.

The `doFilters()` method evaluates items in the qualifier and target rules. It takes an array of filter components, which act to disqualify items from the qualifier or target; see the Qualifier Properties (page 160) section for descriptions of the filters.

Qualifier Properties

The `Qualifier` class contains the following properties:

- `PMDLCache`—The cache that maps pricing model `RepositoryItems` to their parsed PMDL bean representations.
- `pricingModelProperties`—A list of the names of the properties of a pricing model `RepositoryItem`.

The following `Qualifier` class properties determine the objects that the `evaluateQualifier` method can use to evaluate the qualifier element of a PMDL rule. (For more information, see Replacing the Way a PMDL Rule Is Evaluated (page 166).)

- `filterForQualifierNegativePrices`—Determines whether items with negative prices can act as qualifiers. If this property is set to `true` (the default value), negative prices cannot act as qualifiers.
- `filterForQualifierZeroPrices`—Determines whether items with zero prices can act as qualifiers. If this property is set to `true` (the default value), zero prices cannot act as qualifiers.
- `filterForQualifierDiscountedByCurrentDiscountId`—Determines whether items discounted by the current discount can act as qualifiers. If this property is set to `true` (the default value), items discounted by the current discount cannot act as qualifiers.
- `filterForQualifierDiscountedByAnyDiscountId`—Determines whether items discounted by any discount can act as qualifiers. If this property is set to `true` (the default value), it masks the `filterForQualifierDiscountedByCurrentDiscountId` property.

  The following example demonstrates how the `filterForQualifierDiscountedByAnyDiscountId` works. In this example, the following three promotions are being applied to an order:

  - Promotion #1: Buy 1 orange, get 1 apple for $1
  - Promotion #2: Buy 1 apple, get 1 banana for $1
  - Promotion #3: Buy 1 banana, get 1 plum for $1

  The order in this example is for one orange, one apple, one banana, and one plum. The value of the `filterQualifierDiscountedByAnyDiscountId` changes the way the promotions are applied in the following ways:

  - If `filterQualifierDiscountedByAnyDiscountId` is false, then the orange is list price, discounting the apple, banana and plum to $1 each.
  - If `filterQualifierDiscountedByAnyDiscountId` is true, then the orange is list price, discounting the apple to $1, and the banana is list price (since the apple was discounted), discounting the plum to $1.
- `filterForQualifierOnSale`—Indicates whether items that were priced with a sale price should be allowed to act as qualifiers. This property is set to False by default.

The following `Qualifier` class properties determine the items that the `evaluateTarget` method can use to evaluate the target element of a PMDL rule. (For more information, see Replacing the Way a PMDL Rule Is Evaluated (page 166).)

- `filterForTargetNegativePrices`—Determines whether items with negative prices can receive the discount. The default value is `true`.
- `filterForTargetZeroPrices`—Determines whether items with zero prices can receive the discount. The default value is `true`.

- `filterForTargetDiscountedByCurrentDiscountId`—Determines whether items that have been discounted by the current discount can receive the discount again. The default value is `true`.
- `filterForTargetDiscountedByAnyDiscountId`—Determines whether items that have been discounted by any discount can receive the discount again. The default value is `true`.
- `filterForTargetActedAsQualifierForAnyDiscount`—Determines whether items that have acted as a qualifier for any discount can receive the current discount. The default value is `true`.
- `filterForTargetOnSale`—Indicates whether items that were priced with a sale price should be allowed to receive the current discount. The default value is `false`.
- `filterForTargetPriceLessThanOrEqualToPromotionPrice`—Determines whether items with prices that are already less than the price that would be granted by a "fixed price" promotion should receive the promotion. The default value is `true`.

Overriding Qualifier Filters

As described in the Qualifier Properties (page 160) section, the `Qualifer` class includes a number of Boolean filters that determine how the `Qualifier` treats items it is evaluating. You may want to use a standard set of filters for most situations, but have a few special cases in which you want to override the normal filtering.

In order to do this, create an additional qualifier service component and configure the filter flags as necessary for your special case. Then set the `qualifierService` property of the promotion repository item to the new qualifier component. The `qualifierService` property is an expert property that is not available through Oracle ATG Web Commerce Merchandising, but can be set manually or through the ACC. The `qualifierService` property is null by default; if set, it overrides the pricing engine's determination of which qualifier to apply.

Some examples of circumstances in which you may want to override filters are:

- You want an item that is the target of an item promotion to be eligible to act as a qualifying item for a shipping promotion; set `filterForQualifierDiscountedByAnyDiscountId` to false.
- You want an item to simultaneously be the qualifier and the target, for instance "Buy 4 shirts for $20." The promotion should have the following structure:
  - Discount type = fixed price
  - Adjuster = $2.50
  - Qualifier='For next 4 product whose category is shirt'
  - Target='up to 4 product whose category is shirt'

  In addition, set the `filterForTargetActedAsQualifierForAnyDiscount` property to `false`.

Default Qualifier Service

The `QualifierService` component is a preconfigured instance of the helper class `atg.commerce.pricing.Qualifier`. The default Oracle ATG Web Commerce discount calculators use this component to determine the objects to which they should apply their discounts.

The following table describes the properties included in `QualifierService`.

Property	Description
`pricingModelProperties`	Specifies a bean that hosts the names of all of the properties of a pricing model repository item.
`PMDLCache`	An instance of `atg.service.cache.Cache` that maps a Pricing Model to its parsed Java form. In a production environment, caching increases site performance by allowing Commerce to evaluate pricing models more quickly. A setting of -1 indicates an unlimited cache size. In a development environment, however, you may want to disable caching for pricing models so that changes you make to PMDL rules appear on your development site immediately, without requiring you to flush the cache. To disable caching for pricing models, locate the `maximumCacheEntries` property in the appropriate instance of `atg.service.cache.Cache` and set the property to 0.

You can view and modify the `QualifierService` component in the ATG Control Center. The component is located in `/atg/commerce/pricing/QualifierService`.

Evaluating Qualifiers Example

This section describes how qualifiers are evaluated using a "buy 1 get 1 free" example. This example uses a promotion of type "Item Discount" where the fixed price is $0.00 and the PMDL rule is:

> *Condition*:
> When order contains at least 1 (product named Shirt)
> *Apply discount to*:
> up to 1 (product named Hat)

The `discount-structure` in the PMDL would contain the following information:

```
<calculator-type="standard" discount-type="fixed-price"
adjuster="0">
```

If the list price of the shirt is $10.00 and the list price of the hat of $5.00, then first, the `ItemPricingEngine` iterates through each of the pre-calculators:

- The `ItemListPriceCalculator` looks up the list price of each item in the order. Based on the list prices, the shirt is priced at $10.00 and the hat will be priced at $5.00. This will update the `ItemPriceInfo` for both the `CommerceItem` objects in the order.
- The `ItemSalePriceCalculator` looks up the sale price of each item in the order. Because neither item is on sale, this has no effect on the price.

Next, The `ItemPricingEngine` iterates through each of the promotions. In this example, the "Buy 1 shirt, get 1 hat free" promotion is the only promotion.

The pricing engine calls `findQualifyingItems()` and uses the returned `calculator-type` to look up the `ItemDisCountCalculator`.

The `findQualifyingItems` method performs the following functions (as well as some standard parameter verification and error checking):

- `wrapCommerceItems` - creates `FilteredCommerceItems` for each item
- `filterItemsForQualifier` - runs through the list of the qualifier filters. In this example, none of the qualifiers apply.
- `evaluateQualifier` - Three arguments are passed to `evaluateQualifier`.

  `PricingContext` contains the following information, with specifics for this example in parentheses:

  - `pricingModel`—Current promotion (Buy 1 shirt, get 1 hat free)
  - `profile`—Current profile object
  - `locale`—User's locale
  - `order`—The order being priced
  - `orderPriceInfo`—The order's price (not yet calculated for this order)
  - `shippingGroup`—The `ShippingGroup` being priced (null, as what is being priced is not a shipping group)
  - `shippingPriceInfo`—Costs associated with the shipping group being priced (null, as what is being priced is not a shipping group)

  `FilterQualfiedItems` represents a List of `FilteredCommerceItem` objects (Shirt and Hat) and their corresponding `ItemPriceInfo` objects ($10 and $5).

  `ExtraParametersMap` is not needed in this example, but can be used to pass additional parameters as needed.

  In this example, the `evaluateQualifier` method returns `Boolean.TRUE`. The reason this is a boolean value is because this is a *when* rule. There are three choices for the condition: *always*, *when*, and *for*. In the case of *always* and *when*, `evaluateQualifier` returns a boolean value.

  Note: If the rule were a *for* rule, then `evaluateQualifier` would return the list of items that triggered the promotion. If the promotion was "For next 1 (product named shirt)" then this method would return a List containing one `MatchingObject` that wrapped the "shirt" `CommerceItem` and had quantity 1.

- Because the promotion is valid, we must determine which items will receive the discount. The first step is to filter the items for the target. (`filterItemsForTarget`).

  Note: Because this promotion involves a *when* rule, we can immediately evaluate the target. If this was a *for* rule, we would first determine the range within which `DetailedItemPriceInfo`(s) acted as the qualifier.

  Call `evaluateTarget`. Assuming none of the target filters applied, the list of arguments here will be the same as the list passed to `evaluteQualifier`. In this example, one item should be discounted so this method will return a List with one item in it. The item will be a `MatchingObject` with the following property values:

  - `matchingObject` property is the hat `CommerceItem`
  - `quantity` property is 1

- `discounts` property contains a List of `DiscountStructure` objects, representing the `discount-structure` element from the PMDL.

Next, the pricing engine gets the `calculator-type` from the `DiscountStructure` object, looks up the `calculatorTypeCalculators` map to get the calculator component, puts the `QualifiedItem` objects into the extra parameters map, and then calls the calculator.

Finally, the calculator's `findQualifyingItems` method pulls the `QualifiedItem` objects out of the map. The `getAdjuster` and `getDiscountType` methods get the `DiscountStructure` out of the map. The calculator now knows which items should receive the discount, so it calls `priceQualifyingItems`. This method goes through each detail of each item that qualifies (there is only one in our case) and updates the price.

QualifiedItem Class

The `atg.commerce.pricing.QualifiedItem` class holds information about a `CommerceItem` that qualifies for a discount. Each `CommerceItem` is paired with a `DiscountStructure` object, which contains discount information to apply.

Each `QualifiedItem` contains a single `DiscountStructure` to be applied to a quantity of a given `CommerceItem`. If multiple discounts within a single promotion apply to a single `CommerceItem`, multiple `QualifiedItem` objects are returned. For example, a single item might qualify for two discounts within a single promotion, one for $10 off and a second for an additional $5 off. In that case, two `QualfiedItem` objects are returned. (Note that this is a highly unusual case; the default behavior is to prevent a single item from receiving multiple discounts.)

Unlike a `QualifiedItem`, a `MatchingObject` can contain multiple discount structures. The usual purpose of PMDL containing multiple discount structures is to discount one set of items by one discount and a different set of items by the second discount. Note that the Oracle ATG Web Commerce Merchandising promotions user interface does not support assigning multiple discount structures in a single promotion; however, you can create custom templates that include this ability.

`QualifiedItems` are returned from the `Qualifier.findQualifyingItems` method. The `QualifiedItem` class contains the following properties:

- `item`—The `CommerceItem` that qualified for a discount.
- `qualifyingDetailsMap`—A map keyed by the `DetailedItemPriceInfo` objects contained in the `ItemPriceInfo` object (which, in turn, is contained in the `CommerceItem`). The value for each `DetailedItemPriceInfo` object is a List of Range objects, which specifies the objects matched.

  For example, if a commerce item represents two T-shirts and has a single `DetailedItemPriceInfo` and a promotion applies to one of those T-shirts, the map would have one entry. The key would be the `DetailedItemPriceInfo` and the value is a `List` with a single `Range` entry. The `Range` in this case is [0,0]. If both T-shirts qualified, the `Range` would be [0,1].

FilteredCommerceItem

The `FilteredCommerceItem` object represents a `CommerceItem` that is currently participating in a rules evaluation in the Qualifier. This object holds a reference to the object it is wrapping. `Item` points to the wrapped `CommerceItem`. All `CommerceItem` methods except `quantity` call through to the wrapped item.

`atg.commerce.pricing.FilteredCommerceItem` adds four properties that are used to determine if and how the wrapped `CommerceItem` should participate in pricing:

- `quantityAsQualifierDetails`—A map of `DetailedItemPriceInfo` objects to `Range` objects, which state the units of the details that have acted as a qualifier for something.
- `detailsRangesValidForTarget`—A map of `DetailedItemPriceInfo` objects to the number of details that are available for discounting based on the exclusion rules defined by various properties in the `Qualifier`.
- `priceQuote`—The value of the current `ItemPriceInfo`.
- `usePriceQuote`—A Boolean set to `true` after the PMDL has been evaluated. When true, calls to `getPriceInfo` return the `priceQuote`, ensuring that the evaluation process uses the most current price information.

Extending the Qualifier Class

This section describes the following ways of extending the `Qualifier` class:

Adding New Criteria to the Filter Methods

The existing `Qualifier` `doFilters()` method evaluates items in the qualifier and target rules. The method uses the `Qualifier` service's filtering methods to disqualify items before comparing them to a promotion's `PMDLRule`.

The filtering process prevents problems with pricing rules. For example, in the promotion "buy one item, get one item free," most retailers exclude the item that acts as a qualifier from receiving the discount. This prevents a customer from buying just one item and getting that one item free; the customer must put two items in the cart in order to get the discount.

The following example shows how the `filterItemsForQualifier` and `filterOrdersForTarget` methods work. This example uses the rule "for next 1 item that is blue, discount up to 1 item that is green."

The `filterItemsForQualifier` method is invoked first. By default, this method uses the following criteria to remove items from the environment. This prevents the items from helping to satisfy the constraints specified in the "qualifier" portion of the input PMDL rule:

- If an item's price is zero or negative, the item is removed.
- If an item is on sale, it is removed.
- If an item has already acted as a qualifier, the item is removed.
- If an item has already received the discount currently being evaluated, the item is removed.

After items are filtered out of the environment, the `evaluateQualifier` method is invoked. `evaluateQualifier` selects one blue item that acts as a qualifier from the environment. (For more information on these methods, refer to the next section, Replacing the Way a PMDL Rule Is Evaluated (page 166).)

If the qualifier is satisfied, the system evaluates the offer, if there is one, to determine which objects among all those available should receive the discount that is enabled by the one blue item. Oracle ATG Web Commerce currently performs this selection for `CommerceItems` only.

Before the target element is evaluated, the `filterItemsForTarget` method must be invoked. The `filterItemsForTarget` method uses the following criteria to remove items from the environment against which the target is compared:

- If the item price is zero, negative, or less than the promotion price
- If the item is on sale
- If an item already acted as a qualifier
- If an item already received the discount that is currently being evaluated, or any other discount

If the item meets any of those criteria, the item is removed.

After the `filterItemsForTarget` method is invoked, the `evaluateTarget` method is called. This method returns a set of items that can receive the discount because they satisfy the target element of the rule.

You can change the criteria by which items are filtered out of the environment before a rule is evaluated. For example, your filter could allow items with a price of zero to act as qualifiers for a rule. You could rewrite the `filterItemsForQualifier` method to remove that restriction, or write your own filter component using the `atg.commerce.pricing.PromotionQualifyingFilter` interface.

Replacing the Way a PMDL Rule Is Evaluated

The `Qualifier` class contains two protected methods, `evaluateQualifier` and `evaluateTarget`, which are responsible for evaluating different elements of a PMDL rule.

The `evaluateQualifier` method determines if the input environment (the pricing context) satisfies the constraints of the qualifier rule. The input environment is represented by the parameters passed into the method. If the constraints are satisfied, `evaluateQualifier` determines which objects acted as qualifiers in satisfying the rule. For more information, see the Evaluating Qualifiers Example (page 162) section

For example, in the rule "Buy 1 shirt, get 1 hat free," the "one shirt" element of the rule is evaluated through `evaluateQualifier`. If the promotion is a "For next" promotion, the method returns the shirt item that satisfies the constraint; if a "When" promotion, a Boolean value of `true` is returned.

For example, consider the PMDL rule "When there is at least 1 blue item, discount 1 green item." If a blue item is included in the input environment, the qualifier returns a `java.lang.Boolean` indicating that the environment matched the rule. If the rule had been written "For the next blue item, discount one green item," the blue item is returned.

The `evaluateTarget` method is invoked in every promotion situation, in order to retrieve the discount structure from the offer.

Replacing the Way the Qualifier Determines the Result Set

You can replace the way that a helper method determines its result set by extending the `Qualifier` to override any of the existing `findQualifying`*`itemType`* methods.

For example, you might not want to use a PMDL rule to determine the objects to discount. You could use a repository query to select the items that a given promotion should discount. The list of items to discount could be stored in the promotion itself. The `findQualifyingItems` method could access the promotion, read the items to discount, and return them.

Alternatively, you can create your own custom `Qualifier` service with its own `findQualifying`*`itemtype`* implementation. You can then configure a pricing engine to use the new `Qualifier` service, or configure individual promotions to use it through the `qualifierService` property.

Accessing FilteredCommerceItems

The `findQualifyingItems`, `findQualifyingOrder`, and `findQualifyingShipping` methods process `FilteredCommerceItems`. If you extend the `Qualifier` class, and the extension needs access to a wrapped item, it can get it by calling the `getWrappedItem` method of the `FilteredCommerceItem`.

12 Understanding Promotions

Promotions are a way of encouraging a user to make a purchase by highlighting and offering discounts on products and services. Examples of promotion types include the following:

- Specific amount off a particular product
- Percentage amount off a whole order
- Specific amount or percentage off a product based on an attribute
- Free shipping for a specific product
- Buy one, get one free
- Free gift with purchase

This chapter discusses the back-end functionality that supports promotions as a part of the Oracle ATG Web Commerce pricing system.

You can create promotions using the ATG Control Center (ACC) or Oracle ATG Web Commerce Merchandising, and deliver them to users through a variety of methods, including scenarios. See the *Creating and Maintaining Promotions* chapter of the *ATG Commerce Guide to Setting Up a Store* for information on creating and delivering promotions using the ACC. See the *ATG Merchandising Guide for Business Users* for information on creating promotions using Merchandising.

Note that promotions you create in the ACC can be edited in Merchandising; however, not all promotion types can be created using the ACC. Also, promotions created using Merchandising templates cannot be edited in the ACC, due to differences in the Pricing Model Description Language (PMDL) generated.

This chapter contains the following sections:

Promotion Repository Item Properties

The properties of a promotion `RepositoryItem` are documented in the table below. These properties work together to form a description of what discount to give, under which conditions it should be given, and the mechanism by which the discount is applied.

In addition to the standard property types (strings, numbers, dates, etc.) promotions also include a property type unique to promotions items, `pmdlRule`, which stores the PMDL rule for the promotion as a string. This property has a custom property editor associated with it in the Promotions section of the ATG Control Center; see the *ATG Commerce Guide to Setting Up a Store* for information.

The following table describes the pricing model properties available in the Item, Shipping, Tax, and Order descriptors when you create a commerce item. These descriptors are defined in `/atg/commerce/pricing/pricingModels.xml`, which is located in `<ATG10dir>/DCS/config/config.`.

Property Name	Type	Values	Flags
`adjuster`	double	any double	none
(display name: Discount Price or Percentage, depending on the discount type)	Number by which the item is discounted. Works in conjunction with `discountType` to specify the discount to be applied. For example, an adjuster of 15 and a `discountType` of `percentOff` produce a discount of "15 percent off." **Note:** This property is no longer used, and is maintained only for backward compatibility.		
`allowMultiple` (display name: Give to a customer more than once)	Boolean	`true` or `false`	none
	Determines whether the promotion can be given to a customer only once. If set to `false`, the system grants the promotion only once. If set to `true`, the system adds a copy of the promotion to the customer's profile every time the customer is granted the promotion. **Note:** This property is ignored if the `global` property is set to `true`.		
`beginUsable`	timestamp	any date	none
(display name: Usage start date)	The date that the promotion becomes effective. Used when the `relativeExpiration` property is set to `false`. Together with the `endUsable` date, defines the period within which the promotion is valid, including global promotions. If both values are null, the promotion is always valid.		
`creationDate`	date	any date	read-only
(display name: Creation date)	The date when the promotion was created.		
`description`	string	any string	none

Property Name	Type	Values	Flags
(display name: Description)	Provides a short description of the item.		

`discountType`
 	
(display name: Discount type)
	string	`percentOff`
		`amountOff`
		`fixedPrice`
			read-only

	The type of discount this promotion gives. **Note:** This property is no longer used, and is maintained only for backward compatibility.		
`displayName`	string	any string	none
(display name: Name)	Specifies the name visible in the user interface.		
`enabled` (display name: Enabled)	Boolean	`true` or `false`	none
	Specify `true` to enable the promotion. If enabled, the promotion takes effect according to the specified usage period. If disabled, the promotion **never** takes effect regardless of the usage period. **Note**: As a general rule, you should never delete promotions and instead disable them by setting the `enabled` property to false. This approach eliminates the possibility of deleting a promotion that has been used in an order, which produces errors.		
`endDate`	date	any date	none
(display name: Distribute through)	The date that the promotion stops being delivered to people, if the collection filtering feature is implemented to use this property.		
`endUsable`	timestamp	any date	none
(display name: Usage end date)	The date that the promotion stops being effective. Used when the `relativeExpiration` property is set to `false`. Together with the `beginUsable` date, defines the period within which the promotion is valid. If both values are null, the promotion is always valid.		
`giveToAnonymousProfiles`	Boolean	`true` or `false`	none
(display name: Give to anonymous customers)	If both this property and the `global` property are `false`, then only registered visitors can receive the promotion. If this property is `true`, and the `global` property is `false`, then anonymous visitors and registered visitors can receive the promotion. In both cases, the visitors must meet any other conditions specified by the promotion in order to receive it. **Note:** This property is ignored if the `global` property is set to `true`.		
`global`	Boolean	`true` or `false`	none

Property Name	Type	Values	Flags
(display name: Automatically apply to all orders)	Setting the `global` property to `true` indicates that this promotion will be applied an unlimited number of times, to all visitors (including anonymous visitors), for use on an unlimited number of orders, during the specified usage period, regardless of the values set for the following properties: -- `allowMultiple` -- `startDate` -- `endDate` -- `giveToAnonymousProfiles` -- `relativeExpiration` -- `timeUntilExpire` -- `uses` Setting the `global` property to `false` indicates that the system delivers and applies the promotion according to all of the values specified for the promotion.		
`media`	map	map of object to object	none
(display name: Media)	The media, such as icons, associated with this discount.		
`oneUsePerOrder`	Boolean	`true` or `false`	none
(display name: One use per order)	A property used for shipping promotions only. It determines whether a shipping promotion can discount a single order multiple times. If set to `true`, then only one shipping group in the order can use the promotion. If set to `false`, then it is possible for each shipping group in the order to be discounted by the promotion.		
`pmdlRule`	string	any valid PMDL rule	none
(display name: Discount rule)	This is the PMDL rule describing the conditions under which this promotion should take effect. The rule is created in the ATG Control Center using the interface designed for promotion creation. For more information, see the *ATG Commerce Guide to Setting Up a Store*.		
`pricingCalculatorService`	enumerated	a calculator	none
(display name: Pricing Calculator)	The calculator that computes and applies this promotion's discount. **Note:** This property is no longer used, and is maintained only for backward compatibility.		
`priority`	integer	any integer	none

Property Name	Type	Values	Flags
(display name: Order of application)	The priority of the promotion. Promotions are applied in order of priority, with low priority numbers applied first. Engines sort the promotions by the value of this property. Note that this property functions within the context of a particular promotion type. For example, you can specify how a given Item Discount promotion is applied compared to other Item Discount promotions, but not the order in which Item Discounts are applied compared to Shipping Discounts.		
`relativeExpiration` (display name: Usage Period)	Boolean	`true` or `false`	none
	Determines whether the usage period for the promotion is fixed or relative. If `false`, the promotion's usage period is determined by the dates set in the `beginUsable` and `endUsable` properties. If `true`, the promotion's usage period is set according to the date it is received by the user (that is, when the promotion is added to the user's `activePromotions` profile property). The start date and time is set when the user receives the promotion. The end date and time is set by the start date/time and the value of the `timeUntilExpire` property.		
`startDate`	date	any date	none
(display name: Distribute starting)	The date that the promotion begins to be able to be delivered to people, if the collection filtering feature is implemented to use this property.		
`timeUntilExpire` (display name: Redeemable for)	int	any int	none
	Determines the usage period in minutes for the promotion. Used when the `relativeExpiration` property is set to `true`. The promotion becomes active as soon as the user receives the promotion; that is, the promotion is added to the list of promotions in the user's `activePromotions` profile property. The expiration date and time is then determined by the number of minutes in `timeUntilExpire` to the current time. **Note:** This property is ignored when the `global` property is set to `true`.		

Property Name	Type	Values	Flags
`type` (hidden)	enumerated	Oracle ATG Web Commerce 10 and later versions: Item Discount Shipping Discount Order Discount Versions prior to Oracle ATG Web Commerce 10: Item Discount– Percent Off Item Discount– Amount Off Item Discount– Fixed Price Item Discount– Multiplier Shipping Discount- Percent Off Shipping Discount- Amount Off Shipping Discount - Fixed Price Order Discount - Percent Off Order Discount - Amount Off Order Discount - Fixed Price	read-only
	The type of discount this promotion gives. This is set during item creation.		
`uses`	int	any positive integer, zero	none
(display name: Number of uses allowed per customer)	The number of **orders** for a given customer to which the promotion can be applied. If this number hits zero, the promotion can no longer be applied. **Note:** A promotion can sometimes discount a single order multiple times. This is still considered one "use." For shipping promotions only, you can prevent the promotion from discounting a single order multiple times by setting the `oneUsePerOrder` property to `true`. **Note:** This property is ignored when the `global` property is set to `true`.		
`version`	long	any long	hidden
(display name: Version)	Used by the SQL Repository to protect against data corruption caused by two different threads attempting to modify the same item the same time.		
`parentFolder`	`promotionFolder`	any `promotionFolder` item	none
(display name: Parent Folder)	The parent folder of the promotion.		
`sites`	set	set of siteConfiguration items	none

Property Name	Type	Values	Flags
(display name: Sites)	The `siteConfiguration` item or items with which the promotion is associated. Pricing engines should only evaluate promotions that include the current site context, or where both the `sites` and `siteGroups` properties are null.		
`siteGroups`	set	set of `siteGroup` items	none
(display name: Site Groups)	The `siteGroup` or groups with which the promotion is associated. Pricing engines should only evaluate promotions whose `siteGroups` properties include the current site context or where `sites` and `siteGroups` are null.		
`template`	string	any string representing a template name	none
(display name: Template)	The `template` with which the promotion is associated. Promotions templates are used in Oracle ATG Web Commerce Merchandising; see the *ATG Merchandising Guide for Business Users*.		
`templateValues`	map	Map of template placeholder values	none
(hidden)	Placeholder values are used to build the PMDL from a template; see Adding New Promotions Templates (page 185).		
`uiAccessLevel`	int	0 or 1	none
(hidden)	The value in this field identifies whether or not the promotion is read-only (0) or writeable (1).		
`pmdlVersion`	int	1 or 2	none
(hidden)	The value in this field identifies whether the PMDL is pre-Oracle ATG Web Commerce 10 (version 1), or Oracle ATG Web Commerce 10 or later (version 2).		

One important part of promotion creation is making sure that the promotion can only be used in the way you intend it to be used. Incorrectly worded or configured promotions could allow customers to receive greater benefits from the promotion than you intended. The following list describes issues to keep in mind when you are creating promotions:

- Check the `startDate`, `endDate`, `beginUsable`, and `endUsable` dates of all promotions to prevent a promotion from taking effect before you are ready for it.

  Note: When setting dates for a promotion, be aware that a number of factors can cause the promotion to be unusable for a number of minutes after it is set to be active and to be usable after it is set to expire, depending on the schedule set for the pricing engine (for global promotions) or the `PricingModelHolder` (for both global and nonglobal promotions).

- Because it can be resource intensive to collect a user's list of promotions, or pricing models, the session-scoped `UserPricingModels` component (class `atg.commerce.pricing.PricingModelHolder`) stores them in a session cache. When a session starts for a user, `UserPricingModels` queries each pricing engine

for the customer's promotions. `PricingModelHolder` is schedulable, and can also be configured to periodically update the promotions cache.

These promotions include both the promotions in the user's `activePromotions` profile property and the list of global promotions. To retrieve the list of global promotions, the pricing engine uses its `globalPromotionsQuery` property to query the Promotions repository for all promotions where the `global` property is set to `true`, and it does so every *x* minutes as defined by the schedule specified in its `updateSchedule` property. Once collected, the pricing models in `UserPricingModels` are then used for all pricing operations during the user's session.

Consequently, if a user's session is created **after** you have added a new global promotion but **before** the next scheduled job to update the pricing engine's list of global promotions, the user will not receive the new global promotion. You can prevent this situation by manually calling the `pricingEngine.loadGlobalPromotions` method when you add the new global promotion. Additionally, if a user's session was created **before** you added a new promotion (either targeted or global), that user will never receive the new promotion. This is because the user's list of pricing models has already been collected and stored in the user's `UserPricingModels`. To prevent this situation, manually call the `pricingModelHolder.initializePromotions` method when you add any new promotion (targeted or global). The `initializePromotions` method generates a new list of pricing models to store in the user's `UserPricingModels`. (Note that the `pricingModelHolder.initializePromotions` method should be called after the `pricingEngine.loadGlobalPromotions` method in order to collect all new global promotions.)

For more information on the `PricingModelHolder` class, see the PricingModelHolder (page 125) section. For more information on the pricing engine APIs, see the *Commerce Pricing Engines* (page 127) chapter.

- When creating promotions, evaluate the wording of the discount rule carefully to make sure that only the intended products receive the promotion. Be as specific as possible when creating rules. For example, if a particular brand is on sale, specify the brand in the rule rather than relying on an attribute of the brand that you might think is unique.
- Use caution when creating "infinite use" promotions, which a customer can use an infinite number of times during a specified time period. Be certain the `beginUsable` and `endUsable` dates are set correctly.

PromotionFolder Repository Items

Promotions can be stored in a promotions folder. The `promotionFolder` repository item in the Promotions repository includes the following information, which is standard for many repository items:

- `id`
- `name`
- `parentFolder`

PromotionStatus Repository Items

When a promotion is associated with a customer's profile, it is wrapped inside a `PromotionStatus RepositoryItem`. This repository item tracks the number of times a customer can use an individual promotion.

A `PromotionStatus RepositoryItem` is a repository item with an `ItemDescriptor` that describes the status of the promotion.

A customer's profile has an `activePromotions` property that contains a list of `PromotionStatus RepositoryItems`. Each `PromotionStatus` item contains the following information:

- A reference to the underlying promotion that was created in the ACC Promotions editor.
- The number of times that a customer can use the promotion.
- The date on which the user was granted the promotion.

During the pricing process, pricing engines inspect the customer's profile to see which promotions should be considered when generating prices.

Understanding PMDL Discount Rules

This section describes the XML used for constructing discount rules that represent promotions in Oracle ATG Web Commerce.

PMDL XML Structure

The PMDL that describes Oracle ATG Web Commerce promotions discount rules is relatively simple. The PMDL DTD is located in the DCS module at `/atg/dtds/pmdl/pmdl1.0.dtd`. The DTD defines a number of iterators (such as `next` and `every`), quantifiers (used in "when" conditions), operators (such as `and` and `not`), and comparators (such as `contains` and `less-than`) you can use in creating your PMDL rules.

Pricing-Model Element

The root tag for the PMDL.

Offer Element

Every `pricing-model` element requires one `offer` element. The `offer` includes one or more `discount-structure` elements, which contain detailed information about the discount and its target. The `offer` is evaluated by the `evaluateTarget` method.

You can include more than one `discount-structure` element in an offer; this allows you to wrap multiple discounts in a single promotion (note that this is not supported in Oracle ATG Web Commerce Merchandising, but you can build a custom template with this functionality).

If you have multiple discount structures within a single item promotion, you can specify the `filter-collection-name` attribute of the `offer`; this ensures that once a given item has been marked to receive a discount, it cannot receive a discount from any other discount-structures. If `filter-collection-name` is not set, filtering does not take place, and a given commerce item can be the target for more than one discount. The `filter-collection-name` should match the `iterator` element's `collection-name` attribute, which is normally set to `items`. Filtering is not required for single discount structures, or for non-item-based promotions.

Qualifier Element

Every `pricing-model` element requires one `qualifier` element. The element is the root tag for the condition part of the promotion.

Target Element

Your discount structure should not include a `target` element if the promotion is for orders or shipping; only item discounts include `target` as part of the `discount-structure`. The `target` specifies the rule for selecting the items to be discounted.

Discount-Structure Element

The `discount-structure` element has the following attributes:

- `calculator-type`—A calculator service configured in the pricing engine's `calculatorTypeCalculators` map.
- `discount-type`—The calculators use this value to determine how to calculate an adjustment. Examples include `percentOff` and `fixedPrice`.
- `adjuster`—This optional attribute specifies the price adjustment to make for this discount.

A `discount-structure` element can also include `discount-detail` elements.

Discount-Detail Element

This element is optional, and is used for complex discount types. For example, in a tiered discount, each band is represented by a `discount-detail` element. In this case, a `discount-detail` element would include one or more attribute elements that describe the band. For example, the XML representation of a band where buying 5 or more items gives the customer 10% off might look like the following:

```
<attribute name="band" value="5"/>
<attribute name="adjuster" value="10"/>
```

Attribute Element

The `attribute` element allows you to add generic name/value pairs to parent tags, similar to the process used to extend an Oracle ATG Web Commerce repository. During PMDL parsing, the attributes and their values are placed in an attribute `Map`.

Iterator Element

Iterators are evaluation beans that sort a collection of items, then evaluate each item against one or more sub-expressions. It returns those items that match the sub-expressions.

The `iterator` element allows you to create custom iterators. Your new `iterator` element must include a `name` attribute that is unique across the PMDL.

An `iterator` element can have the following attributes and sub-elements:

- `name` attribute—Required
- `sort-by` attribute—Required
- `order` attribute—Required
- `collection-name`—Required
- `element-name`—Required
- `element-quantity`-property—Optional

Quantifier Element

Quantifiers are evaluation beans that evaluate a collection of items against one or more sub-expressions. It returns `true` or `false`, depending on the quantity of items that match the sub-expressions.

The `quantifier` element allows you to create custom quantifiers. Your new `quantifier` element must include a `name` attribute that is unique across the PMDL.

A `quantifier` element can have the following attributes and sub-elements:

- `name` attribute—Required
- `number` attribute—Optional
- `collection-name`—Required
- `element-name`—Required
- `element-quantity-property`—Optional

Operator Element

Operators are evaluation beans that return `true` or `false` based on the Boolean results from their sub-expressions.

The `operator` element allows you to create custom operators. Your new `operator` element must include a `name` attribute that is unique across the PMDL.

An `operator` element can specify any number of `attribute` sub-elements and operates on at least one comparator, operator or quantifier.

Comparator Element

Comparators are evaluation beans that return `true` or `false` depending on the values of their sub-expressions.

The `comparator` element allows you to create custom comparators. Your new `comparator` element must include a `name` attribute that is unique across the PMDL. A `comparator` element can specify any number of `attribute` sub-elements and must specify at least one `value` or array name.

Comparators evaluate using one or more sub-expressions. For example:

```
<comparator name="isoneof">
  <value>item.auxiliaryData.productId</value>
  <constant>
    <data-type>java.lang.String</data-type>
   <string-value>xprod2147</string-value>
    <string-value>xprod2163</string-value>
  </constant>
</comparator>
```

Comparators can also compare two `value` elements, and custom comparators could include any number of `value` or `constant` elements.

Value and String-Value Elements

The `value` element returns the value of a property of the commerce item being evaluated. For example:

```
<value>item.auxiliaryData.productId</value>
```

To represent strings, use a `string-value` element. For example:

```
<string-value>ManufacturerA</string-value>
```

To represent a comma-separated list of strings, use multiple `string-value` elements. For example, to represent "ManufacturerA, ManufacturerB" use the following:

```
<string-value>ManufacturerA</string-value>
<string-value>ManufacturerB</string-value>
```

Constant Element

The constant element returns a constant value against which other values can be compared. For example:

```
  <constant>
    <data-type>java.lang.String</data-type>
   <string-value>xprod2147</string-value>
  </constant>
```

PMDL Example: Bulk Discount

The example that follows shows these structures in operation. Note that in the normal course of events you should not have to work directly with the PMDL unless creating new promotions templates (see Adding New Promotions Templates (page 185)). Occasionally, however, a user creates a promotion in which the PMDL is invalid; in this case, the promotion cannot be edited in Oracle ATG Web Commerce Merchandising or in the ATG Control Center, but must be manually corrected or discarded.

This rule describes a promotion in which a customer buys up to five of a particular product (xprod2104) to receive $5 off. The customer can buy six of the same item and receive $10 off.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE pricing-model SYSTEM dynamosystemresource:/atg/dtds/pmdl/pmdl_1.0.dtd">
  <pricing-model>
    <qualifier/>
    <offer>
      <discount-structure calculator-type="bulk"
        discount-type="amountOff">

        <discount-detail>
          <attribute name="band" value="1"/>
          <attribute name="adjuster" value="5"/>
        </discount-detail>
        <discount-detail>
          <attribute name="band" value="6"/>
          <attribute name="adjuster" value="10"/>
        </discount-detail>

    <target>
      <iterator name="up-to-and-including" number="1"
```

```
        sort-by="priceInfo.listPrice" sort-order="ascending">
        <collection-name>items</collection-name>
        <element-name>item</element-name>
        <element-quantity-property>quantity
        </element-quantity-property>
        <comparator name="equals">
          <value>item.auxiliaryData.productId</value>
          <constant>
            <data-type>java.lang.String</data-type>
            <string-value>xprod2104</string-value>
          </constant>
        </comparator>
      </iterator>
    </target>
  </discount-structure>
</offer>
</pricing-model>
```

Gift with Purchase Promotions

Gift with Purchase promotions contain several features unique to that promotion type. A Gift with Purchase promotion is one in which a customer order can qualify for specific items as a free gift.

The Gift with Purchase architecture relies on order and commerce item markers. See the *ATG Commerce Guide to Setting Up a Store* for information on markers.

Gift with Purchase Repository Items

The Gift with Purchase feature extends the base marker repository definitions to provide `gwpOrderMarker` and `gwpCommerceItemMarker` item types. Each `gwpOrderMarker` represents a single gift selection. Each `gwpCommerceItemMarker` tracks the quantities of that commerce item which was auto-added, selected, or targeted by a Gift with Purchase promotion.

Gift with Purchase markers only persist while the order still qualifies for the promotion. If the promotion no longer applies, associated markers are removed.

The `gwpOrderMarker` item type has the following properties:

Property	Data Type	Description
`key`	String	This value is always `atg.gwp`.
`value`	String	The `promotionId`.
`data`	String	Hash code to track which gift selection in the promotion this is. It allows the commerce item marker to link to this marker. A hash of the PMDL discount detail for a single gift selection given by the promotion.

Property	Data Type	Description
giftType	String	The type of gift, which can be sku, product, category, skuContentGroup, or productContentGroup.
giftDetail	String	String identifying the gift, usually the repository ID of the gift type.
autoRemove	boolean	Flag to indicate whether free gifts should be auto removed if the promotion no longer qualifies. This is configured by the merchandiser when the promotion is created.
quantity	long	The total quantity of SKU for this gift selection, this is the quantity from the PMDL multiplied by the number of times the offer applied (if it applied more than once).
automaticQuantity	long	The quantity of free SKU for this gift selection that has already been auto added to the order.
targetedQuantity	long	The quantity of free SKU for this gift selection that has already been targeted and made free by the calculator in the order.
selectedQuantity	long	The quantity of free SKU for this gift selection that has already been selected by the Shopper and added to the order.
removedQuantity	long	The amount of the free quantity that has been manually removed by a shopper. Keeping track of removed quantities prevents them from being automatically re-added in future pricing operations. The assumption is that the customer does not want the free item.
failedQuantity	long	Tracks any gifts that should have been automatically added but failed, for example due to a site conflict with the promotion.

The key, value and data properties are required to look up marker objects. The combined value and data properties are unique for a given gwpOrderMarker.

The automaticQuantity, targetedQuantity and selectedQuantity properties keep track of the different ways that a free quantity of SKU can be in the cart. The quantity of SKU the shopper still needs to select via a placeholder item can be determined via the following expression:

```
quantity - (automaticQuantity + targetedQuantity + selectedQuantity +
removedQuantity + failedQuantity).
```

The gwpCommerceItemMarker repository item has similar properties, as shown in the following table:

Property	Data Type	Description
key	String	The value is always atg.gwp.
value	String	The promotionId.

Property	Data Type	Description
`data`	String	The gift hash code for the promotion, a hash of the PMDL discount detail for a single gift selection given by the promotion.
`targetedQuantity`	long	The quantity of the free gift that this commerce item is making free and was targeted by pricing.
`automaticQuantity`	long	The quantity of the free gift that this commerce item is making free and was auto added by pricing.
`selectedQuantity`	long	The quantity of the free gift that this commerce item is making free and was selected by the shopper.
`remainingQuantity`	long	The total quantity of the commerce item that doesn't apply to this gift. Used to detect when a commerce item quantity is removed to ensure the gift quantities are updated correctly.

The `value` and `data` properties are used to match the `gwpCommerceItemMarker` to the related `gwpOrderMarker` that relates to this commerce item marker.

The `targetedQuantity`, `automaticQuantity` and `selectedQuantity` track how much of the total quantity of the gift is provided by the commerce item. Multiple commerce items can contribute to the total quantity. For example, a Gift with Purchase promotion could offer a free gift of two watches. The shopper selects two different watches, creating two commerce items. Each commerce item has an associated `gwpCommerceItemMarker`, each of which refers to the same `gwpOrderMarker`. Each `gwpCommerceItemMarker` has a `selectedQuantity` of 1.

Gift with Purchase Classes

The Gift with Purchase feature is supported by an info class and a manager class.

`GWPInfo` objects are used for temporary storage of gift item status for the current item pricing operation. They are processed by the `GWPManager`. Any information that needs to be persisted is stored in markers against the order. There is an instance of `GWPInfo` for each gift with purchase promotion that qualifies and free gift selection given. There is a one to one mapping between `GWPInfo` objects and the order markers that store information against the order.

The map of `GWPInfo` objects is cleared for each pricing operation. If a gift with purchase promotion applies a discount more than once or qualifies more than once, there is still only a single instance; however the `quantity` is incremented.

The `GWPDiscountCalculator` (see the Pricing Calculator Classes (page 140) section of this guide) is responsible for creating and updating a `Map` of `GWPInfo` for the current pricing operation. The Map is keyed by a composite key of the `promotionId` and `giftHashCode`. The `GWPManager` processes the `Map` of `GWPInfo` objects to update the markers and gift items in the cart.

Gift with Purchase Events

When a new commerce item is added or removed or the quantity is changed due to gift with purchase selections, the following events are used to convey the change information:

- `ItemAddedToOrder`
- `ItemRemovedFromOrder`
- `ItemQuantityChanged`

For each event, the `giftWithPurchase` Boolean flag is set to true when the event fires due to gift with purchase actions.

Extending Promotions Functionality

Extending the existing promotions functionality can involve doing any one or more of the following:

- Specifying multiple targets for a discount
- Creating new promotion types (these map directly to pricing engines, and by default the types include item, order, or shipping)
- Creating new discount calculators (by default, Oracle ATG Web Commerce includes standard and tiered calculators). Calculator types map to Nucleus components.

The PMDL stores the discount information and ties together a set of target items with the discount that should be applied. PMDL allows you to include multiple targets and multiple discount structures in the same promotion, but the latter must all be the same promotion/pricing engine type.

For example, say you want to create another tiered pricing calculator to handle a special discount, such as free ground shipping.

1. Write a new discount calculator as a Nucleus component. The new calculator should implement the `calculatorInfoProvider` interface. The interface provides a `calculatorinfo` object that describes the calculator, including the discount types that it supports and the attributes that calculator is interested in.
2. Configure the item pricing engine to add the new calculator type, using the `calculatorTypeCalculators` map in the pricing engine.
3. The new calculator and discount type appears automatically in the Advanced Template (see the *ATG Merchandising Guide for Business Users*), and you can also create your own custom templates that use this discount type.

If you add a new calculator type, you will most likely modify an existing calculator to support it, perhaps by adding a new method (see the *Commerce Pricing Calculators* (page 139) chapter).

Extending the PMDL

As well as adding discount types and calculators, you can extend the PMDL itself; for example, you may have a need for a `<xor>` element, which is not included in the PMDL by default.

For iterators, quantifiers, comparators, and operators, the process is extremely simple:

1. Create a Java class containing the logic for the new tag. For example, `myClasses.pricing.definition.XorElem.class`.

2. Use XML-combine to extend the PMDL schema to include your new tag. For example:

```
<operator name="xor">
…
</operator>
```

1. Add a mapping between your Java class and the PMDL element in the `PMDLParser.xml`. For example:

```
<bean name="xor" class="myClasses.pricing.definition.XorElem.class">
</bean>
```

If you want to add an element other than iterators, quantifiers, comparators, and operators, you must also update the DTD to support the new element.

When adding elements to the PMDL, use the `name` attribute rather than directly naming the new element. For example, if you add a custom xor operator, the following PMDL is valid:

```
<operator name="xor">
….
</operator>
```

Note that the following is *not* valid unless you have also updated the DTD:

```
<xor>
….
</xor>
```

You can use the new tag in your custom promotions templates. Note however that the new tag does not appear in the Advanced condition and offer expression editor.

Adding New Promotion Discount Types

Adding a new discount type involves the following tasks:

1. Extend or create a calculator that supports the discount type.
2. Modify the `CalculatorInfo` object to include the new discount type and calculator information.

The PMDL uses the `CalculatorInfo` to identify which calculator to use.

Adding New Promotions Templates

Oracle ATG Web Commerce Merchandising includes a number of ready-to-use promotions templates that make the job of creating new promotions easy for business users. You may find that your users frequently create promotions that are not based on one of the provided templates. In that case, you may want to create your own templates to streamline the process for them and reduce the chance for user errors.

This section describes the XML grammar used for promotions templates.

Promotion Template Basics

Promotions templates are XML files that are named with a `.pmdt` extension. You must store templates in a root location specified in the `configurationRootPath` property of the `/atg/commerce/promotion/template/registry/PromotionTemplateRegistry` component. The default setting is:

```
configurationRootPath=/atg/registry/data/promotiontemplates
```

You can set this path to any valid value and construct any additional folder structure needed below it as you add templates. Templates are referenced by both name and location, therefore the names do not have to be unique, but it is good practice to give them unique names.

Template files are created manually. You can shorten the process by creating a PMDL rule in the Advanced Condition and Offer Editor, then retrieving the generated PMDL from the `DCS_DISCOUNT_PROMO` table or from the asset detail page of the project. That PMDL can be used for the template's `pmdlrule` repository property, replacing placeholder names with the necessary values. You must still ensure that your template XML file includes the requisite user interface elements.

Note that if you do use this method to create a PMDL rule, you should be careful to delete the promotion you created, to avoiding confusing your users. You cannot delete promotions through Oracle ATG Web Commerce Merchandising the way you can some other assets, but you can create a temporary project in Merchandising, then delete the project without deploying.

What Promotion Templates Do

Templates allow your users to set promotion repository item properties through a simple interface. The `pmdlRule` property one of the most important properties, but other properties such as usage dates can also be set this way (see Promotion Repository Item Properties (page 170) for a full property list).

1. The `ui-description` section of the template lays out the visual elements of the template, which determine how it appears on the user's screen. Much of the `ui-description` relies on standard form elements such as `textInput`.

   The user interface also includes product set criteria (PSCs), which are sets of combination boxes that allow users to specify products to include in or exclude from a promotion. Modeling for these expressions is handled by a custom `PSCExpressionModel` component, which requires no configuration.

   See Creating the PMDT File (page 187) for information on `ui-description` elements.

2. The end-user provides their input to the template in Oracle ATG Web Commerce Merchandising. For example, the user might select products to include in the promotion, set begin and end dates for the promotion, and specify how many times the promotion can be given to a customer.

3. The template associates the user input with the `id` attribute of the element through which it was selected. For example, you could have a `textInput` that allows the user to specify how many items can have the promotion applied:

   ```
   <textInput id="numberOfItemsToDiscount_textInput">
   ```

   In the `textInput` example, the information referenced by the `id` is most likely a simple integer; in the case of PSCs, the `id` references a complex string of PMDL representing the user's selections, their AND or OR conditions, etc.

4. Translator elements in the template perform any transformations necessary to the user input; the translated input is assigned to a placeholder. For example, you may need to change date formatting or capitalization.

For PSCs, the translators add required tags to create well-formed PMDL, which can then be processed by the back-end systems. See Translating User Input Values in Templates (page 194).

5. The `item-properties` section of the template provides the blueprint for the promotion, and defines what property information the promotion stores.

Creating the PMDT File

This section describes the main sections that make up the PMDT file from which the template user interface is created.

Template Header

The first part of the template PMDT file is the header, which provides basic information about the template. The header includes the following attributes:

The template header attributes are:

- `item-type`—Required. Type of repository item the template represents. The default options are item-discount, order-discount, and shipping-discount.
- `author`—Optional. Name of the template author.
- `last-modified-by`—Optional. Name of the person who last modified the template. Must be manually updated if used.
- `creation-date`—Optional. Creation date of the template.
- `deprecated-by`—Optional. Name of the person who deprecated the template. Must be manually updated if used.

For example:

```
<template   item-type="Item Discount"
            repository="some/repository/path"
            author="Template Author"
            last-modified-by="Template Author"
            creation-date="28/09/2009">
```

UI Description Element

If you want your template to appear in the Oracle ATG Web Commerce Merchandising user interface, you must include a `ui-description` element as part of the template.

If the template has no `ui-description` section, it is not displayed in Merchandising's template list. If the template does have a `ui-description` section then it will appear in the template list unless the `available-in-ui` attribute is explicitly set to false.

Note: You may not want the template to appear in the UI if the template is intended for importing or exporting promotions (see Importing and Exporting Promotions (page 202)), or if you want to remove the template from the list from which users can select.

The `ui-description` tag includes the following attributes, all of which are optional:

- `display-name`—String used to label the template in the template list and in the template screen title bar.

- `resource-bundle`—Optional. The location of the resource bundle used to represent this template in the UI.
- `display-name-resource`—Optional. If the `resource-bundle` attribute is present, the `display-name-resource` is used as a key to retrieve the localized value from the indicated resource bundle. If `resource-bundle` is not present or the specified resource bundle cannot be found, the `display-name-resource` attribute value is used for display. This attribute overrides the `display-name` attribute.
- `available-in-ui`—Optional. If this attribute is not included in the XML, it defaults to `true`. If explicitly set to `false`, the template is not displayed on the template selection screen, even if it has a `ui-description` section.

UI Description Child Elements

The `ui-description` can include a number of sub-elements.

The `multi-element-translators` element is used to combine, translate, or insert values into the item properties. See Translating User Input Values in Templates (page 194).

The optional `group-info` element is used to create the template selection menu. The `group-info` element has two optional attributes:

- `groupId`—If `groupId` is specified, that template appears grouped with other templates that have the same `groupId`. If no `groupId` is specified, the templates are grouped into a default group. Templates within a group appear in ascending alphabetical order unless a `templatePriorityId` is specified.
- `templatePriorityId`—If specified for a template, templates are displayed in ascending priority order rather than ascending alphabetical (the default). Zero designates the highest priority.

Additional elements are structural, and provide layout information for the template page:

- The `screen-segment` element provides a means of grouping elements within the template. A `screen-segment` has a `name` and however many `line` elements you want to include.
- The `line` element contains user interface items you want to appear as one line to the user, for example a label and its corresponding input area.

A `line` element is the smallest structural unit of a template.

This example shows a screen segment with a single line of input:

```
<screen-segment display-name="Condition"
  <line>
    <label id="spendLabel"
           display-name="Spend:"
    <textInput id="spend_textInput"
               placeholder-name="spend_value"/>
    <label id="spend_example_label"
           display-name="ex. 100" styleName="infoText"/>
  </line>
</screen-segment>
```

The resulting user interface resembles the following:

Line Child Elements

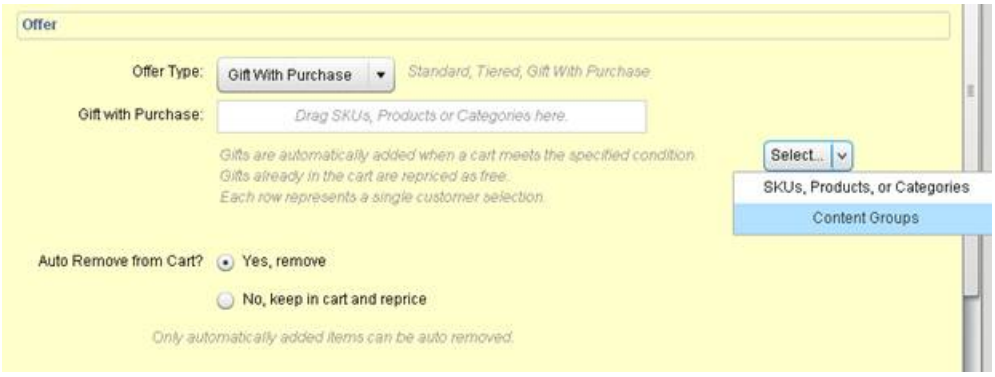
The `line` element can include any of the following sub-elements

- `textInput`
- `comboBox`
- `radioButtonGroup`
- `checkBox`
- `textArea`
- `date`
- `label`
- `spacer`
- `horizontalRule`

Most of these elements represent standard user interface components. Each can include a number of attributes such as `height`, `label`, and a resource bundle reference; see the `PMDT.DTD` for detailed information.

The following elements are unique to promotions templates, and are described in the sections that follow:

- `expression`—Used by the advanced screen to display a condition or offer sequence See the Expression Elements (page 190) section for more information.
- `grid`—Used for table layouts; see the Grid Elements (page 191) section for details.
- `includesProductSetCriteria` and `excludesProductSetCriteria`—Used to construct rules that identify which products are eligible for the promotion. See the Product Set Criteria Elements (page 193) section for more information.
- `switchableDiscount`—This element allows a user to toggle between discount amount and a discount structure grid view. See the SwitchableDiscount Element (page 190) section.
- `assetCollector`—When clicked, pops up an asset picker dialog from which assets can be selected. (See the *ATG Business Control Center Administration and Development Guide*).
- `gwpSelector`—An asset selector with restrictions specific to gift with purchase promotions. The selector supports both drag and drop and asset picker selection, and allows selection of categories, products, SKUs, or content groups as a gift. See the following screen shot for an illustration.



Each element in the user interface has an `id` attribute, which can be used to identify the user input for that element elsewhere in the template. Elements can also have a `place-holder-name` attribute. The `place-holder-name` attribute is used to insert user input into the PMDL statement the template constructs.

Expression Elements

The `expression` element allows you to support complex, freeform conditional statements such as those used in the Advanced Condition & Offer template. It does this using an independent grammar file, and appears in the user interface as an expression editor.

The `expression` element functionality is available through the Advanced Condition & Offer page, and has the following attributes:

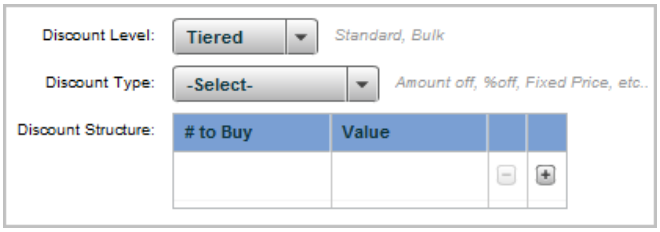
- `id`—Id to use when referencing the expression.
- `model-path`—Path to a Nucleus component that provides the information necessary to configure the expression with the correct sequence information.
- `enabled`—True by default; if false, the expression section appears grayed out and uneditable in the template.
- `required`—True by default; if false, the expression section is included but is not required by the template.

SwitchableDiscount Element

If the offer type is not Gift with Purchase, the `switchableDiscount` element allows a user to toggle between the views defined by the beneath it; for example, between the discount amount and a discount structure grid view.

The examples shown are from the Advanced Condition & Offer:

Discount Amount



Discount Structure

The following code sample illustrates the use in the template:

```
<line>
  <switchableDiscount id="switchable_discount">
    <view>
      <label id="discount_structure_label" display-name-
```

```
resource="template.common.discountStructureLabel" container-width="100" container-
h-align="right"/>
      <grid id="discountStructureData"
validator="/atg/remote/promotion/template/validators/TieredItemDiscountValidator">
        <content-source
path="/atg/remote/promotion/template/contentSource/BandedDiscountStructureContents
">
          <attribute-value-reference name="calculatorType" element-
id="calculatorType"/>
        </content-source>
      </grid>
    </view>
    <view>
      <label id="discount_amount_label" display-name-
resource="template.common.discountAmountLabel" container-width="100" container-h-
align="right"/>
      <textInput id="discountAmount" required="true" restrict="0-9\."
validator="/atg/remote/promotion/template/validators/NumberValidator"/>
      <label id="discount_amount_example_label" display-name-
resource="template.common.discountInfoLabel" styleName="infoText"/>
    </view>
  </switchableDiscount>
</line>
```

If the offer type is Gift with Purchase, the `SwitchableDiscount` element displays the UI for that offer type.

Grid Elements

The `grid` element allows you to include a `DataGrid` component as part of your template's user interface. The main use for grids is to provide users with an easy means to enter data when creating banded promotions. For example:

Users can add or subtract rows in the grid using the – and + buttons.

The `grid` element supports two approaches to defining the `DataGrid` component; these approaches are described in the sections that follow.

When the template is validated on startup, if a `grid` element has both a `content-source` and one or more `grid columns`, or if it has neither a `content-source` nor a `grid-column`, an error is logged, and the template is not included in the template list in the user interface.

Dynamically Determining Grid Structure at Runtime

The first method for creating a grid in a template dynamically generates the grid contents at runtime. The XML used for the user interface would resemble this example:

```
<line>
  <label display-name="Discount Structure:"/>
  <grid id="tieredPromo" placeholder-name="tiered_promo_value">
```

```
      <content-source path="/atg/xxx/yyy">
          <attribute name="calculator-type" value="bulk"/>
      </content-source>
  </grid>
</line>
```

If you use this approach, the `grid` element must include a `content-source` element. The `content-source` points to a Nucleus that provides the grid with the necessary data structure. The provided `/atg/remote/promotion/template/contentSource/BandedDiscountStructureContents` component is specific to discount structures; if you want the grid for some other purpose, you can write your own component.

The component should obtain a `CalculatorInfo` object for the appropriate calculator type (defined in the child `attribute` element of the `content-source`) and the promotion type (Item, Order, Shipping or Tax), both of which are passed to the component as the page is generated.

Explicitly Defining Table Structure

As an alternative to dynamic grid generation, you can also explicitly define the `DataGrid`. In this case, the XML used would resemble the following example:

```
<line>
  <label display-name="Discount Structure:"/>
  <grid id="tieredPromo"
      placeholder-name="tiered_promo_value">
    <grid-column display-name="band">
      <textInput />
    </grid-column>
    <grid-column display-name="adjuster">
      <textInput />
    </grid-column>
  </grid>
</line>
```

The `grid` element in this case explicitly lists the components needed to let the user interface render and build the grid. The child `grid-column` elements control the display and data-input characteristics of each column in the table. The `display-name` attribute of the `grid-column` contains the column header (in this example, it uses a resource bundle). The child elements of the `grid-column` specify the editor type to be presented in the user interface, which can be either `textInput` or `comboBox`.

AssetTable Elements

The `assetTable` element provides a way for merchandisers to add assets to a promotion condition or offer, either by selecting from an asset picker or using drag and drop.

To override the default set of draggable assets, you can define a content-source element, as shown in the example below. This element limits the draggable assets to products and product subtypes:

```
<assetTable id="gwpAssetTable" container-width="100" container-h-align="right">
   <content-source
path="/atg/remote/promotion/template/contentSource/AssetParentChildInfo">
   <attribute name="assetBaseTypes"
        value="atg/commerce/catalog/ProductCatalog:product/>
   </content-source>
</assetTable>
```

Product Set Criteria Elements

The `includesProductSetCriteria` and `excludesProductSetCriteria` elements can be used within a `line` element. They allow users to construct statements for determining products to include in or exclude from the promotion. Each element consists of multiple combo boxes that allow users to specify a sequence of comparative values for properties. The output is a placeholder value that can be used in a PMDL rule.

The `includesProductSetCriteria` and `excludesProductSetCriteria` elements have the following attributes:

- `id`—Identifier for the PSC element
- `required`—Whether or not the PSC is required; the default is true
- `model-path`—The model for a PSC is a Nucleus component that controls how the combination boxes are populated and displayed. For product set criteria elements, the model-path should be:

  `model-path="/atg/remote/promotion/expreditor/psc/PSCExpressionModel`

Four additional attributes control basic display options:

- `container-width`
- `container-height`
- `container-h-align`
- `container-v-align`

This example shows an `includesProductSetCriteria` element in a `ui-description`:

```
<includesProductSetCriteria id="PSC" required="true" model
-path="/atg/remote/promotion/expreditor/psc/PSCExpressionModel" />
```

The user interface representation of an expression element consists of three parts:

- A fully qualified property of an item such as a product or SKU
- A comparator such as *equals* or *is one of*
- The editor to use for the property

The resulting combination box looks like the following example:

The user can add can add criteria by clicking the Add Criteria button in the UI (button inclusion and placement is automatic), and join these criteria as AND or OR combinations.

When the user makes selections and saves the template-based promotion, the information is translated into a PMDL statement by the `PSCExpressionModel`.

Using Optional Fields

It may be necessary to mark fields as optional in your template. You can do this by including an `required="false"` attribute in any user interface element tag. When this attribute is not present, the element is marked with an asterisk as usual for template fields.

If you make a field optional and the user submits no content for that field, a blank area is substituted for the placeholder associated with the field in the associated item property (as linked by the placeholder-name), which could render the PMDL invalid. If you do make fields optional, make sure that your PMDL blueprint permits an empty string.

For example, if the user interface element for a particular placeholder value is optional and might be left empty, the placeholder should include both the start and end tags of the PMDL, or should be positioned so that an empty string does not break the PMDL.

Translating User Input Values in Templates

As part of your template design, take into account whether you need to translate the values users enter into values appropriate for the promotion repository item. For example, you may have a field that allows the user to input a Cents value between 0 and 99. The PMDL expects a dollar amount; therefore, the 25 the user enters must be converted to 0.25. This task is handled by Nucleus components called translators.

Translator components are based on classes that implement the `ElementTranslator` interface (see the *ATG Platform API Reference* for details). Translators are called after any client-side validation takes place. They take the following input:

- The element ID
- The element type
- The raw value to be translated, as entered by the user
- A list of `ElementState` objects, which can be used if the translation for the current element is affected by input to another element
- A set of known repository property names and values for the item type

The output is a value that can be directly applied to the property based on the `placeholder-name`. If further translation is required at this point, you should use a multi-element translator instead.

In addition to basic translation, you may need to combine user-entered values in order to provide a valid PMDL input. For example, it is common to combine the output of an “excludes” product set criteria with an “includes” product set criteria. Components that handle this advanced processing are called multi-element translators; they can take multiple inputs and supply multiple output values.

For example, the provided item discount templates include the `UnlimitedDiscountTranslator`. These templates have a discount section that contains a `textInput` element, in which the user can enter the number of times a customer can use the discount, and a `checkbox` element to indicate unlimited use. The `checkbox` element has the following attribute:

```
placeholder-value-checked="-1"
```

This attribute means that if the checkbox is selected, the output value is -1.

The `UnlimitedDiscountTranslator` takes the user-entered value or the unlimited option and provides a single `placeholder` value as output, which either contains the number of items to discount or a -1 to indicate unlimited. The `placeholder` value is then inserted in the location of its corresponding `placeholder-name`.

For example, the following snippet shows PMDL before translation:

```
<iterator name="up-to-and-including"
number="${no_of_items_to_discount}" sort-by="priceInfo.listPrice"
sort-order="${sort_order}">
```

After translation, if the user enters a value, the PMDL looks like the following::

```
<iterator name="up-to-and-including" number="5" sort
-by="priceInfo.listPrice" sort-order="${sort_order}">
```

If the user checks unlimited, the PMDL looks like the following:

```
<iterator name="up-to-and-including" number="-1" sort-
by="priceInfo.listPrice" sort-order="${sort_order}">
```

Multi-element translators are defined in the `ui-description` section of the template within a `multi-element-translators` element. For example, the following code shows how the `multiElementTranslator` for the unlimited discount functionality is implemented:

```
<multi-element-translators>
        <multi-element-translator id="unlimited_discount_translator"
translator-path="/atg/remote/promotion/template/translators
/UnlimitedDiscountTranslator">
        <placeholder-info placeholder-name="no_of_items_to_discount"
translator-output-name="itemNumber"/>

              <element-info element-id="numberOfItemsToDiscount_textInput"
translator-input-name="itemNumber"/>

              <element-info element-id="unlimited_checkbox"
 translator-input-name="unlimited"/>
 </multi-element-translator>
</multi-element-translators>
```

The `UnlimitedDiscountTranslator` accepts two inputs, which are mapped to the `translator-input-name` element's `itemNumber` and to `unlimited` in the `element-info` child element. The `placeholder-info` element maps the `placeholder-name` to the translator's `output-name` key.

Input and output names for each translator are defined in its properties file. The properties file for the example `UnlimitedDiscountTranslator` above is:

```
$class=atg.remote.promotion.template.translators.UnlimitedDiscount
Translator
$scope=session
#
# Translator inputs
```

```
#
itemNumberInputName=itemNumber
unlimitedInputName=unlimited
#
# Translator outputs
#
itemNumberOutputName=itemNumber
```

By default, Oracle ATG Web Commerce includes the following translators:

- `AssetCollectorTranslator` - Translates selected asset IDs into PMDL
- `RawPMDLTranslator` – Retrieves the PMDL from the promotion item

Commerce also includes the following multi-element translators:

- `DiscountStructureTranslator` – translates the discount structure PMDL
- `QualifierTranslator` – translates the qualifier PMDL
- `TargetTranslator` – translates the target PMDL
- `DiscountTranslator` – Converts separate `discount type` and `adjuster` user interface inputs into separate `discount-type` and `adjuster` output values, allowing for a discount type of free.
- `UnlimitedDiscountTranslator` – Sets up the iterator number attribute for PMDL iterators based on amount to discount or unlimited if a checkbox on the user interface is checked.

You can create your own translators based on the provided framework; see the *ATG Platform API Reference*. Translators should implement either the `ElementTranslator.java` or the `MultiElementTranslator.java` interface.

Working with Repository Item Properties in Templates

The sections that follow describe how to expose and use repository item properties as part of your promotions template.

Note: If creating a template using a custom discount type, make sure the PMDL in that template sets the `discount-type` attribute correctly.

The `item-properties` section of the PMDT file complements the `ui-description` and allows you to set promotion properties using the `place-holder-value` attributes of the template elements. The `property` statements are based around a separate grammar from that used in the UI portion of the template.

The `item-properties` section of the template for a `beginUsable` and `endUsable` looks like the following sample. Note the $ marking where place-holder-value information is inserted:

```
<property name="beginUsable">
  ${beginUsableValue}
</property>

<property name="endUsable">
  ${endUsableValue}
</property>
```

Displaying Static Values in Templates

The default behavior for generating a display name for a property is:

1. If the resource bundle includes an entry of "psc.property.<full property name>" use that value.

   psc.property.item.auxiliaryData.productRef.ancestorCategoryIds=Categories

2. If not, get the supported bean from which the property came and check the resource bundle for an entry of psc.bean.<supported bean name>. This forms the first part of the display name. If no entry is found, use the display name of the DynamicBeanDescriptor for that bean.

   psc.bean.product=Product

3. For each sub-property part of the property name, check the resource bundle for an entry of psc.property.<sub property part name> and append that value to the display name. If no entry is found, use the display name of the DynamicPropertyDescriptor instead.

   psc.property.amount=Price

Displaying Dynamic Properties in Templates

The content-source element dynamically obtains information for display in the template. One common use is populating combo boxes (see the Product Set Criteria Elements (page 193) section), but you can use it to populate any user interface element, provided that the content source's Nucleus component provides the correct output.

For example, consider this template XML:

```
<comboBox id="discountType">
  <content-source path="/atg/remote/promotion/template/contentSource/
CustomDiscountListContent">
    <attribute-value-reference name="calculatorType"
 element-id="calculatorType"/>
  </content-source>
</comboBox>
```

The path attribute is required, and identifies the location of the Nucleus component that returns the dynamic information.

You can use the optional child attribute element to specify parameters to pass to the Nucleus component. Parameter values are either set explicitly or obtained from the user input to another field in the template user interface. The component returns a ContentState object that contains the information used to populate the field.

For example, the attribute-value-reference element in the code shown above tells the content source to watch a second user interface element, that has the element-id="calculatorType". The user's currently selected value for the calculatorType is assigned the name calculatorType and sent to the CustomDiscountListContent component. The component obtains a valid discount list for that calculatorType.

If the user changes the selected calculatorType, the new value is sent to the component to obtain an updated discount list for display.

Content source Nucleus components should implement the atg.remote.promotion.template.contentSource.java interface. See the *ATG Platform API Reference* for information on this interface.

Using an Asset Picker in a Promotions Template

If you have included repository item properties in your template, you need to provide a map to the Asset Picker editor needed to manipulate that property. The asset picker is a Flex `RepositoryItemSetEditor` component used in many Oracle ATG Web Commerce applications (see the *ATG Business Control Center Administration and Development Guide* for general information on asset pickers).

The asset picker requires three pieces of information:

- A Nucleus path to the repository in which the assets are found
- The assets' repository item type
- Whether not multiple assets can be selected

To provide this information, configure the `assetPickerPropertyMap` and the `assetPickerRepositoryMap` components in the `PSCExpressionContext` component.

The `assetPickerRepositoryMap` property provides a map of properties to their repositories. The `assetPickerPropertyMap` property provides a map of properties to their repository item types. If a property appears in this map, the promotions user interface uses the asset picker editor for that property; otherwise, it defaults to a simple text editor.

Whether or not the asset picker allows multiple selections depends on the operator the user selects; for example, "is" allows only single selection, but "is one of" permits multiple selections.

If a property is included in the `assetPickerPropertyMap`, the promotions expression editor displays an asset picker for that property on the right hand side of the product set criteria expression, using the value of the map as the asset type. If an entry for the property is also in the `assetPickerRepositoryMap`, the value of this map is used as the repository path for the asset picker.

If a property is in the `assetPickerPropertyMap` but not in the `assetPickerRepositoryMap`, then the asset picker defaults to using the `RepositoryPropertyDescriptor` for that property to determine the repository to use.

This sample shows configuration for the two properties:

```
assetPickerPropertyMap+=\
  item.auxiliaryData.catalogRef.sites=siteConfiguration
```

```
assetPickerRepositoryMap+=\
  item.auxiliaryData.catalogRef.sites=/atg/multisite/SiteRepository
```

The resulting user interface would display the `Sites` property of the `sku` repository item in an asset picker. The picker would use the `siteConfiguration` asset type and `SiteRepository`.

Automatic Property Filtering in Templates

Oracle ATG Web Commerce repository items used in promotions tend to have many properties that are not useful as promotion criteria. Therefore, the list of properties is filtered by default.

If you add custom properties to your Oracle ATG Web Commerce repository, note that category properties do not appear in the PSC; only product and SKU properties are exposed.

The `/atg/remote/promotion/expreditor/psc/PSCExpressionContext` component includes the `excludedProperties` property. The value for this property can end in * to exclude all sub-properties of a property. Note that a * does not exclude the referenced property itself, but only its sub-properties. The actual property must be excluded separately. For example:

```
excludedProperties+=\
  item.auxiliaryData.productRef.template.*
```

The example above excludes all sub-properties of the `template` property, but does not exclude the `template` property itself. To exclude the `template` property, use the following configuration:

```
excludedProperties+=\
  item.auxiliaryData.productRef.template\
  item.auxiliaryData.productRef.template.*
```

The `PSCPropertyChoiceExpression` class on which the `PSCExpressionContext` component is based extends the `PricingPropertyChoiceExpression` class; therefore, it inherits a standard filtering mechanism which by default filters out properties marked as `expert = true` or `queryable = false` in the repository definition. You can override this behavior using the `mandatoryProperties` property of `PSCExpressionContext`.

Note that the `mandatoryProperties` property does not support wildcards; all mandatory properties must be specified explicitly. This example shows `mandatoryProperties` in use:

```
mandatoryProperties+=\
  item.auxiliaryData.productRef.ancestorCategoryIds
```

This configuration results in the `ancestorCategoryIds` property being included, even though this property is marked as expert in the repository.

Using Promotion Upsell in Templates

Along with other common promotion elements, you can include promotion upsell opportunities in your promotion template. To do so, create a `screen-segment` within the `ui-description` to contain the upsell information. The `screen-segment` should contain one or more `line` elements to provide the closeness qualifier information.

The `screen-segment` should also include the `display-once=true` attribute. In Oracle ATG Web Commerce Merchandising, users can create promotion upsells directly from a promotion only during the initial promotion creation (additional options are available to users for creating promotion upsells; see the *ATG Merchandising Guide for Business Users*).

For example, consider a promotion "Spend over $100, get free shipping". The closeness qualifier in this case is "spend over $X". The user interface for this would be similar to the following:

```
<screen-segment display-name-resource="closenessQualifierTitle" display
-once="true">
   <line>
      <label id="spend_label"
             display-name="Amount to spend:"/>
```

```
      <textInput id="spend_textInput" placeholder-name="spend_value"/>
   </line>
</screen-segment>
```

The matching `item-properties` section in template must contain a `property` named `closenessQualifiers`, which contains the PMDL information required to build the closeness qualifier. The property would be similar to the following:

```
<item-properties>
  <property name="closenessQualifiers">

  <item-properties>
    <property name="pmdlRule">
      <![CDATA[
  <pricing-model>
    <qualifier>
      <comparator name="greater-than-or-equals">
        <value>order.priceInfo.amount</value>
        <constant>
          <data-type>java.lang.Double</data-type>
          <string-value> ${spend_value} </string-value>
        </constant>
      </comparator>
    </qualifier>
  </pricing-model>
  ]]>
</property>
  </item-properties>
</property>
</item-properties>
```

You cannot nest multiple `item-properties` elements within a `property` element.

Validating Promotions

When a user views a promotion, the `PromotionTemplateManager` determines whether the template is valid for that promotion.

If the template fails validation but can still be parsed, Oracle ATG Web Commerce Merchandising may provide an opportunity to open the promotion using the Advanced Condition and Offer page (see the *ATG Merchandising Guide for Business Users*). If the Advanced Condition and Offer is unable to display the template, Merchandising displays the raw PMDL.

When a user creates a new promotion from a template, the promotion is validated through a combination of client-side and server-side activities. In addition to that validation, you can restrict the text users can enter in a field. For `textInput` and `textArea` elements, you can use the following attributes to limit user input:

- `maxChars` – limits the number of characters that can be typed in the field
- `restrict` – indicates the set of characters that a user can enter. If the restrict attribute is a string of characters, the user can only enter those characters into the field. You can specify ranges using a hyphen. Use a double backslash to include characters such as ^, - and \. For example, the following code allows the user to enter a dash (`-`) in the `textInput`:

  ```
  restrict="\\-"
  ```

The `restrict` attribute maps to the Flex `restrict` object, and supports standard Flex functionality.

You can also associate a user interface element with a Nucleus validator component, using the `validator` attribute in the `ui-description` element. The Nucleus component performs the validation when the user clicks OK on the completed Condition and Offer popup.

Each user interface component type has its own validator, but you can use a single validator for multiple UI element types. The validator is passed the type of element the information came from, the information that was supplied by the user and the id of the field it came from. The field id uniquely identifies the user interface element to be validated.

If validation and any subsequent translation are successful, a `PropertyState` object containing the promotion properties is returned. If some validation failed, then an error string is passed back to the user interface inside the `PropertyState` object, to be associated with the user interface element failing validation. The promotion itself is not created or saved at this time.

By default, Oracle ATG Web Commerce includes the following validator components in the `atg.remote.promotion.template.validators` package:

- `BandedDiscountValidator`
- `Number Validator`
- `RegexValidator`

The `RegexValidator` includes several preconfigured variations that can check for the following patterns:

- DecimalValidator—Match any positive floating point number, includes zero

  ```
  regexPattern=(^[0-9]*(\.[0-9]+)?$)
  ```

- DollarValidator—Match any dollar amount to two decimal places, includes zero

  ```
  regexPattern=(^[0-9]*(\.[0-9][0-9])?$)
  ```

- NonZeroDecimalValidator—Match any positive floating point number, excludes zero

  ```
  regexPattern=(^([0-9]*[.][0-9]*[1-9]+[0-9]*)$)|(^([0-9]*[1-9]+
  [0-9]*[.][0-9]+)$)|(^([0-9]*[1-9]+[0-9]*)$)
  ```

You can extend the provided validation framework by adding your own validators. Custom validators must implement the `atg.remote.promotion.template.contentSource.java` interface; see the *ATG Platform API Reference* for details.

Localizing Promotions Templates

You can use resource bundles to localize promotions templates. The resource bundles must be placed in the following directory:

```
<ATG10dir>/ATG10/home/locallib
```

To specify a bundle for key/value retrieval, use the `resource-bundle` attribute of the `ui-description` element:

```
<ui-description available-in-ui="false" resource-
bundle="atg.remote.promotion.template.Resources" display-name-
resource="template.item.advancedItemDiscount.title" >
```

If a user views a template that refers to a resource bundle that has not already been loaded, the client attempts to locate the new bundle before displaying the template. If it cannot do so, the client displays an error message, and the template includes the resource keys instead of the localized strings.

Editing Existing Promotion Templates

You can use XML-combine to override and extend an existing promotions template. If you do this, be careful not to break any promotions that have already been created based on the original template (if the promotion can no longer be displayed due to changes in the template, the user may be able to convert the template to use an Advanced Condition and Offer. Bear in mind that promotion assets cannot be deleted in Oracle ATG Web Commerce Merchandising.

Another option is to make changes in a copy of the original template, and then use the new version for future promotions.

Similarly, do not delete any templates unless you are certain that no promotions based on that template exist.

Importing and Exporting Promotions

If Oracle ATG Web Commerce is not your usual tool for creating and maintaining promotions, you will need to import promotions you create in your external system into Commerce for pricing purposes. Commerce includes an API that allows you to build custom code for this purpose.

If you are using promotions templates, importing promotions is as simple as specifying which template to use and the placeholder values. If you are not using templates, you must first build the PMDL for your promotions.

You can import promotions on either a production instance or an asset management instance. If the latter, the import automatically takes the form of a Content Administration project. This topic is addressed in the Configuring the PromotionImportWorkflowAutomator Component (page 209) section.

If you are using Commerce to create and maintain promotions, but need to access them in a third-party system, you can export your promotions using a similar mechanism. In this case, you are responsible for understanding the third-party system's requirements.

Architecture Overview

The promotions import/export API relies mainly on the following two components:

- `/atg/commerce/promotion/PromotionImportExport`—The main component used for importing and exporting promotions
- `/atg/epub/PublishingWorkflowAutomator`—Used automatically when you import promotions into a versioned repository; you should never need to call this component yourself

Supporting functionality is provided by the following additional components:

- `/atg/commerce/promotion/PromotionImportExportTools`
- `/atg/commerce/claimable/ClaimableTools`
- `/atg/commerce/promotion/PromotionTools`
- `/atg/commerce/promotion/template/PromotionTemplateManager`

In addition to these components, there are a number of classes used for importing and exporting promotions.

`PromotionImportExportInfo` is a data class used to transfer information between Oracle ATG Web Commerce and your external promotion management system (see Mapping Promotion Properties (page 206)). You can use either PMDL version 1 (for promotions created before Oracle ATG Web Commerce 10, which do not use templates) or 2 (for template-based promotions created with Oracle ATG Web Commerce 10 or later).

See the *ATG Platform API Reference* for detailed information on the other classes used for promotion import/export.

Performing a Promotions Import or Export

The `PromotionImportExport` component contains the main methods used for importing and exporting promotions. Both processes start and end with `startImportExportSession()` and `endImportExportSession()` methods.

Warning: The `atg.commerce.promotion.PromotionImportExport` class on which this component is based allows you to run multiple concurrent import/export sessions. However, it provides no safeguards against making multiple concurrent updates to a given repository item.

The sections that follow describe the methods in this component at a high level. See the *ATG Platform API Reference* for additional details.

startImportExportSession()

To import promotions, first call `startImportExportSession()`. This method sets up the import session, and must be called before doing the import itself.

The method performs the following tasks:

1. If the session ID is not specified, the method generates a session ID randomly and creates a new `PromotionImportExportSession` object.
2. The method retrieves the promotions repository, using the `PromotionTools` component.
3. If the promotions repository is versioned, the method calls the `PublishingWorkflowAutomator.startWorkflowSession()` method to set up the Content Administration project workflow. The `PublishingWorkflowAutomator` performs the following tasks:
   - Creates a new `PublishingWorkflowSession` object
   - Creates a project name by appending the session ID to the `PublishingWorkflowAutomator` component's `projectNameStub` property, separated by a dash. The workflow is configured in the `PublishingWorkflowAutomator` component; see Configuring the PromotionImportWorkflowAutomator Component (page 209).

- Performs the necessary interactions with Content Administration security
- Creates the process and workspace for the project

4. The method creates a new transaction in preparation for the import-export session.
5. The method checks the component's `integrators` property to establish whether or not any integrators have been configured. (See Using the PromotionImportExportIntegrator Interface.) For each configured integrator, call its `preImportExportSession` method, passing the session object as a parameter.

The method returns a `PromotionImportExportSession` object.

importPromotion()

This method enables you to create or update an individual promotion and its associated items (such as closeness qualifiers, promotion folders, coupons, and coupon folders), depending on the action specified.

Always make sure that you have a `PromotionImportExportSession` object before using `importPromotion()`.

The method performs the following tasks:

1. Creates a new `PromotionImportExportStatus` object.
2. Checks the component's `integrators` property to establish whether or not any integrators have been configured. For each configured integrator, call its `preImportPromotion` method, passing the session object and the promotion object, as parameters.

The `action` property on the `PromotionImportExportInfo` object determines whether the repository item is created, updated, or deleted. Each item associated with the promotion also specifies its own action, but the available actions depend on the parent promotion's action, as shown by this table:

Promotion Action	Supported Closeness Qualifier Actions	Supported Coupon Actions
ADD	UPDATE DELETE	DELETE
UPDATE	(none)	(none)

When adding promotions, property values from the template manager take precedence over the equivalent values specified in the `PromotionImportExportInfo` object; for updates, the opposite holds.

The method returns a `PromotionImportExportSessionStatus` object.

exportPromotionsById()

This method enables you to export promotions by specifying a list of promotion item IDs (for an alternative export method, see the exportPromotionsByRQLQuery (page 205) section.

This method first calls the `PromotionManager.getPromotionsById()` method using the specified list of promotion IDs (if the list is null, all promotions are returned). The `getPromotionsById()` method returns a list

of promotion repository items. The `exportPromotionsById()` method then processes each of the returned promotion repository items as follows:

1. Create a `PromotionImportExportInfo` object.
2. Populate the object's `promotionPropertyValues`, `templateId` and `templateValues` properties from the promotion repository item.
3. Retrieve the promotion's folder (if not null).
4. For each closeness qualifier in the `closenessQualifiers` property of the promotions repository item, execute the following tasks:
   - Call the `PromotionImportExportTools.getClosenessQualifier()` method to get the `closenessQualifier` repository item.
   - Create a new `ClosenessQualifierImportExportInfo` object.
   - Populate its `closenessQualifierPropertyValues` map from the repository item properties.
   - Add the `ClosenessQualifierImportExportInfo` object to the `closenessQualifiers` list property in the `PromotionImportExportInfo` object.
5. Call the `ClaimableTools.getCouponsForPromotion()` method to retrieve the coupon details associated with the promotion. That method returns a list of coupon repository items. For each coupon item in the list, execute the following tasks:
   - Create a new `CouponImportExportInfo` object and populate its `couponPropertyValues` map from the repository item properties.
   - If the coupon repository item's `parentFolder` property is not null, then call the `ClaimableTools.getCouponFolderPath()` method. That method returns the full path for the coupon folder, which is used to set the `couponFolderPath` property in the `CouponImportExportInfo` object.
   - Add the `CouponImportExportInfo` object to the coupons list property in the `PromotionImportExportInfo` object.
6. Add the `PromotionImportExportInfo` object to the list to be returned to the caller.
7. Return the list of `PromotionImportExportInfo` objects.

exportPromotionsByRQLQuery

This method enables the user to export promotions by specifying a list of promotion via an RQL query.

This method calls the `PromotionImportExportTools.getPromotionsByRQLQuery()` method to query the promotions item descriptor. The method returns a list of promotion repository items. Once the list of repository items has been returned the remaining processing is identical to that of the `exportPromotionsById` method.

endImportExportSession

The `endImportExportSession()` method ends your import session. The method performs the following tasks:

1. If the session ID is not specified, throws an exception.
2. Checks the component's `integrators` property to establish whether or not any integrators have been configured. (See Using the PromotionImportExportIntegrator Interface (page 209).) For each configured integrator, call its `postImportExportSession` method, passing the session object as a parameter. In case of

exceptions, the `PublishingWorlflowAutomator`'s `endWorkflowSession` method must be called to clean up the session.

3. Checks if the current transaction is marked for roll back; if it is, then the roll back is executed, otherwise, the transaction is committed.

4. If running against a versioned repository and the `sessionStatus` is set to `SESSION_ERROR`, the workflow session is rolled back by calling the `abandonWorkflowSession` method on the `PublishingWorkflowAutomator` component.

   If running against a versioned repository and the `sessionStatus` is set to `SESSION_OK`, then the workflow session is committed by calling the `stopWorkflowSession` method on the `PublishingWorkflowAutomator` component.

After finishing the import, the workflow advances to the next stage.

Mapping Promotion Properties

The `atg.commerce.promotion.PromotionImportExportInfo` data class provides a means by which you can map properties between Oracle ATG Web Commerce promotions and your external system.

Property	Required	Type	Description

`action`	Yes	int	The action is required for imports but is null when returning export information. The valid values are:
			
			`ACTION_ADD`
			`ACTION_UPDATE`
			
			Note that there is no `DELETE` action available for promotions. This means that you cannot accidentally delete promotions that are being used by merchandisers.

`promotionPropertyValues`	Yes	Map<String,String>	A map of promotion repository item property names and their corresponding values. If you are not using templates, specify the PMDL rule for the promotion in this map. If you are using templates, you can use the `templateValues` map (see below in this table) to build the PMDL instead. **Note:** The `templateId` and `templateValues` properties are separated from other promotion properties, and do not need to be included in the map (see below in this table).

Property	Required	Type	Description
`promotionFolderPath`	No	String	Promotions can be organized using a logical folder structure. This property specifies the full folder path. For example: "/Summer/Shoes" The Promotions Import Export API converts this path into individual promotion folders, creates those which do not already exist, and links them together into a tree structure. Note that no two children of the same parent folder can have the same name, but folders can have the same name if they are child folders of different parent folders.
`templateId`	No	String	If the promotion is based on a template, the path and filename of the template to use.
`templateValues`	No	Map<String,String>	A map of template placeholders and their corresponding values. See any of the existing promotions templates for an understanding of this mapping.
`closenessQualifiers`	No	List<ClosenessQualifierImportExportInfo>	A list of `ClosenessQualifierImportExportInfo` objects.
`coupons`	No	List<CouponImportExportInfo>	A list of `CouponImportExportInfo` objects.

A similar class (`atg.commerce.promotion.ClosenessQualifierImportExportInfo`) exists for closeness qualifier information.

Property	Required	Type	Description

action	Yes	int	The action is required for imports but is null when returning export information. The valid values are:
			
			`ACTION_ADD`
			`ACTION_UPDATE`
			`ACTION_DELETE`

Property	Required	Type	Description
closenessQualifierPropertyValues	No	Map<String,String>	A map of closeness qualifier repository item property names and their corresponding values.
closenessQualifierTemplateValues	No	Map<String,String>	A map of template placeholders and their corresponding values, specifically for the closeness qualifier section of the template.

A similar class (`atg.commerce.promotion.CouponImportExportInfo`) exists for coupon information.

Property	Required	Type	Description

action	Yes	int	The action is required for imports but is null when returning export information. The valid values are:
			
			`ACTION_ADD`
			`ACTION_UPDATE`
			`ACTION_DELETE`

couponPropertyValues	Yes	Map<String,String>	A map of coupon repository item property names and their corresponding values.
couponFolderPath	No	String	Coupons can be organized using a logical folder structure. This property specifies the full folder path. For example: "/Summer/Shoes" The Promotions Import Export API converts this path into individual coupon folders, creates those which don't already exist, and links them together into a tree structure. Note no two children of the same parent folder can have the same name, but folders can have the same name if they are child folders of different parent folders.

Using the PromotionImportExportIntegrator Interface

You can customize the import/export process by adding your own components and registering them with the `PromotionImportExport` component. In order to be used automatically by the existing import/export classes, your components must implement the `atg.commerce.promotion.PromotionImportExportIntegrator` interface.

The interface includes four methods:

- `preImportExportSession()`
- `postImportExportSession()`
- `preImportPromotion()`
- `postImportPromotion()`

Extending these methods allows you to insert your custom code at different points in the import process performed by the `PromotionImportExport` component.

After creating the component, configure the integrators property of the `PromotionImportExport` component to add your new component:

```
integrators=/Nucleus_path/to/custom/component
```

Your components are inserted into the import process.

Configuring Import/Export Batching

The import/export API supports transactional batching. The import API logically groups all items associated with an individual promotion, such as upsell qualifiers, folders, and coupons. This means that if an error occurs, you can roll back all items associated with that promotion.

To change the batch size, in the `/atg/commerce/promotion/PromotionImportExport` component, change the following property:

```
batchSize=1
```

The default is 1, so that each promotion (and its associated items, if any) is created in a separate transaction. This simplifies error recovery, but may impact performance. Use the default setting unless performance is unacceptable, in which case you may want to increase the batch size.

Configuring the PromotionImportWorkflowAutomator Component

If you plan to use promotions import/export against a versioned repository, configure the `/atg/commerce/promotions/PromotionImportWorkflowAutomator` component. The import/export API calls this component automatically when it detects that it is running against versioned repositories. Any repository actions carried out within the context of the import threads are associated with the ATG Content Administration project (after a call is made to `startWorkflowSession`).

The `PromotionImportWorkflowAutomator` component provides the information that is used by the ATG Content Administration project for your import or export. This component has the following property

- `username`—Appended to the `personaPrefix` to give the persona object ID
- `workflowName`—Workflow name used when creating the process for the import
- `taskOutcomeId`—Determines the task in the workflow to be advanced to after the import
- `projectNameStub`—Stub name for the project; the full ATG Content Administration project name for the import is created by appending the `sessionId` to this value

For example:

```
userName=publishing
workflowName=/Common/commonWorkflow.wdl
taskOutcomeId=4.1.1
projectNameStub=Automated Import
```

Performance Issues Related to Promotion Delivery

In general, the more promotions a customer has in their profile, the longer it takes to generate a price for that customer. A customer can have hundreds of promotions without it significantly affecting the time it takes to price an item. However, performance is noticeably affected when a customer profile contains thousands of promotions. As a general rule, assign customers as few promotions as possible to accomplish the business goals for the site.

Do not rely on promotions to do the bulk of price generation for a site. Use properties of the SKU (for example, the `salePrice` property) to provide varied pricing. In general, if a promotion does not refer to information that might change from one request to another, there may be a more efficient way of implementing that promotion.

For example, referring to a customer's profile is an efficient way to structure a promotion. The profile will probably be different for every price being generated throughout the site, since many different customers will be using the site.

It is not always efficient to refer to the date, or to other conditions that do not change on a request basis, as part of the promotion evaluation process. For example, consider a situation in which you put all black shoes on sale one week. Rather than creating a promotion that puts black shoes on sale and giving it to every customer, you could set the sale price of the black shoes in either the SKU repository or the underlying database. This way, the promotion is still applied, but no undue load is placed on the promotion evaluation engine.

13 Using Price Lists

Price Lists allow you to target a specific set of prices to a specific group of customers. For example, price lists can be used to implement business to business pricing where each customer can have its own unique pricing for products based on contracts, RFQ and pre-negotiated prices. Price lists are managed through a single interface in the ACC, which includes a list of product IDs, SKU IDs, and configurable SKU IDs. Pricing can be inherited based on products and/or SKUs. For example, if a price is defined as $9.99 for product X, all SKUs that are in product X will be given a price of $9.99 unless the price is explicitly overwritten.

This chapter contains information on the following price list topics:

Overview of Setting Up Price Lists

You can create multiple lists. Each list has the following properties:

- Name
- Base price list
- Creation date
- Last modified date
- Start date

- End date
- Locale

The list will display the following information about a product and a SKU:

- Product ID
- SKU ID
- Description
- Pricing scheme
- List price
- Complex price

Note: Either the List Price or the Complex Price is required, not both.

The following steps describe how to price items using price lists.

1. Assign a price list to a user.

   The price list that is used to price an order is stored in the `priceList` property (of type `priceList`) in the user's profile.

   Oracle ATG Web Commerce users can also store this price list in a contract used by the customer's organization.

2. Price an item with a price list.

   There are price list-specific versions of each of the precalculators used by the `ItemPricingEngine`. `ItemPriceListCalculator` is the precalculator for list pricing. `ConfigurableItemPriceListCalculator` is the precalculator for configurable item pricing.

3. View a price through JSP code.

   The `PriceDroplet` servlet bean is used for looking up the price of an item. For more information on the `PriceDroplet` servlet bean, see the *PriceDroplet* section in this chapter in the *ATG Commerce Guide to Setting Up a Store*.

For information on setting up price lists using the ACC, see the *Managing Price Lists* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Caching Price Lists

`PriceCache` allows you to cache the prices in price lists. Price lists can contain a large number of prices. If you do not want the `PriceCache` to hold all the prices in the prices list, adjust the `PriceCache` settings in your `liveconfig` directory. The `PriceCache` settings are located in `liveconfig/atg/commerce/pricing/priceLists/PriceCache.properties`.

Using Price Lists in Combination with SKU-Based Pricing

Oracle ATG Web Commerce supports three pricing model options: SKU-based pricing alone, price lists alone, and a combination of both. In the combination case, your pricing system is configured to use price lists, but if no

price is found in the lists you have specified, it falls back to the catalog price. Consider using combined price lists and SKU-based pricing if most of your customers pay the same prices for most of your products, with only a few variations.

To use this pricing method, create the price lists as normal; however, your lists need only include the specific products for which you want to offer multiple prices. Then configure the following two properties of the `atg.commerce.pricing.priceLists.ItemPriceCalculator` component:

```
noPriceIsError=false
noPriceCalculator=path_to_an_ItemPricingCalculator
```

Which `ItemPriceCalculator` component to configure depends on the desired behavior. For example, if you want to allow missing non-sale prices to use SKU pricing, configure the `ItemPriceListCalculator` as shown:

```
noPriceIsError=false
noPriceCalculator=/atg/commerce/pricing/calculators/ItemListPriceCalculator
```

Note that the component being configured is the *PriceList*Calculator, and the path points to the *ListPrice*Calculator. See the Price List Calculators (page 215) section of this chapter for further information on these components.

Description of Volume Pricing

Price lists can be used to implement many pricing models. Two popular models are bulk pricing and tiered pricing.

Bulk pricing calculates the price of a product based on the minimum quantity that is ordered. For example, you could:

- purchase up to 10 steel beams for $50 each
- purchase 11 to 20 steel beams for $45 each
- purchase 21 or more steel beams for $40 each

In this bulk pricing example, if you bought 23 steel beams, the total cost of the order would be $920. Each of the 23 beams would cost $40.

Tiered pricing calculates the price of a product using fixed quantity or weight at different pricing levels. For example, you could:

- purchase up to 10 steel beams for $50 each
- after purchasing 10 beams for $50 each, purchase beams 11 through 20 for $45 each.
- after purchasing 10 beams for $50 each and purchasing beams 11 through 20 for $45 each, purchase any more than 20 beams for $40 each

In this tiered pricing example, 23 steal beams would cost $1070:

- 10 beams (beams 1-10) for $50= $500
- 10 beams (beams 11-20) for $45= $450
- 3 beams (beams 21-23) for $40= $120

Setting up Price List Functionality

Oracle ATG Web Commerce users do not have a price list functionality available by default. To add price list functionality, configure the `ItemPricingEngine` to use the appropriate precalculator.

There are price list-specific versions of each of the precalculators used by the `ItemPricingEngine`. `ItemPriceListCalculator` is the precalculator for list pricing. `ConfigurableItemPriceListCalculator` is the precalculator for configurable item pricing.

The following code example shows how the `/atg/commerce/pricing/ItemPricingEngine.properties` file should change to use the `priceList` calculators

```
preCalculators=\
        calculators/ItemPriceListCalculator,\
        calculators/ConfigurableItemPriceListCalculator
```

Note: The configurable item calculator is optional. It only needs to be used if your sites support configurable commerce items.

When an item is priced, the pricing calculators will use the price lists defined here to determine what price to use.

PriceListManager

The `PriceListManager` class maintains the price lists. A price may be retrieved from the `PriceListManager` from a given price list by product, by SKU, or by a product/SKU pair.

The most important method in `PriceListManager` is `getPrice`. This is used during pricing of an order to get the correct price for a given product/SKU pair.

PriceListManager can be used to assign a default price list (`DefaultPriceListId`) and a default sale price list(`DefaultSalePriceListId`) in the event that one cannot be found for a customer, using the `defaultPriceListId`. Using `DefaultSalePriceListID`, the property name of a default price list can be added as an input parameter, determining which default list should be displayed. For example:

```
public RepositoryItem getDefaultPriceList(String pPriceListName)
```

Assigning a Price List to a User

Oracle ATG Web Commerce uses similar mechanisms for assigning catalogs and price lists to customer profiles. It adds `CatalogProfilePropertySetter` and `PriceListProfilePropertySetter` components to the `profilePropertySetters` property of the `/atg/dynamo/servlet/dafpipeline/ProfilePropertyServlet` component in the DAF servlet pipeline:

```
profilePropertySetters+=/atg/userprofiling/CatalogProfilePropertySetter,\
```

	/atg/userprofiling/PriceListProfilePropertySetter

To set the profile's priceList and salePriceList properties, the PriceListProfilePropertySetter component calls the /atg/commerce/pricing/priceLists/PriceListManager component's determinePriceList method, which calls the /atg/commerce/util/ContextValueRetriever component. If this component's useProfile property is false (the default), the following logic is applied:

- If there is a current site (the application is running in a multisite environment), use the value of the defaultListPriceList and defaultSalePriceList properties of the siteConfiguration item for the current site. For more information, refer to Assigning Price Lists and Catalogs in a Multisite Configuration (page 70).
- Otherwise, use the DefaultPriceListId and DefaultSalePriceListId values set in the PriceListManager component.

For details on the ContextValueRetriever, including information on when you should override the useProfile property for price lists, see ContextValueRetriever Class (page 49).

Price List Calculators

An ItemPriceCalculator class maintains all the functionality common to all the pricing schemes. The ItemPriceCalculator has the following important properties:

- PriceListManager: holds the reference to PriceListManager
- PricingSchemeNames: holds the key/Value pair for each allowed pricing scheme and its corresponding Calculator.

  For example:

  listPrice corresponds to the PriceListsListCalculator
  bulkPrice corresponds to the PriceListsBulkCalculator
  tieredPrice corresponds to the PriceListsTieredCalculator

Also, see the Using Price Lists in Combination with SKU-Based Pricing (page 212) section for properties related to that capability.

The public API exposed by this class includes:

- getPricingScheme: returns the pricing scheme for the CommerceItem
- priceItem: examines the allowed PricingSchemeNames HashMap. When a match is found, it will call the corresponding Calculator's priceItem method. Otherwise, an exception is thrown to indicate that the pricing scheme is not found.

Three sub-calculators correspond to the three different pricing schemes. The three different schemes are calculating the list price of an item, calculating the price of an item using bulk pricing, and calculating the price of an item using tiered pricing. For more information on bulk and tiered pricing, see the *Using Price Lists* (page 211) section.

All of these calculators implement the ItemSchemePriceCalculator, which only has a priceItem method.

- `ItemListPriceCalculator`: Calls the `getPrice` method from the `PriceListManager` to retrieve the list price of the `CommerceItem`. It then multiplies the price by the quantity returned by the `getQuantity` method of `CommerceItem` to get the total price. The `ItemPriceInfo` will contain one `DetailedItemPriceInfo` for each `ShippingGroupCommerceItemRelationship` in the `CommerceItem`. This is because of the `Range` property in both `ShipItemRels` and `DetailedItemPriceInfos`.

- `ItemBulkPriceCalculator`: Calls the `getPrice` method from the `PriceListManager` to retrieve the complex price for the `CommerceItem`. It will check each price level of that complex price based on the quantity of the `CommerceItem` to decide the correct unit price for the item. The `ItemPriceInfo` will contain one `DetailedItemPriceInfo` for each `ShippingGroupCommerceItemRelationship`.

- `ItemTierPriceCalculator`: Calls the `getPrice` method from the `PriceListManager` to retrieve the complex price for the `CommerceItem`. It will check each price level of that complex price to decide which unit price is used for each tier. The `ItemPriceInfo` will might several `DetailedItemPriceInfos` to reflect different unit prices for each tier.

Using ItemPriceInfo with Price Lists

One `ItemPriceInfo` class fits three different pricing schemes. Each calculator uses a different description for the `PricingAdjustment` added to the `ItemPriceInfo`.

The `priceList` property of `ItemPriceInfo` is set to the `priceList` that was actually used to calculate it. This is nullable since other calculators other than those mentioned here will not set this. The `ItemPriceCalculator` is responsible for setting this value.

Implementing Sale Prices using Price Lists

By default, there is no sale pricing configured when using price lists. In the standard pricing model (where the price is stored directly in the SKU in the product catalog) there is a `listPrice` property and a `salePrice` property. The SKU also has a boolean `onSale` property that indicates that the given SKU should be priced using the sale price.

In the price list model, a price repository item has a `listPrice` (or a `complexPrice`) but no `salePrice`. This section describes how to implement sale pricing with price lists.

The quickest way to implement sale pricing using price lists is to create a sale price list.

In this situation, you could store all the list prices for a specific user in one price list and all the sale prices for a specific user in another price list. This set up provides flexibility. For example, you could have different sale prices for two different users, even if they have the same price list normally. It also allows us to inherit sale prices while overriding the list prices (or vice versa).

Follow these steps to implement sale pricing using price lists.

1. Creating the sale price list

   Create a sale price list the same way you create other price lists. Structurally there is no difference between a sale price list and any other price list.

2. Assign the sale price list to a user.

Since we want the flexibility of keeping sale prices completely separate from list prices, user's will need to have two price lists assigned to them. There will need to be an additional property in the user's profile to store the `salePriceList`.

Note: Oracle ATG Web Commerce users can create an additional property for the user and the contract. Create this new property by copying the `priceList` property and changing the name.

3. Pricing an item with a price list

   If you want to price an item without price lists, the following steps occur: There are two precalculators in the `ItemPricingEngine`. The item is first priced with the list price. The item is then priced with the sale price. The `ItemPriceInfo` stores both pieces of information, allowing users to calculate the discount that the user received. Price lists use a similar approach:

   The first calculator in the list by default is: `/atg/commerce/pricing/calculators/ItemPriceListCalculator`

   This is an instance of `atg.commerce.pricing.priceLists.ItemPriceCalculator`. It configures the name of the profile property that stores the price list as well as the map that configures which calculator to use for each `pricingScheme`. For sale pricing, create a new instance of this calculator called `ItemSalePriceCalculator`.

```
# The ItemSalePriceCalculator which prices an item on sale
#
$class=atg.commerce.pricing.priceLists.ItemPriceCalculator
loggingIdentifier=ItemSalePriceCalculator
profilePriceListPropertyName=salePriceList
useDefaultPriceList=false
noPriceIsError=false
pricingSchemePropertyName=pricingScheme
priceListManager=/atg/commerce/pricing/priceLists/PriceListManager
pricingSchemeNames=\
listPrice=/atg/commerce/pricing/calculators/SalePriceListsListCalculator,\
bulkPrice=/atg/commerce/pricing/calculators/SalePriceListsBulkCalculator,\
tieredPrice=/atg/commerce/pricing/calculators/
SalePriceListsTieredCalculator
```

   The following list describes the important properties of `ItemSalePriceCalculator`:

   - `profilePriceListPropertyName=salePriceList`

     This property forces the calculator to use the sale price list for pricing.

   - `useDefaultPriceList=false`

     When using list pricing, you can assign a default price list. This is usually not needed when using sale pricing so this is set to false.

   - `noPriceIsError=false`

     When calculating a list price, it is an error if there is no price defined (since we wouldn't know how much to charge). It is most probably not an error if there is no sale price, this would only mean the item is not on sale. If this is false, then no error is thrown, and no change is made to the price.

   - `pricingSchemeNames=\`
     `listPrice=/atg/commerce/pricing/calculators/SalePriceListsListCalculator,\`

```
bulkPrice=/atg/commerce/pricing/calculators/SalePriceListsBulkCalculator,\
tieredPrice=/atg/commerce/pricing/calculators/SalePriceListsTieredCalculator
```

These calculators are provided by default with Oracle ATG Web Commerce because sale price calculators manipulate the `ItemPriceInfo` in a different way than list price calculators. Sale price calculators add different `PricingAdjustments` and update `salePrice` instead of `listPrice`.

You also need to price the configurable items. A `ConfigurableItemPriceListSaleCalculator` provided out of the box. The `ItemPricingEngine` defines the following precalculators:

```
preCalculators=\
calculators/ItemPriceListCalculator,\
calculators/ItemPriceListSaleCalculator,\
calculators/ConfigurableItemPriceListCalculator,\
calculators/ConfigurableItemPriceListSaleCalculator
```

4. View a price through a JSP using JSP code.

   Using the `PriceDroplet` servlet bean retrieves the list price. Another instance of this servlet bean retrieves the sale price. For example:

   ```
   $class=atg.commerce.pricing.priceLists.PriceDroplet
   $scope=global
   priceListManager=/atg/commerce/pricing/priceLists/PriceListManager
   profilePriceListPropertyName=salePriceList
   useDefaultPriceList=false
   ```

   This servlet bean is used in the same way as the `PriceDroplet` servlet bean.

Calculating Prices with a Specific Price List

You can specify a price list that will be used to price items regardless of what price list is specified in a user's profile.

`ItemPriceCalculator`, `ConfigurableItemPriceListCalculator`, and `ConfigurableItemPriceListSaleCalculator` all look in `pExtraParameters` for the price list before looking in the profile. You can set a specific price list by adding an entry to `pExtraParameters` with a key of `profilePriceListPropertyName` and a value of the `priceList` that you with to use (or the ID of the `priceList`). For example, if `extraParameters` maps the string `priceList` to a price list (or a price list ID) and `profilePriceListPropertyName` is set to `priceList` (default), then the price list in the map is used instead of the profile's price list.

One way to implement this is to use the generic pipeline processor `AddExtraParamsEntry`. It adds pipeline support for specifying price lists. `AddExtraParamsEntry` adds a string key and string value to the extra parameters map.

To use this with pricing, create a properties file in `/atg/commerce/pricing/processor/UseDifferentPriceList.properties` with the following code:

```
$class=atg.service.pipeline.processor.AddExtraParamsEntry
     value=200005
```

```
     key=priceList
```

Then modify the `repriceOrderChain` in the `commercepipeline.xml` as follows:

```
<pipelinechain name="repriceOrder" transaction="TX_REQUIRED"
               headlink="useDifferentPriceList">
   <pipelinelink name="useDifferentPriceList" transaction="TX_MANDATORY">
      <processor jndi="/atg/commerce/pricing/processor/UseDifferentPriceList"/>
      <transition returnvalue="1" link="priceOrderTotal"/>
      <transition returnvalue="2" link="priceOrderTotal"/>
   </pipelinelink>
   <pipelinelink name="priceOrderTotal" transaction="TX_MANDATORY">
      <processor jndi="/atg/commerce/pricing/processor/PriceOrderTotal"/>
   </pipelinelink>
</pipelinechain>
```

This causes all orders (regardless of what is in the profile) to be priced with `priceList` 100012.

Using the CurrencyConversionFormatter to Convert Currency

The `CurrencyConversionFormatter` servlet bean can be used to convert and format a numeric amount. The amount can be converted from one currency to another. For more information on this servlet bean, see *Appendix B: ATG Servlet Beans* in the *ATG Page Developer's Guide*.

Price List Security Policy

The ATG Control Center allows users to create, edit, and delete price lists. When a user attempts to view or edit a price list, the security system checks the security information associated with the object and grants or denies access based on the information. For example, if a user does not have write access to a particular item, then the ACC will display the item in gray characters. Additionally, certain objects might not be visible to certain users. The ACC is capable of checking this security information for all items contained in the price list repository:

- Price List
- Prices
- Complex Prices
- Folders

While having the ability to specify security information for each item is a very powerful concept, it can place a burden on both the system as well as the administrator entering security information. To alleviate this burden, policies can be created that group logical items together. By having a logical policy, users would only need to enter data for some of the items and then other items could derive their security information from these

few items. This prevents an administrator from having to enter security information for every object in the repository.

Note: You can also plug in a different security policy if your business needs are not met by the policy described in this section.

For more information, see the discussion on security measures for deployment in the *ATG Installation and Configuration Guide*.

The default security policy returns the ACL information stored on each repository item. The price list security policy "walks" up the tree until an item finds the `priceList` to which it belongs and then retrieves the security information from the price list item.

In the price list security policy, all security information flows from a `priceList` down. This means that if there is a group of `price` and `complexPrice` that live in a `priceList`, these objects will have the same security information as the `priceList`. Therefore, if only users in the admin group can edit a particular price list, then those same users would be the only ones that could edit the price entries in the price list.

In the following example, all objects under Price List A would share the same security information.

Price List A
Price Entry for SKU A
Price Entry for SKU B
Complex Price

The PriceListSecurityPolicy Class

The `PriceListSecurityPolicy` class is located in the `atg.commerce.security` package. The class needs to has the following signatures:

```
public class PriceListSecurityPolicy
  extends SecuredRepositorySecurityPolicy
{
  // overridden method from the super class.  This is method
  // that will perform special logic to get ACLs for a repository
  // item that lives in the PriceList repository.  It should
  // figure out if the repository item type is "interesting"
  // and then dispatch to an appropriate method. The methods
  // it could dispatch to are below.
  public AccessControlList getEffectiveAccessControlList(Object pObject);

  // get the ACL for a Price repository item
  protected AccessControlList getACLForPrice(SecuredRepositoryItem pItem;)

  // get ACL for complexPrice repository item
  protected AccessControlList getACLForComplexPrice(SecuredRepositoryItem pItem);
}
```

Configuring the Price List Security Policy

Follow these steps to implement the security policy used by the `SecuredPriceList` repository:

Note: These configuration steps should be performed at the Commerce configuration layer.

1. Create a new `PriceListSecurityPolicy` component located in `/atg/dynamo/security/`. This component should have `PriceListSecurityPolicy` as its class.

2. Create a new `PriceListSecurityConfiguration` component located in `/atg/dynamo/security/`. This component should reference the `PriceListSecurityPolicy` component created in the previous step.

3. Edit the configuration of the `SecurePriceLists` component located in `/atg/commerce/pricing/priceLists/`. Point the `securityConfiguration` property to the `PriceListSecurityConfiguration` component defined in the previous step.

Converting a Product Catalog to Use Price Lists

Oracle ATG Web Commerce includes a tool that can be used to convert your existing product catalog without price lists so that it can use price lists.

The `PriceListMigration` component first creates two price lists: one for the list price, and the other for the sales price. For each SKU in the product catalog, the `PriceListMigration` component creates a price that points to the list price list and sets its list price as the SKU's list price. If the `onSale` property for the SKU is true, it creates another price that points to the sales price list and set its list price as the SKU's `salePrice`.

To use the `PriceListMigration` component:

1. Run the price list SQL script against your catalog database. This script can be found at:

   ```
   <ATG10dir>/DCS/sql/db_components/dbvendor/priceLists_ddl.sql
   ```

2. Start the `/atg/commerce/pricing/priceLists/PriceListMigration` component.

3. Open the component editor for the `PriceListMigration` component. Invoke the `runMigration` method from the methods tab in the component editor.

14 Working With Purchase Process Objects

This chapter includes basic information on the various commerce objects used in the purchase process and how they interrelate. It includes the following sections:

The Purchase Process Subsystems

The purchase process can be broken down into the following subsystems:

- Base Commerce Classes and Interfaces (page 224)
  The base commerce interface implementations hold and manage other commerce interface implementations. For example, an `Order` interface implementation contains implementations of the `CommerceItem`, `ShippingGroup`, `PaymentGroup`, and `Relationship` interfaces. These interfaces define a mechanism for accessing the data stored in an object in a manner that frees it from the underlying implementation.

- Address Classes (page 227)
  While not commerce-specific objects, the `Address` and `ContactInfo` classes play important roles in the purchase process.

- Business Layer Classes (page 227)
  The business layer classes hold business logic for tasks like adding an item to an order, retrieving an order, adding shipping methods, and adding payment methods. These classes use the base commerce classes and the base commerce interfaces.

- Pipelines (page 232)
  The purchase process pipelines execute a series of operations when called by the business layer classes.

- Order Repository (page 232)
  The Order repository is the layer between Oracle ATG Web Commerce and the database server.

Base Commerce Classes and Interfaces

The base commerce classes and interfaces are core objects that are used throughout Oracle ATG Web Commerce. These objects store the data that represent shopping carts, items to be purchased, shipping information, pricing information, and payment information. All the individual parts of ATG Commerce use these classes.

The Commerce interfaces are described in this section.

Interface	Description
`Order`	The `Order` interface represents the delivery and payment information for a collection of items. An `Order` contains `CommerceItems`, `ShippingGroups`, `PaymentGroups`, and `Relationships`.
`CommerceItem`	The `CommerceItem` interface represents information about a product to be purchased. A `CommerceItem` contains the SKU (also called the `catalogRefId`) and the quantity of the item purchased.
`ShippingGroup`	The `ShippingGroup` interface contains information about the delivery of a collection of `CommerceItem` objects. A `ShippingGroup` could contain a physical delivery address.
`PaymentGroup`	The `PaymentGroup` interface contains payment information, shipping costs, and tax information for each item or the entire `Order`. This includes information such as a credit card number, an expiration date, and the amount to be charged.
`Relationship`	The `Relationship` interface represents an association between two or more of the commerce objects listed above, such as the relationship between a `CommerceItem` and a `ShippingGroup`. It is important to understand the concept of relationships, although they are usually hidden from the Commerce user. The commerce-specific interfaces that extend `Relationship` are `CommerceItemRelationship`, `ShippingGroupRelationship`, `PaymentGroupRelationship`, and `OrderRelationship`. For more information, see the Using Relationship Objects (page 243) section.
`HandlingInstruction`	The `HandlingInstruction` interface describes special handling for a `CommerceItem` within a given `ShippingGroup`. Gift wrapping is an example of `HandlingInstruction`.

The following types of classes implement the interfaces described above:

Order Classes (page 225)

Item Classes (page 225)

Shipping Classes (page 225)

Payment Classes (page 226)

Relationship Classes (page 226)

Handling Classes (page 227)

In some cases, there may be only one implementation. Other interfaces are implemented by more than one class. For example, `PaymentGroupImpl`, `CreditCard`, and `GiftCertificate` all implement the `PaymentGroup` interface.

Order Classes

Class	Description
`OrderImpl`	This class implements `Order`. It contains data structures for managing collections of other commerce objects. It manages collections of `CommerceItem`, `ShippingGroup`, `PaymentGroup`, and `Relationship` objects.

Order equality is determined by comparing the `orderId`, the `lastModified` information, and transient properties.

If you write any new code that modifies the `Order` object, make sure the code synchronizes on the `Order` object before it is modified.

Item Classes

Class	Description
`CommerceItemImpl`	This class implements `CommerceItem`. It stores the data about a specific item in an `Order`.

Shipping Classes

Classes	Description
`ShippingGroupImpl`	This class implements `ShippingGroup`. It stores the data describing where and how to ship `CommerceItems`, as well as the `Relationships` to items in the shipping group. This class provides no functionality itself; it is used through the `HardgoodShippingGroup` and `ElectronicShippingGroup` subclasses (described below).
`HardgoodShippingGroup`	This class implements `ShippingGroup` and extends `ShippingGroupImpl`. In addition to storing inherited data, it stores information about how the `CommerceItems` are to be shipped to a physical street address, such as the carrier and postal address.

Classes	Description
ElectronicShippingGroup	This class implements ShippingGroup and extends ShippingGroupImpl. In addition to storing inherited data, it stores information about how CommerceItems are to be delivered electronically, such as an e-mail address.

Payment Classes

Payment Classes	
PaymentGroupImpl	This class implements PaymentGroup. It stores the payment information for CommerceItems, shipping costs, and tax. It also can contain Relationships to those items, shipping costs, or tax costs in the PaymentGroup. This class provides no functionality itself, but is used through the CreditCard and GiftCertificate subclasses (described below in this table).
CreditCard	This class implements PaymentGroup and extends PaymentGroupImpl. In addition to storing inherited data, it stores information about how CommerceItems, shipping costs, and tax costs are paid for using a credit card.
GiftCertificate	This class implements PaymentGroup and extends PaymentGroupImpl. In addition to storing inherited data, it stores information about how CommerceItems, shipping costs, and tax costs are paid for using a gift certificate.
StoreCredit	This class implements PaymentGroup and extends PaymentGroupImpl. In addition to storing inherited data, it stores information about how CommerceItems, shipping costs, and tax costs are paid for using a store credit.

Relationship Classes

Classes	Description
ShippingGroupCommerceItemRelationship	When this Relationship is added to a ShippingGroup and CommerceItem, the CommerceItem is shipped using the information in the ShippingGroup. This Relationship object contains data such as the quantity to be shipped.
PaymentGroupCommerceItemRelationship	When this Relationship is added to a PaymentGroup and CommerceItem, the CommerceItem is paid for using the information in the PaymentGroup. This Relationship object contains data such as the quantity of the item to be paid for using the PaymentGroup.

Classes	Description
`PaymentGroupShippingGroupRelationship`	When this `Relationship` is added to a `PaymentGroup` and `ShippingGroup`, the shipping cost is paid for using the information in the `PaymentGroup`. This `Relationship` object contains data such as the amount of the shipping cost to be paid for using the `PaymentGroup`.
`PaymentGroupOrderRelationship`	When this `Relationship` is added to a `PaymentGroup` and `Order`, the tax cost or order cost is paid for using the information in the `PaymentGroup`. This `Relationship` object contains data such as the amount of the tax cost to be paid for using the `PaymentGroup`.

Handling Classes

Classes	Description
`HandlingInstructionImpl`	This class implements `HandlingInstruction`. It contains a `ShippingGroup` ID, `CommerceItem` ID, and `quantity`, as well as data about the quantity of `CommerceItems` in a `ShippingGroup` needs special handling. This class provides no functionality itself, but should be used through a subclass.
`GiftlistHandlingInstruction`	This class implements `HandlingInstruction` and extends `HandlingInstructionImpl`. In addition to storing all the basic data that it inherits, it also stores data about which `CommerceItems` in the `Order` were added from a gift list.

Address Classes

There are two address classes that are not technically a part of Oracle ATG Web Commerce, but they play an important role in many commerce processes. They are `atg.core.util.Address` and `atg.core.util.ContactInfo`. These objects are referenced when a user checks out a shopping cart. You may need to extend these objects to track additional information about your users.

See the *Setting Up a Profile Repository* chapter in the *ATG Personalization Programming Guide* for information on adding properties to user profiles.

Business Layer Classes

The business layer classes contain logic and business rules for the purchase process. The methods in these classes are used to make changes to an `Order`. These methods contain logic that alters the `Order`'s data structure and maintains its accuracy; all calls to alter an `Order` should be made through these classes.

Class	Description
`OrderTools`	Low-level interface containing the logic for editing an `Order` data structure. It is not meant to contain business logic or to be used directly. In general, only the `OrderManager` or `SimpleOrderManager` makes calls to an `OrderTools` object. For more information, see the OrderTools (page 228) section.
`OrderManager`	Contains most of the functionality for working with an `Order`, including methods such as `createOrder()`, as well as properties referencing the other manager classes listed below. The methods in this class are higher level than the methods in `OrderTools`.
`CommerceItemManager`	Contains functionality for working with a `CommerceItem`, including methods such as `addCommerceItemToOrder()` and `addItemQuantityToShippingGroup()`.
`ShippingGroupManager`	Contains functionality for working with `ShippingGroups`, including `createShippingGroup()` and `splitShippingGroup()`.
`HandlingInstructionManager`	Contains functionality for working with handling instructions, including methods such as `createHandlingInstruction()` and `removeHandlingInstructionsFromShippingGroup()`.
`PaymentGroupManager`	Contains functionality for working with `PaymentGroups`, including `createPaymentGroup()` and `intializeCreditCard()`.
`OrderQueries`	Contains lookup methods such as `getOrdersForProfile()` and `getOrderIdsWithinDateRange()`.
`SimpleOrderManager`	This class extends `OrderManager`. It is a very high-level interface for altering an `Order`; only one method call in `SimpleOrderManager` is required to make a series of changes to an `Order`. For more information, see the Using the SimpleOrderManager (page 242) section.

OrderTools

`OrderTools` contains a set of properties that you can use to customize the purchase process. The default property settings should be suitable for most sites, but can be changed. The `OrderTools` component is located in Nucleus at `/atg/commerce/order/`.

The following `OrderTool` properties can be customized:

orderTypeClassMap (page 229)

defaultOrderType (page 229)

commerceItemTypeClassMap (page 229)

defaultCommerceItemType (page 229)

shippingTypeClassMap (page 230)

defaultShippingGroupType (page 230)

defaultShippingGroupAddressType (page 230)

paymentTypeClassMap (page 230)

defaultPaymentGroupType (page 231)

defaultPaymentGroupAddressType (page 231)

relationshipTypeClassMap (page 231)

beanNameToItemDescriptorMap (page 231)

You can view the `OrderTools` component in the ACC to see the configured values for these properties.

orderTypeClassMap

This property defines the type-to-class name mapping for `Order` objects. You can have more than one type of `Order` object. When creating a new `Order`, a string is passed as a parameter to the create method. (For example, the string "default" could be passed.) This constructs an instance of an `Order` class that is mapped to that string.

Below is a sample of how a mapping is defined in the properties file. The following code defines the default values:

```
orderTypeClassMap=\
default=atg.commerce.order.OrderImpl,\
shoppingcart=atg.commerce.order.OrderImpl
```

Note: The `shoppingcart` order type is no longer used and remains defined only for backwards compatibility.

defaultOrderType

This property defines the default `Order` type. In the example below, the default type is defined as the string "default," which in turn maps to a class in the `orderTypeClassMap` property. The following code defines the default value:

```
defaultOrderType=default
```

commerceItemTypeClassMap

This property defines the type-to-class name mapping for `CommerceItem` objects. You can have more than one type of `CommerceItem` object. When creating a new `CommerceItem`, a string is passed as a parameter to the create method. (For example, the string "default" could be passed.) This constructs an instance of a `CommerceItem` class that is mapped to that string.

Below is a sample of how a mapping is defined in the properties file. The following code defines the default values:

```
CommerceItemTypeClassMap=\
     default=atg.commerce.order.CommerceItemImpl
```

defaultCommerceItemType

This property defines the default `CommerceItem` type. In the example below, the default type is defined as the string "default," which in turn maps to a class in the `commerceItemTypeClassMap` property. The following code defines the default value:

```
defaultCommerceItemType=default
```

shippingTypeClassMap

This property defines the type-to-class name mapping for `ShippingGroup` objects. You can have more than one type of `ShippingGroup` object. When creating a new `ShippingGroup`, a string is passed as a parameter to the create method. (For example, the string `hardgoodShippingGroup` could be passed.) This constructs an instance of a `ShippingGroup` class that is mapped to that string.

Below is a sample of how a mapping is defined in the properties file:

```
shippingTypeClassMap=\
default=atg.commerce.order.ShippingGroupImpl,\
hardgoodShippingGroup=atg.commerce.order.HardgoodShippingGroup
```

defaultShippingGroupType

This property defines the default `ShippingGroup` type. In the example below, the default type is defined as the string `hardgoodShippingGroup`, which in turn maps to a class in the `shippingTypeClassMap` property. The following code defines the default value:

```
defaultShippingGroupType=hardgoodShippingGroup
```

defaultShippingGroupAddressType

This property defines the default `ShippingGroupAddressType`.

```
defaultShippingGroupAddressType=RepositoryContactinfo
```

To customize your address information, subclass `RepositoryContactInfo` and use your new class name for the `defaultShippingGroupAddressType`.

paymentTypeClassMap

This property defines the type-to-class name mapping for `PaymentGroup` objects. You can have more than one type of `PaymentGroup` object. When creating a new `PaymentGroup`, a string is passed as a parameter to the create method. (For example, the string `creditCard` could be passed.) This constructs an instance of a `PaymentGroup` class that is mapped to that string.

Below is a sample of how a mapping is defined in the properties file. The following code defines the default values:

```
paymentTypeClassMap=\
default=atg.commerce.order.PaymentGroupImpl,\
creditCard=atg.commerce.order.CreditCard,\
giftCertificate=atg.commerce.order.GiftCertificate,\
storeCredit=atg.commerce.order.StoreCredit
```

defaultPaymentGroupType

This property defines the default `PaymentGroup` type. In the example below, the default type is defined as the string `creditCard` , which in turn maps to a class in the `paymentTypeClassMap` property. The following code defines the default value:

```
defaultPaymentGroupType=creditCard
```

defaultPaymentGroupAddressType

This property defines the default `PaymentGroupAddressType`.

```
defaultPaymentGroupAddressType=RepositoryContactinfo
```

To customize your address information, subclass `RepositoryContactInfo` and use your new class name for the `defaultPaymentGroupAddressType`.

relationshipTypeClassMap

This property defines the type-to-class name mapping for `Relationship` objects. You can have more than one type of `Relationship` object. Relationships are not created directly by a user. Relationships are created by methods based on the type of relationship that the method needs.

The `relationshipTypeClassMap` property maps a name to a class. It is used to configure the class type that will be instantiated when a request to construct a relationship is made. By overriding the default values, you can customize the environment to use a `Relationship` class you have subclassed.

The example below demonstrates how a mapping is defined in the properties file. This mapping does not need to be modified unless the system is extended. The following code defines the default values:

```
relationshipTypeClassMap =\

shippingGroupCommerceItem=atg.commerce.order.
                          ShippingGroupCommerceItemRelationship,\
paymentGroupCommerceItem=atg.commerce.order.
                         PaymentGroupCommerceItemRelationship,\
paymentGroupShippingGroup=atg.commerce.order.
                          PaymentGroupShippingGroupRelationship,\
paymentGroupOrder=atg.commerce.order.
                  PaymentGroupOrderRelationship
```

beanNameToItemDescriptorMap

This property maps a bean name to an `OrderRepository` item descriptor name. When saving an Order, the processors look for an `OrderRepository` item descriptor with the same name as the bean class. If no such item descriptor is found, it looks for one with the same base type as the given item descriptor. The `beanNameToItemDescriptorMap` property contains this mapping.

All objects that can be mapped to an item descriptor are listed in the `beanNameToItemDescriptorMap`. The format is *bean name=repository item descriptor*. The example below demonstrates how a mapping is defined in the properties file. The following code defines the default values:

```
beanNameToItemDescriptorMap=\
atg.commerce.order.OrderImpl=order,\
atg.commerce.order.CommerceItemImpl=CommerceItem,\
atg.commerce.order.ShippingGroupImpl=shippingGroup,\
atg.commerce.order.HardgoodShippingGroup=hardgoodShippingGroup,\
atg.commerce.order.PaymentGroupImpl=paymentGroup,\
atg.commerce.order.CreditCard=creditCard,\
atg.commerce.order.GiftCertificate=giftCertificate,\
atg.commerce.order.ShippingGroupCommerceItemRelationship=shipItemRel,\
atg.commerce.order.PaymentGroupCommerceItemRelationship=payItemRel,\
atg.commerce.order.PaymentGroupShippingGroupRelationship=payShipRel,\
atg.commerce.order.PaymentGroupOrderRelationship=payOrderRel,\
atg.payment.PaymentStatus=paymentStatus,\
atg.commerce.pricing.OrderPriceInfo=orderPriceInfo,\
atg.commerce.pricing.ItemPriceInfo=itemPriceInfo,\
atg.commerce.pricing.TaxPriceInfo=taxPriceInfo,\
atg.commerce.pricing.ShippingPriceInfo=shippingPriceInfo,\
atg.commerce.pricing.DetailedItemPriceInfo=detailedItemPriceInfo,\
atg.commerce.pricing.PricingAdjustment=pricingAdjustment
```

Pipelines

A pipeline is an execution mechanism that allows for modular code execution. Oracle ATG Web Commerce uses pipelines to execute tasks such as loading, saving, and checking out `Orders`. The `PipelineManager` implements the pipeline execution mechanism.

The commerce pipeline is defined in an XML file, which can be found at `<ATG10dir>/DCS/config/atg/commerce/commercepipeline.xml`. To execute a pipeline through JSPs, define a handle method in a form handler class that calls the `PipelineManager`.

In the Nucleus hierarchy, the `PipelineManager` is located at `/atg/commerce/PipelineManager`. When you deploy an application that includes Commerce, a new instance is created and the commerce pipeline configuration is loaded. The commerce pipeline configuration file contains the definition for the `processOrder` chain. To insert a new link, add a new element to the XML file that references the new pipeline processor. The new functionality is inserted into the execution chain without affecting the existing code.

A pipeline should generally be executed from an `OrderManager` method. This is the case for the `loadOrder()`, `updateOrder()`, and `processOrder()` methods.

For more information, see the *Processor Chains and the Pipeline Manager* (page 363) chapter in this manual.

Order Repository

The Order repository is the layer between Oracle ATG Web Commerce and the database server. The repository is where `Orders` are saved after processing and stored in between customers' visits. It is implemented using a SQL repository.

The Order repository definition file defines the item descriptors for all commerce classes; for every class that is saved, there exists a corresponding item descriptor. Each item descriptor defines a repository item type that describes all the properties that are common to the repository items of that type. Additionally, each item descriptor subtype inherits all of the properties of its parent item descriptor. For example, the `hardgoodShippingGroup` item descriptor extends the `shippingGroup` item descriptor, so it inherits all of the properties of the `shippingGroup` item descriptor.

The Order repository definition file is located at `<ATG10dir>/DCS/config/atg/commerce/order/orderrepository.xml`.

The `beanNameToItemDescriptorMap` property of the `OrderTools` component maps the Order repository item descriptors to bean names (see the beanNameToItemDescriptorMap (page 231) section in this chapter for details). In Oracle ATG Web Commerce, the processors that save and load an `Order` look for an item descriptor that is mapped to the corresponding commerce object class; the `beanNameToItemDescriptorMap` property contains this mapping.

For more information about the `OrderTools` component, see the OrderTools (page 228) section of this chapter. For information about saving and loading orders, see the *Configuring Purchase Process Services* (page 263) chapter. For more information on SQL repositories, see the *ATG Repository Guide*.

Creating Commerce Objects

The Oracle ATG Web Commerce manager classes contain methods for creating commerce objects and adding them to and removing them from an `Order`. This section describes how to use the manager classes to create commerce objects. It includes the following subsections:

Creating an Order

The first step in working with Commerce objects is to create an `Order`. A shopping cart is an implementation of the `Order` interface. To create an `Order`, you must have a reference to an `OrderManager` or `SimpleOrderManager`. Once you have the reference, use `createOrder()` to create the new `Order`.

There are many versions of the `createOrder` method, each of which takes a different set of parameters:

```
createOrder(String pProfileId)

createOrder(String pProfileId, String pOrderType)

createOrder(String pProfileId, String pOrderId, String pOrderType)

createOrder(String pProfileId, OrderPriceInfo pOrderPriceInfo,
TaxPriceInfo pTaxPriceInfo, ShippingPriceInfo pShippingPriceInfo)
```

```
createOrder(String pProfileId, OrderPriceInfo pOrderPriceInfo,
TaxPriceInfo pTaxPriceInfo, ShippingPriceInfo pShippingPriceInfo,
String pOrderType)
```

```
createOrder(String pProfileId, String pOrderId, OrderPriceInfo
pOrderPriceInfo, TaxPriceInfo pTaxPriceInfo, ShippingPriceInfo
pShippingPriceInfo, String pOrderType)
```

All methods create an `Order` object and assign it a unique ID. The type of `Order` created depends on the method used. If the method takes an `orderType` parameter, that parameter determines the type of object that is constructed. Otherwise, the `defaultOrderType` property of the `OrderTools` component (see OrderTools (page 228) in this chapter) defines the `Order` type.

By default, an `Order` contains one empty `ShippingGroup` and one empty `PaymentGroup` when it is created, and their types are determined by the `defaultShippingGroupType` and `defaultPaymentGroupType` properties of the `OrderTools` component.

Note: If you do not want to create an empty `ShippingGroup` and an empty `PaymentGroup` for every new `Order`, set the `createDefaultShippingGroup` and `createDefaultPaymentGroup` properties in the `OrderTools` component to false.

The following example demonstrates how to create an `Order`:

```
// Get a reference to the OrderManager
OrderManager orderManager = (OrderManager)
  request.resolveName("/atg/commerce/order/OrderManager");

// Create the Order
Order order = orderManager.createOrder(profileId);
```

By default, orders are persistent. To disable persistence, set the `persistOrders` property in the `ShoppingCart` component to `false`. The `ShoppingCart` component is located in Nucleus at `/atg/commerce/`.

Using Orders in a Multisite Environment

If you are using Oracle ATG Web Commerce's multisite feature, you may want to provide users with the ability to place items from multiple sites in a single order. You do not need to do any additional configuration to use this feature; the `ShoppingCart` is registered as a shareable component by default and works the same way in a multisite environment as in a single site.

To register the `ShoppingCart` as a shareable component, it is added by default to the `NucleusComponentShareableType` component:

```
shareableTypes+=/atg/multisite/ShoppingCartShareableType
```

The shareable component that refers to the `ShoppingCart` can be configured in the following properties file:

```
/atg/commerce/ShoppingCartShareableType.properties
```

See the *ATG Multisite Administration Guide* for information on shareable components and how to use sharing groups in your multisite configuration.

Creating Multiple Orders

Customers can have an unlimited number of orders at one time. They can place items in different shopping carts, switch between carts, retrieve a list of saved carts, delete carts, and can check out one cart's contents while waiting until later to check out the contents of others.

Using multiple orders requires `atg.commerce.order.OrderHolder` in addition to `atg.commerce.order.Order`. This class maintains the current `Order` object as well as a collection of saved `Order` objects. The component that utilizes `OrderHolder` is `/atg/commerce/ShoppingCart`, a session-scoped component whose handleXXX methods add, delete, and switch between carts, as explained in the rest of this section.

You implement multiple shopping carts using the `handleCreate` method of the `OrderHolder` class. This method creates a new `Order` and sets it as the `currentOrder` in the `OrderHolder`. Any previously existing `Order` object is placed into the collection of saved carts. Refer to the following JSP example:

```
<dsp:form action="ShoppingCart.jsp" method="post">
  <dsp:input bean="ShoppingCart.create" value="Create" type="submit"/> another
   shopping cart.<BR>
</dsp:form>
```

The `handleSwitch()` method allows customers to switch between shopping carts. It switches the current `Order` object out to the saved collection of orders and sets the current `Order` to the `Order` identified by the `handlerOrderId` property. If a customer has several shopping carts saved, you can allow them to switch between any of the `Order` objects using the following JSP code:

```
<dsp:form action="ShoppingCart.jsp" method="post">
 <dsp:select bean="ShoppingCart.handlerOrderId">
  <dsp:droplet name="ForEach">
   <dsp:param bean="ShoppingCart.saved" name="array"/>
   <dsp:param value="SavedOrder" name="elementName"/>
   <dsp:oparam name="output">
    <dsp:getvalueof id="option12" param="SavedOrder.id" idtype="java.lang.String">
<dsp:option value="<%=option12%>"/>
</dsp:getvalueof>
     <valueofparam="SavedOrder.id"></dsp:valueof>
    </dsp:oparam>
   </dsp:droplet>
 </dsp:select>
 <dsp:input bean="ShoppingCart.switch" value="Switch" type="submit"/>
</dsp:form>
```

The example iterates through the list of saved shopping carts, displays the shopping carts to the customer, and gives the customer the option to select one of the saved carts. The `handlerOrderId` property would be set to the selected `Order` ID, and the corresponding `Order` would be set as the current `Order`.

The `handleDelete()` and `handleDeleteAll()` methods remove a single `Order` or all orders (both current and saved), respectively.

Creating Commerce Items, Shipping Groups, and Payment Groups

After an `Order` has been created, the next step is to add `CommerceItems` and possibly additional `ShippingGroups` and `PaymentGroups`. Creating these objects follows the same pattern as

creating `Orders`. First, call `create*ObjectType*()` in the appropriate manager class, for example, `CommerceItemManager.createCommerceItem()`. The call returns the type of class specified by the `objectType` parameter, or the default type when the method used does not accept that parameter.

After creating the objects, use the following methods to add them to the `Order`:

- `CommerceItemManager.addItemToOrder()`
- `ShippingGroupManager.addShippingGroupToOrder()`
- `PaymentGroupManager.addPaymentGroupToOrder()`

These methods take an `Order` and their respective object type as parameters. For `PaymentGroups`, the order in which they are added to the `Order` determines their precedence.

For more information on creating these commerce objects and adding them to an order, refer to the following subsections:

- Creating a Standard Commerce Item (page 236)
- Creating a Configurable Commerce Item (page 237)
- Creating a Shipping Group (page 238)
- Creating a Payment Group (page 239)

Creating a Standard Commerce Item

Follow these steps to create a new `CommerceItem` and associate it with an `Order`:

1. Call `CommerceItemManager.createCommerceItem()`.
2. Make any changes to the `CommerceItem`, such as setting the quantity.
3. Call `CommerceItemManager.addItemToOrder(pOrder, pCommerceItem)` to add the `CommerceItem` to the `Order`.

Refer to the following example:

```
// Get a reference to the OrderManager
OrderManager orderManager = (OrderManager)
  request.resolveName("/atg/commerce/order/OrderManager");

// Create the CommerceItem
CommerceItem commerceItem =
  commerceItemManager.createCommerceItem(pCatalogRefId);
commerceItem.setQuantity(3);
// Add the CommerceItem to the Order
commereceItemManager.addItemToOrder(pOrder, commerceItem);
```

Note: `createCommerceItem()` will work even if you pass it a nonexistent catalog reference ID. This allows you to use Oracle ATG Web Commerce as an ordering system with multiple catalogs, some of which may not have repositories. If you want to prevent this behavior, you must institute a check.

`CommerceItem` objects can include an auxiliary data construct. This structure allows you to store arbitrary data with a `CommerceItem`. Examples of auxiliary data could include size and color options for a `CommerceItem`. If

your system includes remote components, auxiliary data can be serialized at any time; however, when defining `AuxiliaryData` objects, the classes must be defined as serializable.

By default, the class includes `ProductID`, `ProductRef`, `PropertyValue`, and `CatalogRef` properties.

Creating Commerce Items in a Multisite Environment

If you are using Oracle ATG Web Commerce in a multisite-enabled installation, some additional methods and tools for manipulating Commerce items may be useful.

Follow these steps to create a new `CommerceItem` and associate it with an `Order` in a multisite setting:

1. Call `CommerceItemManager.createCommerceItem(…, …, pSiteId)`.

   The `createCommerceItem()` method is overloaded in the Oracle ATG Web Commerce API. One version is intended for use with Oracle ATG Web Commerce's multisite feature, and includes a `pSiteId` parameter. If the `siteId` is null but multisite is in use, the method retrieves the ID from `SiteContextManager.getCurrentSiteId`.

2. Make any changes to the `CommerceItem`, such as setting the quantity.

3. Call `CommerceItemManager.addItemToOrder(pOrder, pCommerceItem)` to add the `CommerceItem` to the `Order`.

   In a multisite setting, the `addItemToOrder()` method and the related `addAsSeparateItemToOrder()` method use the `validateSiteCompatibility()` method to ensure that the item can be added to the order. If the item's `siteId` is not null, that `siteId` is added to the order's `siteId`. If the order's `creationSiteId` is null, the item's `siteId` is copied there as well.

   The `validateSiteCompatibility()` check is also used when creating configurable Commerce items (see Creating a Configurable Commerce Item (page 237)).

Refer to the following example:

```
// Get a reference to the OrderManager
OrderManager orderManager = (OrderManager)
  request.resolveName("/atg/commerce/order/OrderManager");

// Create the CommerceItem
CommerceItem commerceItem =
  commerceItemManager.createCommerceItem(pCatalogRefId);
commerceItem.setQuantity(3);
// Add the CommerceItem to the Order
commereceItemManager.addItemToOrder(pOrder, commerceItem);
```

Auxiliary data holds the `siteId` property in a multisite-enabled configuration; you can access this property at `item.auxiliaryData.siteId`.

Creating a Configurable Commerce Item

Configurable commerce items are items with other items as optional components, and are described in the *Using and Extending the Product Catalog* (page 23) chapter of this manual.

Follow these steps to create a `ConfigurableCommerceItem` and associate it with an `Order`:

1. Call `CommerceItemManager.createCommerceItem()` to create the base commerce item.

2. Call `CommerceItemManager.addSubItemToConfigurableItem()` or `addAsSeparateSubItemToConfigurableItem()` to add options to the base item.

   Note: If you are using Oracle ATG Web Commerce's multisite feature, these methods check to ensure that the item can be added to the customer's cart (if more than one site is involved, the sites must share their shopping cart). See Creating Commerce Items in a Multisite Environment (page 237) for details.

The example below illustrates how to programmatically create a `ConfigurableCommerceItem` with subSKU items and then add it to an `Order`:

```
ConfigurableCommerceItem configurableItem = (ConfigurableCommerceItem)
getCommerceItemManager().createCommerceItem("configurableCommerceItem",
 "sku10001", null, "prod10001", null, 1, null, null, new ItemPriceInfo());

SubSkuCommerceItem subskuItem = (SubSkuCommerceItem)
getCommerceItemManager().createCommerceItem("subSkuCommerceItem",
"sku20001", null, "prod20001", null, 1, null, null, new ItemPriceInfo());
getCommerceItemManager().addSubItemToConfigurableItem(configurableItem,
subskuItem);

subskuItem = (SubSkuCommerceItem)
getCommerceItemManager().createCommerceItem("subSkuCommerceItem",
"sku20002", null, "prod20002", null, 1, null, null, new ItemPriceInfo());
getCommerceItemManager().addSubItemToConfigurableItem(configurableItem,
subskuItem);

getCommerceItemManager().addItemToOrder(order, configurableItem);
```

Creating a Shipping Group

A `ShippingGroup` contains information on the shipping address and delivery method for a group of commerce items. By default, a new `Order` has one default `ShippingGroup`. As items are added to the `Order`, these items automatically become part of the default `ShippingGroup`. Once a second `ShippingGroup` is added to the `Order`, all the items in the `Order` are removed from the default `ShippingGroup` and must be explicitly added to one of two shipping groups. Relationships must now be created to associate the items with shipping groups. (For more information, see Assigning Items to Shipping Groups (page 248).)

Follow these steps to create a new `ShippingGroup` and add it to an `Order`:

1. Call `ShippingGroupManager.createShippingGroup()`.

2. Make any changes to the `ShippingGroup`, such as setting the address.

3. Call `ShippingGroupManager.addShippingGroupToOrder(pOrder, pShippingGroup)` to add the `ShippingGroup` to the `Order`.

Refer to the following example:

```
// Get a reference to the OrderManager
OrderManager orderManager = (OrderManager)
  request.resolveName("/atg/commerce/order/OrderManager");

// Create the ShippingGroup
ShippingGroup shippingGroup = shippingGroupManager.createShippingGroup();
// Add the ShippingGroup to the Order
```

```
shippingGroup.addShippingGroupToOrder(pOrder, shippingGroup);
```

When setting the shipping and billing addresses, normally you pass a `RepositoryContactInfo` object to `setShippingAddress()` or `setBillingAddress()`. If you want to use a `ContactInfo` object instead, but do not want to subclass `RepositoryContactInfo` (see defaultShippingGroupAddressType (page 230) in the OrderTools (page 228) section), you must modify some Nucleus components. List the properties of your address object in the `savedProperties` property of `/atg/commerce/order/processor/SaveShippingGroupObjects` and the `loadProperties` property of `atg/commerce/order/processor/LoadShippingGroupObjects`.

Creating Multiple Shipping Groups

Multiple shipping groups (which implement `Multishipment`) on a commerce site permit a customer to ship parts of an order to different addresses using different methods of delivery.

For example, suppose a customer enters the checkout process on a site that supports multiple shipping methods for a single order. The customer chooses to have the items shipped to different locations. The site must provide a UI that allows the customer to enter an address, and then associate it with a shipper, such as UPS or US Postal.

After the customer selects a shipper, a `ShippingGroup` is created. The site must then provide a UI that allows the customer to associate items with that shipping group. If there is only one shipping group, then all the items to be shipped will go into that shipping group. If more than one shipping group is associated with the order, then the customer must decide which items go into each group.

Creating a Payment Group

A `PaymentGroup` contains information about the payment method that will be used to purchase a group of commerce items. By default, a new `Order` has one default `PaymentGroup`. As items are added to the `Order`, these items automatically become part of the default `PaymentGroup`. Once a second `PaymentGroup` is added to the `Order`, all the items in the `Order` are removed from the default `PaymentGroup` and must be explicitly added to one of the two payment groups. Relationships must now be created to associate the items with payment groups. (For more information, see the Assigning Costs to Payment Groups (page 249) section.)

Payment groups also contain a `requisitionNumber` property for orders that require approval before a means of payment can be specified. Orders with requisition numbers are automatically assumed to require approval. (See the *Managing the Order Approval Process* (page 445) chapter for information on approvals.)

Follow these steps to create a new `PaymentGroup` and add it to an `Order`:

1. Call `PaymentGroupManager.createPaymentGroup()`.
2. Make any changes to the `PaymentGroup`. For example, you could set the credit card number and expiration date.
3. Call `PaymentGroupManager.addPaymentGroupToOrder(pOrder, pPaymentGroup)` to add the payment group to the order.

Refer to the following example:

```
// Get a reference to the OrderManager
OrderManager orderManager = (OrderManager)
             request.resolveName("/atg/commerce/order/OrderManager");

// Create the PaymentGroup
```

```
PaymentGroup paymentGroup = paymentGroupManager.createPaymentGroup();
// Add the PaymentGroup to the Order
paymentGroupManager.addPaymentGroupToOrder(pOrder, paymentGroup);
```

Creating Multiple Payment Groups

Multiple payment groups follow a similar pattern to multiple shipping groups. The payment groups implement `Multipayment`, and permit a customer to split the cost of an order by amount or by items. For example, the customer might put the first $1000 of a $1250 order on a Visa credit card, then pay the the remaining $250 using points earned on the site during previous visits.

While customers can select payment methods by item level, amount level, or any combination of the two, you can limit the ways in which an order's costs can be split, if necessary.

Adding an Item to an Order via a URL

The `CommerceCommandServlet` provides a foundation for URL-based actions. Out of the box, you can allow a user to add an item to his or her order by clicking a URL. Because this process is part of the request-handling pipeline (which handles all Oracle ATG Web Commerce requests and responses), each time a page is requested, the request will be checked to see if it includes Commerce item information.

Consider the following example request URL:

```
http://yoursite.com/yourpage.jsp?dcs_action=additemtocart&url_catalog_ref_
id=sku10001&url_product_id=prod10001&url_quantity=1&dcs_ci_catalogKey=en_
US&dcs_subsku=sku10001,prod10001,2&dcs_subsku=sku10002,prod10002,1
```

The `dcs_action` flag notifies the request-handling pipeline of the action to be performed, and additional parameters give specifics for the action. The following table explains the URL elements:

Element	Function
`dcs_action`	(Required) When `CommerceCommandServlet` receives the `dcs_action additemtocart`, it calls the `AddItemToCartServlet`. You can extend the `CommerceCommandServlet` and the pipeline to let the user trigger some other action by clicking a URL. The `AddItemToCartServlet` component includes a `pricingOperation` parameter, which determines what action the servlet takes when called. The default action is `ORDER_TOTAL`. The possible actions are included in `atg.commerce.pricing.PricingConstants`.
`url_catalog_ref_id`	(Required) The SKU of the product to add to the cart.
`url_product_id`	(Required) The product ID of the product to add to the cart.
`url_quantity`	(Required) The quantity of the SKU to add to the cart.
`url_shipping_group_id`	The ID of the `ShippingGroup` to which to add the item.

Element	Function
`url_item_type`	A string specifying which `CommerceItem` type to use.
`url_commerce_item_id`	The ID of the `CommerceItem` which this configuration should replace. If this parameter is supplied, then this is a reconfiguration.
`dcs_ci_*`	An identifier for setting a `CommerceItem` property. For example, `dcs_ci_catalogKey=en_US` causes the property `catalogKey` to be set to the value `en_US`.
`dcs_subsku`	An identifier for a configurable property of a `CommerceItem`. The format is: `dcs_subsku=`*`sku id, product id, individual quantity`*. The individual quantity portion of this parameter reflects the quantity of the item which will be added to a single `ConfigurableCommerceItem`. For example, if you are adding a computer with 2 hard drives, the individual quantity would be 2. If you were adding 1000 computers, each with 2 hard drives, the individual quantity would still be 2. Oracle ATG Web Commerce handles the multiplication. If you need to add two different computer configurations, you must issue two separate requests to Commerce to add each of the configurations as separate `ConfigurableCommerceItems`. For example, `dcs_subsku=sku10001,prod10001,2&dcs_subsku=sku10002,prod10002,1` causes the servlet to add a `CommerceItem` with SKU/product combination sku10001, prod10001 and quantity 2 and SKU/product combination sku10002,prod10002 and quantity 1 to the `ConfigurableCommerceItem`.

The first four parameters listed in the table above are required. If no other parameters are supplied, the servlet creates a `CommerceItem` and adds it to the cart.

If a `dcs_subsku` parameter is in the URL, then the base SKU/product is represented in a `ConfigurableCommerceItem` object and the subSKU is represented in a `SubSkuCommerceItem` object. `SubSkuCommerceItem` is a subclass of `CommerceItemImpl`.

If all required parameters and `shipping_group_id` are supplied, then the item is added to the `ShippingGroup` with the specified ID.

Preventing Commerce Items from Being Added to Types of Shipping Groups

By default, all items are allowed to go into any type of `ShippingGroup`. However, it might be necessary to prevent specific commerce items from being added to certain types of shipping groups. For example, only certain types of items can be assigned to an `ElectronicShippingGroup`. It does not make sense for a box of golf balls to be shipped electronically.

Follow these steps to prevent commerce items from being added to specific types of shipping groups:

1. Add a new `shippingGroupsAllowed` property to the SKU item descriptor in the product repository. The `shippingGroupsAllowed` property should contain a list of shipping groups to which the SKU that this `CommerceItem` represents can be added.

   Note: The available `ShippingGroup` types are listed in the `shippingTypeClassMap` property of the `OrderTools` component.

2. Override any necessary methods to make sure the `shippingGroupsAllowed` property in the SKU that this `CommerceItem` represents matches the `shippingGroupClassType` property of the `ShippingGroup` to which the `CommerceItem` is being added.

   These methods are `addItemToShippingGroup()` in the `ShippingGroupManager`, and `addItemQuantityToShippingGroup` and `addRemainingItemQuantityToShippingGroup()` in the `CommerceItemManager`.

Alternatively, you could create a `shippingGroupsNotAllowed` property for the SKU item descriptor in the product repository. In this property, list the names of the `ShippingGroup` types to which a `CommerceItem` is not allowed.

Removing Commerce Objects from an Order

The counterparts to the `add` methods are the methods for removing commerce items, shipping groups, and payment groups from an `Order`. They are:

- `CommerceItemManager.removeItemFromOrder()`
- `ShippingGroupManager.removeShippingGroupFromOrder()`
- `PaymentGroupManager.removePaymentGroupFromOrder()`

These methods take an `Order` and their respective object ID as parameters. The object ID is found in the `id` property of the `CommerceItem`, `ShippingGroup`, or `PaymentGroup`.

To remove subitems from configurable commerce items, use one of these methods:

- `CommerceItemManager.removeSubItemFromConfigurableItem()`
- `CommerceItemManager.removeAllSubItemsFromConfigurableItem()`

Using the SimpleOrderManager

`SimpleOrderManager` extends `OrderManager`. By default, a `SimpleOrderManager` object is configured in Nucleus at `/atg/commerce/order/OrderManager`. Logically, the `SimpleOrderManager` sits above the `OrderManager` and provides higher-level functionality. What takes several lines of code to do in the `OrderManager` takes only a single line of code in the `SimpleOrderManager`. You can use `SimpleOrderManager` in place of `OrderManager` to simplify your code, as the example below shows.

```
String skuId = getSkuId();
String productId = getProductId();
long quantity = getQuantity();
ShippingGroup shippingGroup = getShippingGroup();

getSimpleOrderManager().addItemToShippingGroup(order, skuId, productId, quantity,
                                               shippingGroup);
```

Using Relationship Objects

A `Relationship` represents an association between two other objects. For example, a `ShippingGroupCommerceItemRelationship` represents a relationship between a `CommerceItem` and a `ShippingGroup`. The `Relationship` indicates that the `CommerceItem` will be handled according to the information in the `ShippingGroup`. The `Relationship` contains the `CommerceItem` to ship, the `ShippingGroup`, and the quantity of this item to ship to the address in the `ShippingGroup`.

Another `Relationship` example is a `PaymentGroupOrderRelationship`, which represents a relationship between an `Order` and the `PaymentGroup` responsible for the whole or partial cost of the `Order`. The `PaymentGroup` contains the credit card information. The `PaymentGroupOrderRelationship` contains the amount to charge, the `PaymentGroup`, and the `Order`.

In Oracle ATG Web Commerce, a new `Order` has one default payment group, one default shipping group, and no `Relationships`. While you never explicitly add a `Relationship` to an `Order`, you do so implicitly by calling methods like `addItemQuantityToShippingGroup()` or `addShippingCostAmountToPaymentGroup()`. When you create these additional associations, relationships handle the details of which payment or shipping method to use for which items. Below is a code sample that associates part of an order's cost with a specific `PaymentGroup`. A `PaymentGroupOrderRelationship` is constructed implicitly.

```
// Get a reference to the OrderManager
OrderManager om = (OrderManager)
                   request.resolveName("/atg/commerce/order/OrderManager");
// Creates a PaymentGroupOrderRelationship implicitly
om.addRemainingOrderCostToPaymentGroup(pOrder, pPaymentGroup);
```

Unlike most business objects, `Relationship` objects do not have their own manager class. Methods where the operator is a `Relationship` are placed in the operand's manager class. For example, a method for adding an `Order` to a `ShippingGroup` is in the `OrderManager`, and a method for assigning a `CommerceItem` to a `ShippingGroup` is in the `CommerceItemManager`. Methods that operate on a subclass of a given commerce object are placed in the superclass's manager class.

Relationship Types

`Relationship` objects have associated types. These types determine what happens when an `Order` is checked out, as well as what types the relationship's member variables have. This section describes the following `Relationship` objects and their possible relationship types:

- ShippingGroupCommerceItemRelationship Object (page 244)
- PaymentGroupOrderRelationship Object (page 244)
- PaymentGroupCommerceItemRelationship Object (page 245)
- PaymentGroupShippingGroupRelationship Object (page 246)

For details on assigning items to relationships of specific types, see the next two sections, Assigning Items to Shipping Groups (page 248) and Assigning Costs to Payment Groups (page 249).

Note: For all of the following relationship types, quantities and amounts designated by a number are processed as "up to and including" the given number. For example, if a `PaymentAmount` relationship exists with quantity 6 and references an item with quantity 10, 6 of those items will be paid for using the `PaymentGroup` that the relationship references. The remaining 4 will be handled by the next relationship whose priority is less than the first relationship. On the other hand, if the relationship contained quantity 15, then all 10 items will be paid for using that first `PaymentGroup`.

Also note that it is not possible to have more than one Relationship of a specific "remaining" type (described below) in any given object. For example, a given CommerceItem cannot have more than one PaymentGroupRelationship of a remaining type. However, it can have a PaymentGroupRelationship and a ShippingGroupRelationship – both of a remaining type – because the two relationships are separate entities.

ShippingGroupCommerceItemRelationship Object

A ShippingGroupCommerceItemRelationship represents a relationship between a CommerceItem and a ShippingGroup. This relationship can be assigned the relationship type ShippingQuantity or ShippingQuantityRemaining. These relationship types assign CommerceItems to ShippingGroups that specify which items in the Order will be shipped to each destination. The following list describes the relationship types:

- ShippingQuantity: This relationship type indicates that a specific quantity of the item will be shipped using the information in the ShippingGroup. The quantity to ship is stored inside the Relationship, and the quantity purchased is stored inside the CommerceItem. If the quantity in the Relationship is greater than the quantity in the CommerceItem, then all the CommerceItems are shipped to the location in the ShippingGroup.
- ShippingQuantityRemaining: This relationship type indicates that the remaining quantity of the item unaccounted for by other ShippingGroupCommerceItemRelationship objects will be shipped using the information in the ShippingGroup.

  Note: Any ShippingQuantityRemaining relationships have the lowest priority. A CommerceItem can have only one Relationship of type ShippingQuantityRemaining.

For example, a customer places an order for CommerceItem Apple with quantity 10. Two ShippingGroups already exist, SG1 (home) and SG2 (office).

The customer wants three apples shipped to his home, so a ShippingGroupCommerceItemRelationship is created between CI1 (apple) and SG1 (home). This relationship type is ShippingQuantity and the quantity to ship is 3.

The customer wants the remaining seven apples shipped to his office. A ShippingGroupCommerceItemRelationship is created between CI1 (apple) and SG2 (office). This relationship can have either the relationship type ShippingQuantity with quantity 7, or the relationship type ShippingQuantityRemaining.

The difference between creating a second relationship with type ShippingQuantity and using ShippingQuantityRemaining is that, using ShippingQuantity, if the quantity of CI1 increases, then the new items would not be assigned to a ShippingGroup. If the second relationship is of type ShippingQuantityRemaining, then the new items default to the ShippingGroup in that relationship.

Both relationship types use an atg.core.util.Range object to determine which particular CommerceItems to include for a given quantity. Set the Range's lowBound and highBound to indicate which items to include. Bounds are inclusive. For example, if one CommerceItemShippingGroupRelationship accounts for four out of six CommerceItems, setting the lowBound to 1 and the highBound to 4 means the first four CommerceItems are shipped using this relationship. Setting the lowBound to 3 and the highBound to 6 means the last four are shipping using this relationship.

Range information is calculated when the end user checks out, ensuring correct pricing for each CommerceItem regardless of any changes the user may make to ShippingGroups while browsing.

PaymentGroupOrderRelationship Object

A PaymentGroupOrderRelationship represents a relationship between an Order and a PaymentGroup. There are two ways to split the cost of an Order across PaymentGroups. Either split the entire cost by using

PaymentGroupOrderRelationships, or assign different types of costs (such as item cost vs. tax) to separate PaymentGroups.

A PaymentGroupOrderRelationship can be assigned the type OrderAmount, OrderAmountRemaining, TaxAmount, or TaxAmountRemaining. These relationship types assign the order and tax amounts to different PaymentGroups. The following list describes the relationship types:

- OrderAmount: This relationship type indicates that a specific amount of the total cost of the Order (including CommerceItems, ShippingGroups, and tax) will be paid using the information in the PaymentGroup. The amount must be greater than zero. If the relationship amount is greater than the total amount of the Order, then the cost of the entire Order will be paid using the referenced PaymentGroup.
- OrderAmountRemaining: This relationship type indicates that the remaining cost of the Order unaccounted for by other PaymentGroupOrderRelationship objects will be paid using the information in the PaymentGroup.

  Note: An Order can have only one Relationship of type OrderAmountRemaining.
- TaxAmount: This relationship type indicates that a specific amount of the tax charged for the Order will be paid using the information in the PaymentGroup. The amount must be greater than zero. If the relationship amount is greater than the total amount of the Order, then all the tax will be paid for using the referenced PaymentGroup.
- TaxAmountRemaining: This relationship type indicates that the tax cost unaccounted for by other PaymentGroupOrderRelationship objects will be paid using the information in the referenced PaymentGroup.

  Note: An Order can have only one Relationship of type TaxAmountRemaining.

Example #1

The following example describes how to use the OrderAmount and OrderAmountRemaining types.

A customer places an Order with a total of $600. The customer wants to pay for the order using two credit cards (Visa and MasterCard). A different PaymentGroup represents each credit card. Two relationships are created to split the payment:

- One PaymentGroupOrderRelationship is created between the Visa's PaymentGroup and the Order. The relationship type is set to OrderAmount, and the amount is set to $400.
- One PaymentGroupOrderRelationship is created between the MasterCard's PaymentGroup and the Order. The relationship type is set to OrderAmountRemaining.

When the Order is charged, $400 will be charged on the Visa and $200 will be charged on the MasterCard.

Example # 2

The following example describes how to use the TaxAmountRemaining type.

A customer places an order with a total tax of $100. The customer wants to pay for the tax with a separate credit card than the rest of the order payment. A PaymentGroupOrderRelationship is created between the PaymentGroup that represents the credit card and the Order. The relationship type is set to TaxAmountRemaining, which covers whatever the tax amount turns out to be.

PaymentGroupCommerceItemRelationship Object

A PaymentGroupCommerceItemRelationship represents a relationship between a CommerceItem and a PaymentGroup. This relationship type is used to split payment for a single CommerceItem between multiple payment groups.

The relationship can have the relationship type PaymentAmount or PaymentAmountRemaining. The following list describes the relationship types:

- PaymentAmount: This relationship type indicates that a specific amount of the item's cost will be paid for using the information in the PaymentGroup. The amount must be greater than zero. If the amount is greater than the total amount of the item stored in the CommerceItem, then the entire cost of the item will be paid for using the referenced PaymentGroup.
- PaymentAmountRemaining: This relationship type indicates that any remaining payment amount unaccounted for by other PaymentGroupCommerceItemRelationship objects will be paid using the information in the PaymentGroup. If there is only one relationship between a given CommerceItem and a PaymentGroup of type PaymentAmountRemaining, then the entire cost of that item will be paid for using the information in the PaymentGroup.

  Note: A CommerceItem can have only one Relationship of type PaymentAmountRemaining.

For example, a customer places an order for a CommerceItem (Car) with the total cost $10,000. The customer wants to split the payment of this item between three credit cards (Visa, MasterCard, and American Express). A different PaymentGroup represents each credit card. To split the item between three payment groups, PaymentGroupCommerceItemRelationships must be created between the CommerceItem (Car) and the three PaymentGroup objects:

- The first PaymentGroupCommerceItemRelationship connects the CommerceItem (car) and the first PaymentGroup (Visa). The relationship type is set to PaymentAmount, and the amount is set to $4000.
- The second PaymentGroupCommerceItemRelationship connects the CommerceItem (car) and the second PaymentGroup (MasterCard). The relationship type is set to PaymentAmount, and the amount is set to $4000.
- The third PaymentGroupCommerceItemRelationship connects the CommerceItem (car) and the third PaymentGroup (American Express). The relationship type is set to PaymentAmountRemaining.

PaymentGroupShippingGroupRelationship Object

A PaymentGroupShippingGroupRelationship represents a relationship between a ShippingGroup and a PaymentGroup. This type of Relationship is used to assign shipping costs in a ShippingGroup to PaymentGroups.

This relationship can be assigned the relationship types ShippingAmount or ShippingAmountRemaining. The following list describes the relationship types:

- ShippingAmount: This relationship type indicates that a specific amount of the shipping cost will be paid using the information in the PaymentGroup. The amount must be greater than zero. If the amount is greater than the total amount of the item stored in the ShippingGroup, then the cost of the entire item will be paid using the referenced PaymentGroup.
- ShippingAmountRemaining: This relationship type indicates that any remaining shipping cost amount unaccounted for by other PaymentGroupShippingGroupRelationship objects will be paid using the information in the PaymentGroup. If there is only one relationship between a given ShippingGroup and a PaymentGroup of type PaymentAmountRemaining, then the entire shipping cost will be paid using the information in the PaymentGroup.

  Note: A ShippingGroup can have only one Relationship of type PaymentAmountRemaining.

For example, an order is shipped to one location stored in a ShippingGroup. The shipping costs are $10. The customer wants to pay for the shipping costs with a credit card. The credit card information is stored in the only PaymentGroup. A PaymentGroupShippingGroupRelationship of type ShippingAmount is created, and the amount is set to $10.

Commerce Item Relationships

The `CommerceItemRelationship` interface, which is implemented by `ShippingGroupCommerceItemRelationship` and `PaymentGroupCommerceItemRelationship` objects as described previously, contains a set of numeric methods (`get/setQuantity()`, `get/setAmount()`, `get/setRange()`). Certain method calls are valid depending on the relationship type. If an invalid method is called, the returned value is undefined. The table below summarizes which methods are valid with which relationship types.

Relationship Type	Valid Methods
`ShippingQuantity`	`get/setQuantity(), get/setRange()`
`ShippingQuantityRemaining`	`get/setRange()`
`OrderAmount`	`get/setAmount()`
`OrderAmountRemaining`	N/A
`PaymentAmount`	`get/setAmount()`
`PaymentAmountRemaining`	N/A
`ShippingAmount`	`get/setAmount()`
`ShippingAmountRemaining`	N/A
`TaxAmount`	`get/setAmount()`
`TaxAmountRemaining`	N/A

Relationship Priority

The priority of a `Relationship` is important during order processing. A relationship's type determines its priority. If relationships are the same types, the priority is determined by the order in which the relationships were created.

When an `Order` is being processed or fulfilled, the system moves through the relationships of `CommerceItems`, `ShippingGroups`, and `PaymentGroups` in the following order:

1. Shipping Priority
   - `ShippingQuantity`
   - `ShippingQuantityRemaining`
2. Commerce Item Payment Priority
   - `PaymentAmount`
   - `PaymentAmountRemaining`
3. Shipping Cost Payment Priority:

- ShippingAmount
- ShippingAmountRemaining

4. Tax Cost Payment Priority:
   - TaxAmount
   - TaxAmountRemaining
5. Order Cost Payment Priority
   - OrderAmount
   - OrderAmountRemaining

Assigning Items to Shipping Groups

When an Order is first created, it has an empty ShippingGroup, which serves as the default ShippingGroup for the Order. The type of default ShippingGroup that is created is determined by the defaultShippingGroupType property of the OrderTools component. By default, this property is set to hardgoodShippingGroup.

When a CommerceItem is added to the Order, that item is automatically a part of the default ShippingGroup because it is assumed that when there is only one ShippingGroup, all items are a part of that group. However, once a second ShippingGroup is added to the Order, the existing CommerceItem and any new commerce items must be explicitly assigned to one of the two shipping groups; any items that were in the default ShippingGroup are no longer a part of any ShippingGroup.

Before the Order is checked out, all of the order's commerce items must be a part a ShippingGroup. This requirement is checked during the checkout process by the validateForCheckout pipeline, which is executed by the processOrder pipeline. Each processor in the validateForCheckout pipeline validates a different part of the Order as complete. The validateShippingGroupsForCheckout processor, in specific, validates the shipping groups in the Order. The shipping groups are considered complete if the following criteria are met:

1. None of the required fields (name, address, city, state, and postal code) are empty in any ShippingGroup.
2. All of the commerce items in the Order are assigned to a ShippingGroup. This requirement must be met according to the following rules:
   - If there is only one ShippingGroup in the Order and **no** relationships exist between that ShippingGroup and the commerce items in the Order, then the shipping of all commerce items in the Order implicitly is accounted for by that ShippingGroup.
   - If there is only one ShippingGroup in the Order and relationships exist between that ShippingGroup and the commerce items in the Order, or if there is more than one ShippingGroup in the Order, then **every** CommerceItem in the Order must have its shipping explicitly accounted for with one or more ShippingGroupCommerceItemRelationship objects, as follows:

     If a CommerceItem has a ShippingGroupCommerceItemRelationship of type ShippingQuantityRemaining, then the item's shipping is accounted for regardless of whether it has other shipping relationships. This is because a "remaining" relationship type covers any quantity of the CommerceItem that is not accounted for by other shipping relationships.

If a CommerceItem has one or more ShippingGroupCommerceItemRelationship objects of type ShippingQuantity, but no relationship of type ShippingQuantityRemaining, then the total quantity of the CommerceItem covered by the relationships must be equal to or greater than the quantity in the CommerceItem.

The range property in the ShippingGroupCommerceItemRelationship identifies which particular items out of the total quantity of a CommerceItem are associated with a given ShippingGroup.

Note: The priority of a Relationship in an Order, which is determined by the relationship's type, plays a role in when the Relationship is processed as the Order proceeds through the checkout process. For more on relationship types and priority, see the Relationship Priority (page 247) subsection of the Using Relationship Objects (page 243) section in the *Working With Purchase Process Objects* (page 223) chapter.

For more information on ShippingGroupCommerceItemRelationship objects, see the Relationship Types (page 243) section in this chapter. For more information on the processOrder and validateForCheckout pipelines, see the Checking Out an Order (page 294) section of the *Configuring Purchase Process Services* chapter.

Assigning Costs to Payment Groups

When an Order is first created, it has an empty PaymentGroup, which serves as the default PaymentGroup for the Order. The type of default PaymentGroup that is created is determined by the defaultPaymentGroupType property of the OrderTools component. By default, this property is set to CreditCard.

When a CommerceItem is added to the Order, that item is automatically a part of the default PaymentGroup because it is assumed that when there is only one PaymentGroup, all CommerceItem costs – along with all shipping and tax costs -- are a part of that group. However, once a second PaymentGroup is added to the Order, the existing CommerceItem, any new commerce items, and all shipping and tax costs must be explicitly assigned to one of the two payment groups; any items that were in the default PaymentGroup are no longer a part of any PaymentGroup.

Before the Order is checked out, all of the order's costs must be a part of a PaymentGroup. This requirement is checked during the checkout process by the validateForCheckout pipeline, which is executed by the processOrder pipeline. Each processor in the validateForCheckout pipeline validates a different part of the Order as complete. The payment groups in the Order are considered complete if the following criteria are met:

1. None of the required fields (name, address, city, state, and postal code) are empty in any PaymentGroup.
2. All of the costs associated with the Order, which include the commerce item costs, shipping costs, and tax, are accounted for by one or more payment groups.

If the Order has only one payment group, then the assignment of Order costs is automatic. However, if the Order has more than one payment group, you must explicitly assign the Order costs to the payment groups. You can do so using one of following methods:

- Assign the **total** cost of the Order to one or more payment groups. This approach is more frequently used, and it is the most straightforward way to account for an order's payment because you are dealing with the Order costs as a whole. See Assigning an Order's Total Cost to Payment Groups (page 250).

- Assign the component costs of the `Order` -- the commerce item costs, shipping costs, and tax -- to one or more payment groups. You should use this method if you need more control over where to assign the component costs of an `Order`. See Assigning an Order's Component Costs to Payment Groups (page 251).

For more information on the `processOrder` and `validateForCheckout` pipelines, see the Checking Out an Order (page 294) section of the *Configuring Purchase Process Services* chapter. For information on the `PaymentGroupCommerceItemRelationship` and `PaymentGroupShippingGroupRelationship` objects that are described in the following subsections, see the Relationship Types (page 243) section in this chapter.

Note: The priority of a `Relationship` in an `Order`, which is determined by the relationship's type, plays a role in when the `Relationship` is processed as the `Order` proceeds through the checkout process. For more on relationship types and priority, see the Relationship Priority (page 247) subsection of the Using Relationship Objects (page 243) section in the *Working With Purchase Process Objects* (page 223) chapter.

Assigning an Order's Total Cost to Payment Groups

To assign the **total** cost of the `Order` to one or more payment groups, use the `addOrderAmountToPaymentGroup()` and `addRemainingOrderAmountToPaymentGroup()` methods in the `OrderManager`. These methods add `PaymentGroupOrderRelationship` objects (of type `OrderAmount` or `OrderAmountRemaining`, respectively) to the `Order`.

Example 1

This example assigns an order's total cost to a single `PaymentGroup`. The order's total cost is $20.90. You can account for the total cost of the `Order` by calling:

- `addRemainingOrderAmountToPaymentGroup()` and passing in the required parameters.

  -- or --

- `addOrderAmountToPaymentGroup()` and passing in the value 20.90 for the amount parameter.

The disadvantage to calling `addOrderAmountToPaymentGroup()` instead of `addRemainingOrderAmountToPaymentGroup()` is that if the order's total amount increases above $20.90, then you must call `removeOrderAmountFromPaymentGroup()` and then call `addOrderAmountToPaymentGroup()` again and pass in the new amount.

Example 2

This example assigns an order's total cost to more than one `PaymentGroup`. The order's total cost is $20.90, and there are two payment groups in the `Order`. You want to assign $10.00 to the first `PaymentGroup` and $10.90 to the second `PaymentGroup`. Follow these steps to assign the amounts to the different payment groups:

1. Call `addOrderAmountToPaymentGroup()` and pass in the first `PaymentGroup` and 10.00 for the amount parameter.

2. Call `addRemainingOrderAmountToPaymentGroup()`, or call `addOrderAmountToPaymentGroup()` and pass in the second `PaymentGroup` and 10.90 for the amount parameter.

Accounting for an Order's Total Cost

If the total cost of an `Order` is assigned to one or more payment groups, then during checkout the order's total cost is accounted for according to the following rules:

- If the `Order` contains a `PaymentGroupOrderRelationship` of type `OrderAmountRemaining`, then the order's total cost is accounted for regardless of whether or not the `Order` has other payment relationships.

This is because a "remaining" relationship covers everything not accounted for by other payment relationships.

- If an `Order` contains one or more `PaymentGroupOrderRelationship` objects of type `OrderAmount`, then the sum of the amounts in the relationships must be equal to or greater than the total amount of the `Order` to account for the order's total cost. (The total amount of an `Order` is the sum of its `CommerceItem` costs, `ShippingGroup` costs, and tax.)

Assigning an Order's Component Costs to Payment Groups

An order's component costs include its commerce item costs, shipping costs, and tax. You can assign these various costs to payment groups using the following methods:

- `CommerceItemManager.addItemAmountToPaymentGroup()`
- `CommerceItemManager.addRemainingItemAmountToPaymentGroup()`
- `ShippingGroupManager.addShippingCostAmountToPaymentGroup()`
- `ShippingGroupManager.addRemainingShippingCostToPaymentGroup()`
- `OrderManager.addTaxAmountToPaymentGroup()`
- `OrderManager.addRemainingTaxAmountToPaymentGroup()`

For example, suppose an `Order` has two commerce items, two payment groups, and one shipping group. The cost of the first item is $5.99, and the cost of the second item is $9.99. Both items are in the shipping group, and the shipping cost is $5.00. The tax for the order is $0.80. You might account for the order's costs by doing the following:

1. Call `addItemAmountToPaymentGroup()` for the first item and pass 5.99 for the amount parameter to the first payment group.
2. Call `addItemAmountToPaymentGroup()` for the second item and pass 9.99 for the amount parameter to the second payment group.
3. Call `addShippingCostAmountToPaymentGroup()` for the shipping group and pass 5.00 for the amount parameter to the first payment group.
4. Call `addTaxAmountToPaymentGroup()` for the order's tax and pass 0.80 for the amount parameter to the second payment group.

   The order's costs are now accounted for because the commerce item costs, shipping costs, and tax have all been assigned to a payment group.

Accounting for Commerce Item Costs

If the `CommerceItem` costs of an `Order` are assigned to one or more payment groups, then during checkout the order's `CommerceItem` costs are accounted for according to the following rules:

- If there is only one `PaymentGroup` in the `Order` and **no** relationships exist between that `PaymentGroup` and the commerce items in the `Order`, then the cost of all commerce items in the `Order` implicitly is accounted for by that `PaymentGroup`.
- If there is only one `PaymentGroup` in the `Order` and relationships exist between that `PaymentGroup` and the commerce items in the `Order`, or if there is more than one `PaymentGroup` in the `Order`, then **every** `CommerceItem` in the `Order` must have its costs explicitly accounted for with one or more `PaymentGroupCommerceItemRelationship` objects, as follows:

If a `CommerceItem` has a `PaymentGroupCommerceItemRelationship` of type `PaymentAmountRemaining`, then that relationship accounts for the cost of the `CommerceItem` regardless of whether the `CommerceItem` has other payment relationships. This is because a "remaining" relationship covers everything not accounted for by other payment relationships.

If a `CommerceItem` has one or more `PaymentGroupCommerceItemRelationship` objects of type `PaymentAmount`, but no `PaymentGroupCommerceItemRelationship` of type `PaymentAmountRemaining`, then the total cost covered by all of the relationships must be equal to or greater than the total cost of the `CommerceItem`.

Accounting for Shipping Costs

If the shipping costs of an `Order` are assigned to one or more payment groups, then during checkout the order's shipping costs are accounted for according to the following rules:

- If there is only one `PaymentGroup` in the `Order` and **no** relationships exist between that `PaymentGroup` and the shipping groups in the `Order`, then the shipping costs for the `Order` implicitly are accounted for by that `PaymentGroup`.
- If there is only one `PaymentGroup` in the `Order` and relationships exist between that `PaymentGroup` and the shipping groups in the `Order`, or if there is more than one `PaymentGroup` in the `Order`, then **every** `ShippingGroup` in the `Order` must have its costs explicitly accounted for by one or more `PaymentGroupShippingGroupRelationship` objects, as follows:

  If a `ShippingGroup` has a `PaymentGroupShippingGroupRelationship` of type `ShippingAmountRemaining`, then that relationship accounts for the cost of the `ShippingGroup` regardless of whether the `ShippingGroup` has other payment relationships. This is because a "remaining" relationship covers everything not accounted for by other payment relationships.

  If a `ShippingGroup` has one or more `PaymentGroupShippingGroupRelationship` objects of type `ShippingAmount`, but no `PaymentGroupShippingGroupRelationship` of type `ShippingAmountRemaining`, then the total cost covered by all of the relationships must be equal to or greater than the total cost of the `ShippingGroup`.

Accounting for Tax

If the tax of an `Order` is assigned to one or more payment groups, then during checkout the order's tax cost is accounted for according to the following rules:

- If the `Order` contains a `PaymentGroupOrderRelationship` of type `TaxAmountRemaining`, then the order's tax cost is accounted for regardless of whether or not the `Order` has other payment relationships. This is because a "remaining" relationship covers everything not accounted for by other payment relationships.
- If the `Order` contains one or more `PaymentGroupOrderRelationship` objects of type `TaxAmount`, then the sum of the amounts in the relationships must be equal to or greater than the total tax amount of the `Order` to account for tax payment.

Setting Handling Instructions

Handling instructions are specific instructions that can be associated with the commerce items in a shipping group. For example, a customer could set handling instructions for gift wrapping.

HandlingInstruction Objects

`HandlingInstruction` objects are constructed using one of the `createHandlingInstruction()` methods in the `HandlingInstructionManager`. After the object is created and populated with the required data, call the `addHandlingInstructionToShippingGroup()` method to add the handling instructions to the `ShippingGroup` that contains the commerce item(s) whose IDs are specified in the `HandlingInstruction`.

A customer can set more than one handling instruction for a given `CommerceItem` in a `ShippingGroup`. For example, if an `Order` contains a single `CommerceItem` that was added to a user's shopping cart from another user's gift list and needs to be gift wrapped, two `HandlingInstructions` must be created – one for the gift wrapping and another for the gift list. Each `HandlingInstruction` would be of a different class type: `GiftwrapHandlingInstruction` and `GiftlistHandlingInstruction`, respectively. Each `HandlingInstruction` would contain:

- The ID of the `ShippingGroup` that contains the `CommerceItem`.
- The ID of the `CommerceItem` to which the handling instruction applies.
- A quantity of 1.
- Any other necessary fields of `GiftwrapHandlingInstruction` and `GiftlistHandlingInstruction`.

The sum of `HandlingInstruction` quantities cannot exceed the quantity in the `ShippingGroup` for a particular `HandlingInstruction` class. In the above example, there can be only one `GiftwrapHandlingInstruction` with a quantity of 1 in the `ShippingGroup` because there is only one item in the `ShippingGroup`. However, there can be a `GiftlistHandlingInstruction` in the `ShippingGroup` that references the same item because the two `HandlingInstruction` objects are different types.

For details on `GiftListHandlingInstructions`, see the *Configuring Commerce Services* (page 71) chapter.

Adding Handling Instructions to a Shipping Group

Handling instructions can be added to either shipping groups or individual commerce items. This section provides an example of how to create handling instructions for gift wrapping one item in a shipping group. The example also uses a repository with a wrapping paper item descriptor.

For additional, detailed examples of how to extend the purchase process to support new commerce objects, see the Extending the Purchase Process (page 341) section in this guide. For information on how to create new item descriptors in a repository, see the *SQL Repository Data Models* chapter in the *ATG Repository Guide*. For more information on `HandlingInstruction`, see the *ATG Platform API Reference*.

Step 1: Create a `GiftwrapHandlingInstruction` class that extends the `atg.commerce.order.HandlingInstructionImpl` class. Add a property to identify the wrapping paper. For example:

```
public String getWrappingPaperId();
public void setWrappingPaperId(String pWrappingPaperId);
```

Step 2: Add support for this class to the `OrderTools` component. For example, in the `localconfig` directory add the following lines to `/atg/commerce/order/OrderTools.properties`:

```
handlingTypeClassMap+=giftwrapHandlingInstruction=
mypackage.GiftwrapHandlingInstruction
beanNameToItemDescriptorMap+=mypackage.GiftwrapHandlingInstruction=
```

```
giftwrapHandlingInstruction
```

Step 3: Add support for the new item descriptor to `orderrepository.xml`. The `orderrepository.xml` file is located in `/atg/commerce/order/orderrepository.xml` in the `configpath`.

First, add the new type to the `handlingInstruction` item descriptor:

```
<table name="dcspp_hand_inst" type="primary"
       id-column-name="handling_inst_id">
  <property name="type" data-type="enumerated"
            default="handlingInstruction" expert="true"
            display-name="Type">
    <attribute name="useCodeForValue" value="false"/>
      <option value="handlingInstruction" code="0"/>
      <option value="giftlistHandlingInstruction" code="1"/>
  </property>
     . . .
</table>
```

Second, add the new item descriptor:

```
<item-descriptor
      name="giftwrapHandlingInstruction"
      super-type="handlingInstruction"
      sub-type-value="giftwrapHandlingInstruction"
      cache-mode="locked"
      display-name="Gift wrap Handling Instruction">
  <attribute name="writeLocksOnly" value="true"/>
  <table name="dcspp_wrap_inst"
         id-column-name="handling_inst_id">
    <property name="wrappingPaperId"
              column-name="wrap_id"
              data-type="string"
              display-name="Wrapping Paper Id"/>
  </table>
</item-descriptor>
```

Step 4: Create a method that adds the handling instruction to the order in a class that extends the `HandlingInstructionManager`:

```
public wrapCommerceItemInShippingGroup(ShippingGroup
  pGiftShippingGroup, CommerceItem pWrappedItem,String pWrappingPaperId)
  {
     GiftwrapHandlingInstruction giftwrap = (GiftwrapHandlingInstruction)
     handlingInstructionManager.createHandlingInstruction(
     "giftwrapHandlingInstruction",
     pGiftShippingGroup.getId(), pWrappedItem.getId(),
                                 pWrappedItem.getQuantity());
     giftwrap.setWrappingPaperId(wrappingId);
     pGiftShippingGroup.addHandlingInstruction(giftwrap);
  }
```

Step 5: Add any additional processors. For example, you could add a processor to the `validateForCheckout` pipeline to validate the wrapping paper type.

Oracle ATG Web Commerce States

In Oracle ATG Web Commerce, the subclasses of `atg.commerce.states.ObjectStates` represent the possible states for the order objects. For example, the `atg.commerce.states.CommerceItemStates` class represents the possible states of a `CommerceItem`, the `atg.commerce.states.PaymentGroupStates` class represents the possible states of a `PaymentGroup`, and so on. The state names are defined in static `String` variables in each state class, and Commerce code uses the state name to set the state of a given object. For example:

```
// set the state of shippingGroup to the integer value of the
// PENDING_SHIPMENT state. sgStates.PENDING_SHIPMENT is the
// name of the state
ShippingGroupStates sgStates = getShippingGroupStates();
shippingGroup.setState(sgStates.getStateValue(sgStates.PENDING_SHIPMENT));
```

Commerce provides the following configured instances of the state classes, which are located in Nucleus at `/atg/commerce/states/`:

- `OrderStates`: indicates the states of an `Order`.
- `CommerceItemStates`: indicates the states of a `CommerceItem`.
- `PaymentGroupStates`: indicates the states of a `PaymentGroup`.
- `ShippingGroupStates`: indicates the states of `ShippingGroup`.
- `ShipItemRelationshipStates`: indicates the states of a `ShippingGroupCommerceItemRelationship`.

The properties files of these state objects configure the following properties, which provide mappings of the order object's state names to corresponding `String` values (for easy reading) and integer values (for easy comparisons):

- `stateValueMap`: maps each state name to an Integer value.
- `stateStringMap`: maps each state's Integer value to a display name. This is the `String` value that users see and that is stored in the repository.
- `stateDescriptionMap`: maps each state's Integer value to a `String` description of the state.

Note that a state's name and display name are two different values.

Each state class (`OrderStates`, `CommereItemStates`, and so on) contains several methods for retrieving a requested state's Integer value, display name, or description from the state mappings. The following table describes these methods:

Method name	Description
`getStateValue`	Given a state name, this method returns its Integer value.
`getStateFromString`	Given a state's display name, this method returns its Integer value.

Method name	Description
`getStateString`	Given a state's Integer value, this method returns its display name.
`getStateDescription`	Given a state's Integer value or display name, this method returns its description.
`getStateAsUserResource`	Given a state's Integer value or display name (as configured in the properties file of the relevant states component, for example, `OrderStates.properties` or `CommerceItemStates.properties`), this method returns the display name that is defined in the resource file that is appropriate for the current locale. **Note:** Used for internationalization, as described below.
`getStateDescriptionAsUserResource`	Given a state's Integer value or display name (as configured in the properties file of the relevant states component, for example, `OrderStates.properties` or `CommerceItemStates.properties`), this method returns the state's description that is defined in the resource file that is appropriate for the current locale. **Note:** Used for internationalization, as described below.

The methods in the preceding table are called by several methods in the implementation classes of the order objects. Consequently, to retrieve the Integer value for the state of an order object, you can simply call the `getState` method of that order object. For example, call `OrderImpl.getState` to retrieve the corresponding Integer value for the state of the `Order` from `OrderStates.properties`. Similarly, call `CommerceItemImpl.getState` to retrieve the corresponding Integer value for the state of the `CommerceItem` from `CommerceItemStates.properties`.

Additionally, if your commerce site **isn't** internationalized, use the following order object methods to retrieve the display name or description, respectively, for the state of the order object. Like the `getState` method, these methods retrieve a value specified in the properties file of the relevant states component:

- `getStateAsString`
  For example, call `OrderImpl.getStateAsString` to retrieve the display name for the state of the `Order` object from `OrderStates.properties`. Similarly, call `CommerceItemImpl.getStateAsString` to retrieve the display name for the state of the `CommerceItem` from `CommerceItemStates.properties`.

- `getStateDetail`
  For example, call `OrderImpl.getStateDetail` to retrieve the description for the state of the `Order` object from `OrderStates.properties`. Similarly, call `CommerceItemImpl.getStateDetail` to retrieve the description for the state of the `CommerceItem` from `CommerceItemStates.properties`.

If your commerce site **is** internationalized, and you, therefore, need to provide one or more translations of the display names and descriptions for the states of the Oracle ATG Web Commerce order objects, then you should

specify those values in `ResourceBundle.properties` files that are placed in the CLASSPATH. By default, Commerce provides a base `ResourceBundle.properties` file named `StateResources.properties`, which is used when a locale-specific resource file isn't found during a request. The `StateResources.properties` file maps each state's configured display name to a translated display name and description using the following key-value format:

- For the display names of states:

```
ORDER.INCOMPLETE=INCOMPLETE
ORDER.SUBMITTED=SUBMITTED
SHIPPING.INITIAL=INITIAL
SHIPPING.PROCESSING=PROCESSING
ITEM.INITIAL=INITIAL
ITEM.PENDING_REMOVE=PENDING_REMOVE
```

- For the descriptions of states:

```
ORDERDESC.INCOMPLETE=The order is incomplete
ORDERDESC.SUBMITTED=The order has been submitted to Fulfillment
SHIPPINGDESC.INITIAL=The shipping group has been initialized
SHIPPINGDESC.FAILED=The shipping group has failed
ITEMDESC.INITIAL=The item has been initialized
ITEMDESC.PENDING_REMOVE=The item is pending remove request
```

Note that the keys are the display names of the states as configured in the properties files of the states components (`OrderStates.properties`, `ShippingStates.properties`, and so on). Also note that, because different order objects may use the same display name for a given state, each key is prepended with a prefix to avoid conflict.

To add an additional resource file, copy `StateResource.properties` and rename the file according to Java naming guidelines for `ResourceBundle` inheritance, using any appropriate language, country, and variant suffixes. Then translate the file according to the translation guidelines provided in the *Internationalizing a Dynamo Web Site* chapter in the *ATG Platform Programming Guide*. (You can refer to that chapter for more information on working with `ResourceBundles`.) Finally, place the resource file in the CLASSPATH. By default, `StateResources.properties` is located in the CLASSPATH at `atg.commerce.states.StateResources.properties`, and each states component refers to this file in its `resourceFileName` property (`OrderStates.resourceFileName`, `CommerceItemStates.resourceFileName`, and so on).

If your commerce site is internationalized and, therefore, makes use of `StateResources.properties` resource files, use the following order object methods to retrieve a locale-specific display name or description, respectively, for the state of the order object:

- `getStateAsUserResource`
  For example, call `OrderImpl.getStateAsUserResource` to retrieve a locale-specific display name for the state of the `Order` object from the appropriate `StateResources_XX.properties` file.

- `getStateDescriptionAsUserResource`
  For example, call `OrderImpl.getStateDescriptionAsUserResource` to retrieve a locale-specific description for the state of the `Order` object from the appropriate `StateResources_XX.properties` file.

Like the other order object methods, these methods call through to the methods in the states classes (`OrderStates`, `CommerceItemStates`, and so on), namely the `getStateAsUserResource` and `getStateDescriptionAsUserResource` methods. Given the current locale and the basename specified in the `resourceFileName` property of the states component (`OrderStates.resourceFileName`, `CommerceItemStates.resourceFileName`, and so on), these latter methods use `ResourceBundle`

inheritance rules to retrieve the most appropriate resource file for the current locale. For example, if the locale is `fr_FR`, the code first looks for a `StateResources_fr_FR.properties` file. If the file doesn't exist, it then looks for a `StateResources_fr.properties` file. If that file doesn't exist, it retrieves the default resource file, `StateResources.properties`. Once the appropriate resource file is obtained, the appropriate prefix is appended to the key, and the requested value is retrieved.

As previously mentioned , in Oracle ATG Web Commerce the subclasses of `atg.commerce.states.ObjectStates` represent the possible states for the order objects. You can refer to the subsections that follow for descriptions of these states:

- CommerceItem States (page 258)
- Order States (page 258)
- PaymentGroup States (page 259)
- ShippingGroup States (page 260)
- ShippingGroupCommerceItemRelationship Object (page 244)

CommerceItem States

The following table describes the possible states of a `CommerceItem`:

State Name	Description
BACK_ORDERED	The item isn't available in the inventory; it has been backordered.
DISCONTINUED	The item isn't available in the inventory; it cannot be backordered.
FAILED	The item has failed.
INITIAL	The item is in an initial state, that is, it is not yet associated with any shipping group.
ITEM_NOT_FOUND	The item could not be found in the inventory.
OUT_OF_STOCK	The item isn't available in the inventory, and it has not been backordered.
PENDING_REMOVE	The item will be removed pending verification that all item relationships referring to it can be removed.
PRE_ORDERED	The item isn't available in the inventory; it has been preordered.
REMOVED	The item has been removed from the order.
SUBITEM_PENDING_DELIVERY	The item is available in the inventory, and it is being prepared for shipment to the customer.

Order States

The following table describes the possible states of an `Order`:

State Name	Description
APPROVED	The approval process for the order is complete, and the order has been approved.
FAILED	The order failed.
FAILED_APPROVAL	The approval process for the order is complete, and the order has been rejected.
INCOMPLETE	The order is in the purchase process.
NO_PENDING_ACTION	The order has been fulfilled, and processing of the order is complete. All shipping groups in the order are in a NO_PENDING_ACTION or REMOVED state, and order payment has been settled.
PENDING_APPROVAL	The order requires approval (or an additional approval) by an authorized approver.
PENDING_CUSTOMER_ACTION	Processing of the order requires the customer's attention for some reason, such as an incorrect customer address.
PENDING_CUSTOMER_RETURN	This is an unused state. It is placed in the list of states for the convenience of those who might want to implement this state.
PENDING_MERCHANT_ACTION	Processing of the order requires merchant attention for some reason, such as the failure of a payment group in the order.
PENDING_REMOVE	A request was made to remove the order. The order is placed in this state until all shipping groups in the order are set to a PENDING_REMOVE state.
PROCESSING	The order is being processed by Fulfillment.
QUOTED	This is an unused state. It is placed in the list of states for the convenience of those who might want to implement this state.
REMOVED	The order has been removed successfully.
SUBMITTED	The order has completed the purchase process and has been submitted to Fulfillment.
TEMPLATE	The order is a template order used by a scheduled order.

PaymentGroup States

The following table describes the possible states of a `PaymentGroup`:

State Name	Description
AUTHORIZE_FAILED	Authorization of the payment group has failed.
AUTHORIZED	The payment group has been authorized and can be debited.
CREDIT_FAILED	Credit of the payment group has failed.
INITIAL	The payment group hasn't been acted on yet.
REMOVED	The payment group has been removed.
SETTLE_FAILED	Debit of the payment group has failed.
SETTLED	The payment group has been debited successfully.

ShippingGroup States

The following table describes the possible states of a `ShippingGroup`:

State Name	Description
INITIAL	The shipping group is in a pre-fulfillment state.
PROCESSING	The shipping group has started the fulfillment process.
PENDING_REMOVE	A request for the removal of the entire order was made, and the removal of this shipping group is possible.
REMOVED	The shipping group has been removed.
FAILED	The shipping group has failed to process.
PENDING_SHIPMENT	The shipping group awaits shipment. This is used by Oracle ATG Web Commerce to determine which shipping groups are ready to be shipped.
NO_PENDING_ACTION	The shipment of all the items in the shipping group is complete.
PENDING_MERCHANT_ACTION	An error occurred while trying to process the shipping group; the error requires the merchant's attention.

ShippingGroupCommerceItemRelationship States

The following table describes the possible states of a `ShippingGroupCommerceItemRelationship`:

State Name	Description
BACK_ORDERED	The item isn't available in the inventory; it has been backordered.
DELIVERED	The item has been delivered. This state occurs when the shipping group containing this relationship has shipped.
DISCONTINUED	The item isn't available in the inventory; it cannot be backordered.
FAILED	The item relationship failed.
INITIAL	The order fulfillment framework has not acted on the item relationship.
ITEM_NOT_FOUND	The item could not be found in the inventory.
OUT_OF_STOCK	The item isn't available in the inventory, and it has not been backordered.
PENDING_DELIVERY	The item has been allocated in the inventory system and is ready to be delivered.
PENDING_REMOVE	A request to remove the shipping group was made, and the items have not yet shipped.
PENDING_RETURN	This is an unused state. It is placed in the list of states for the convenience of those who might want to implement this state.
PENDING_SUBITEM_DELIVERY	The item is available in the inventory, and it is being prepared for shipment to the customer.
PRE_ORDERED	The item isn't available in the inventory; it has been preordered.
REMOVED	The item relationship has been removed.
RETURNED	This is an unused state. It is placed in the list of states for the convenience of those who might want to implement this state.

15 Configuring Purchase Process Services

Oracle ATG Web Commerce enables you to build sites that support simple or complex purchasing processes. A simple purchase process might provide customers with a single shopping cart, and enable customers to purchase products using a single payment method and to ship those products to a single location. In contrast, a more complex purchase process might include multiple shopping carts, payment methods, and shipping locations. You can use Commerce to customize a purchase process that fills all the requirements of your sites.

This chapter includes information on the following purchase process services:

Describes how to set restrictions on placing `Orders`.

Tracking the Shopping Process (page 328)
Describes how to track stages an `Order` goes through in the purchase process.

Troubleshooting Order Problems (page 330)
Provides important information if you have modified the `OrderManager` and are now experiencing problems with orders.

Handling Returned Items (page 331)
Describes how the purchase process handles returned items.

Extending the Oracle ATG Web Commerce Form Handlers (page 333)
Describes how the Oracle ATG Web Commerce form handlers manage transactions. Also provides information to assist you when extending them.

For detailed information on the various classes and interfaces used in theCommerce purchase process, see the *Working With Purchase Process Objects* (page 223) chapter.

Loading Orders

The actual loading of an `Order` object occurs by calling the `loadOrder()` method in the `OrderManager`. The `loadOrder()` method calls into the `PipelineManager` to execute the `loadOrder` pipeline, which creates and populates the `Order` object.

The following table describes the individual processors in the `loadOrder` pipeline. They are listed in order of execution.

PipelineLink name	Description
`loadOrderObject`	Given an `Order` ID supplied by the `PipelineManager`, this processor creates an `Order` object and loads its properties from the Order Repository. Note that while the `Order` object is loaded, none of the contained objects, such as the `CommerceItems` or `ShippingGroups`, are loaded. Later, when an `Order` property is accessed (for example, by calling a method like `getCommerceItems()` or `getShippingGroups()` in the `Order`), the rest of the objects in the `Order` are loaded. See Refreshing Orders (page 265) below for details. The `atg.commerce.order.processor.ProcLoadOrderObject` class implements this functionality.

PipelineLink name	Description
loadPriceInfoObjects	Creates OrderPriceInfo and TaxPriceInfo objects for the given Order and loads their properties from the Order Repository. Note that only the OrderPriceInfo and TaxPriceInfo objects are loaded at this point. Later, when an Order property is accessed (for example, by calling getPriceInfo() or getCommerceItems() in the Order), the rest of the AmountInfo objects in the Order are loaded, such as the ItemPriceInfo objects in the CommerceItems and the ShippingPriceInfo objects in the ShippingGroups. See Refreshing Orders (page 265) below for details. The atg.commerce.order.processor.ProcLoadPriceInfoObjects class implements this functionality. For more information about PriceInfo objects, see the *Commerce Pricing Engines* (page 127) chapter.

For more information about the OrderManager, see the *Working With Purchase Process Objects* (page 223) chapter. For more information about pipelines, the PipelineManager, and the transactional modes and transitions of the processors in the loadOrder pipeline, see Appendix F, *Pipeline Chains* (page 699).

Refreshing Orders

In Oracle ATG Web Commerce, the purchase process controls the refreshing of Orders.

When an Order is loaded from the Order Repository, the loadOrder() method in the OrderManager calls into the PipelineManager to execute the loadOrder pipeline, which creates and loads the Order object, as well as its OrderPriceInfo and TaxPriceInfo objects. (See Loading Orders (page 264) above for details.) Later, when an Order property is accessed (for example, by calling getCommerceItems() or getPriceInfo()), the refreshOrder pipeline is invoked, which creates and loads the rest of the contained objects in the Order.

The refreshOrder pipeline is called only when the Order is first accessed and subsequently when an Order is invalidated and, therefore, needs to be reloaded from the Order Repository.

The following table describes the individual processors in the refreshOrder pipeline. They are listed in order of execution.

PipelineLink name	Description
loadOrderObjectForRefresh	Given an Order object supplied by the PipelineManager, this processor reloads its properties from the Order Repository. The atg.commerce.order.processor.ProcLoadOrderObject class implements this functionality.
loadCommerceItemObjects	Creates CommerceItem objects for the Order and loads the properties for those CommerceItem objects from the Order Repository. The atg.commerce.order.processor.ProcLoadCommerceItemObjects class implements this functionality.

PipelineLink name	Description
`loadShippingGroupObjects`	Creates `ShippingGroup` objects for the `Order` and loads the properties for those `ShippingGroup` objects from the Order Repository. The `atg.commerce.order.processor.ProcLoadShippingGroupObjects` class implements this functionality.
`loadHandlingInstructionObjects`	Creates `HandlingInstruction` objects for the `ShippingGroups` in the `Order` and loads the properties for those `HandlingInstruction` objects from the Order Repository. The `atg.commerce.order.processor.ProcLoadHandlingInstructionObjects` class implements this functionality.
`loadPaymentGroupObjects`	Creates `PaymentGroup` objects for the `Order` and loads the properties for those `PaymentGroup` objects from the Order Repository. The `atg.commerce.order.processor.ProcLoadPaymentGroupObjects` class implements this functionality.
`loadPaymentStatusObjects`	Creates `PaymentStatus` objects for all the payment groups in the `Order` and loads the properties for those `PaymentStatus` objects from the Order Repository. The `atg.commerce.order.processor.ProcLoadPayment StatusObjects` class implements this functionality.
`loadRelationshipObjects`	Creates `Relationship` objects for the `Order` and loads the properties for those `Relationship` objects from the Order Repository. The `atg.commerce.order.processor.ProcLoadRelationshipObjects` class implements this functionality.
`loadPriceInfoObjects`	Reloads the `OrderPriceInfo` and `TaxPriceInfo` objects in the given `Order`. Also creates the rest of the `AmountInfo` objects for the `Order`, such as the `ItemPriceInfo` objects in the `CommerceItems` and the `ShippingPriceInfo` objects in the `ShippingGroups`, and loads their properties from the Order Repository. For more information about `AmountInfo` objects, see the *Commerce Pricing Engines* (page 127) chapter. The `atg.commerce.order.processor.ProcLoadPriceInfoObjects` class implements this functionality.

PipelineLink name	Description
`setCatalogRefs`	Sets the `catalogRef` property in the `AuxiliaryData` object of each `CommerceItem` in the `Order`. This processor looks up the catalog reference in the Catalog Repository using the `catalogRefId` in the `CommerceItem`. Note that, if `SetCatalogRefs.substituteRemovedSku` is true, this processor replaces all deleted SKUs in the `Order` with the "dummy" SKU defined by `SetCatalogRefs.substituteDeletedSkuId`. For more information, see Managing Orders that Contain Deleted Products and SKUs (page 268) below. The `atg.commerce.order.processor.ProcSetCatalogRefs` class implements this functionality.
`setProductRefs`	Sets the `productRef` property in the `AuxiliaryData` object of each `CommerceItem` in the `Order`. This processor looks up the catalog reference in the Catalog Repository using the `productRefId` in the `AuxiliaryData` object. Note that, if `SetProductRefs.substituteRemovedProduct` is true, this processor replaces all deleted products in the `Order` with the "dummy" product defined by `SetProductRefs.substituteDeletedProductId`. For more information, see Managing Orders that Contain Deleted Products and SKUs (page 268) below. The `atg.commerce.order.processor.ProcSetProductRefs` class implements this functionality.
`removeExpiredCommerceItems`	Used in conjunction with `SetCatalogRefs` and `SetProductRefs`. If the state of the `Order` is one that is defined in `RemoveExpiredCommerceItems.openOrderStates`, this processor removes from the `Order` any `CommerceItem` that contains a "dummy" SKU or product that was substituted by `SetCatalogRefs` or `SetProductRefs`. A "dummy" SKU is automatically removed. A "dummy" product is removed only if `RemoveExpiredCommerceItems.removeItemsWithDeletedProducts` is set to true; the default is true. For more information, see Managing Orders that Contain Deleted Products and SKUs (page 268) below. The `atg.commerce.order.processor.ProcRemoveExpiredCommerceItems` class implements this functionality.

For more information about pipelines, the `PipelineManager`, and the transactional modes and transitions of the processors in the `refreshOrder` pipeline, see Appendix F, *Pipeline Chains* (page 699).

Managing Orders that Contain Deleted Products and SKUs

As described in the previous section, Refreshing Orders (page 265), the last three processors in the `refreshOrder` pipeline can be used to operate on the commerce items in an order that refer to products and/or SKUs that have been deleted from the catalog. If your catalog administrators delete products and/or SKUs in the ongoing management of the product catalog, you should configure these processors to handle affected orders appropriately.

To configure the `refreshOrder` pipeline to manage deleted SKUs, do the following:

1. Create a new SKU in the product catalog. In an `Order`, this "dummy" SKU will be substituted for any SKU that has been deleted from the catalog.

2. Set the `SetCatalogRefs.substituteDeletedSkuId` property to the ID of the dummy SKU you created in step 1.

3. Set the `SetCatalogRefs.substituteRemovedSku` property to true. If this property is true, the processor replaces any deleted SKU found in the `Order` with the SKU defined in the `substituteDeletedSkuId` property.

4. If the `Order` is in an open state, all dummy SKUs in the `Order` are removed automatically by a later processor in the `refreshOrder` pipeline, `RemoveExpiredCommerceItems`. See `RemoveExpiredCommerceItems` in the table above for more information.

To configure the `refreshOrder` pipeline to manage deleted products, do the following:

1. Create a new product in the product catalog. In an `Order`, this "dummy" product will be substituted for any product that has been deleted from the catalog.

2. Set the `SetCatalogRefs.substituteDeletedProductId` property to the ID of the dummy product you created in step 1.

3. Set the `SetProductRefs.substituteRemovedProduct` property to true. If this property is true, the processor replaces any deleted product found in the `Order` with the product defined in the `substituteDeletedProductId` property.

4. If you want all dummy products in an `Order` (in an open state) to be removed later on in the `refreshOrder` pipeline by the `RemoveExpiredCommerceItems` processor, set its `removeItemsWithDeletedProducts` property to true. (See `RemoveExpiredCommerceItems` in the table above for more information.)

It's important to note that deleting products and SKUs is **not** recommended because of its impact on customers' order histories. Site pages that render order histories typically draw order information (descriptions, media, and so on) from product and SKU repository items. If those items are deleted, order histories cannot be rendered accurately.

If you need to remove products and SKUs from your database (for example, because of a high turnover rate), you should implement a strategy that addresses its impact on order histories. Possible strategies include:

- Storing the relevant description information in the `CommerceItem`. Note that this will cause a significant duplication of information across multiple items.

- Removing all historical orders that contain products or SKUs that have been removed.

- Keeping all historical orders that contain products or SKUs that have been removed. These orders will display description information for the "dummy" SKUs and products instead of for the actual items that were purchased.

Modifying Orders

You can modify an `Order` by adding items to it, removing items from it, and changing the quantities of the items in the `Order`. `CartModifierFormHandler` is provided to support these modification processes.

This section includes information about the following:

- Understanding the CartModifierFormHandler (page 269)
- Modifying the Current Order (page 274)

Understanding the CartModifierFormHandler

The `CartModifierFormHandler` is used to add items to and remove items from an `Order`, modify the quantity of items in the `Order`, and prepare the `Order` for the checkout process.

`CartModifierFormHandler` is an instance of class `atg.commerce.order.purchase.CartModifierFormHandler`; it is located in Nucleus at `/atg/commerce/order/purchase/CartModifierFormHandler`.

Many of the methods in `CartModifierFormHandler` call `OrderManager.updateOrder()` to save the `Order` in its present state to the Order Repository. For information on `OrderManager.updateOrder()` and the `updateOrder` pipeline that it executes, see the Updating an Order with the OrderManager (page 278) subsection of *Saving Orders* in this chapter.

The following sections describes the important methods in `CartModifierFormHandler`. As can be seen from the method descriptions that follow, the `handleAddXXX` and `handleRemoveXXX` methods of `CartModifierFormHandler` automatically reprice the `Order` whenever the user makes changes to it.

However, you should note that users can also make changes to their orders through other purchase process form handlers that do not reprice orders, such as the form handlers that create and manage shipping groups. In these situations, where the user can change the current `Order` in ways that affect its price, and where the form handler used to process those changes does not reprice the `Order`, you should use the `RepriceOrderDroplet` servlet bean to reprice the `Order` before displaying its price to the user. For more information on `RepriceOrderDroplet`, see *Repricing Orders* section of the *ATG Commerce Guide to Setting Up a Store*.

getQuantity

Retrieves the quantity for the given item.

getCatalogKey

Retrieves a string that identifies the catalog to use when obtaining a `catalogRef` and `productRef` for the creation of a `CommerceItem`. The string is determined by the user's locale, which is obtained from the `Request` object. Consequently, the key is the user's locale and the value is the corresponding repository to use (for example, `en_US=ProductCatalog, fr_FR=FrenchCatalog`). `/atg/commerce/catalog/CatalogTools` maintains the key-to-catalog mapping.

getShippingGroupCommerceItemRelationships

Retrieves the list of `ShippingGroupCommerceItemRelationships` within the order.

handleAddItemToOrder

Adds items to an order by calling `addItemToOrder()`, which actually adds the items to the `Order`. It then calls `OrderManager.updateOrder()`.

See the `addItemToOrder()` method for more information.

handleSetOrder

Performs the actual work necessary to save an `Order`. It calls `modifyOrder()` to validate the user's changes and modify the `Order`. It then calls `runProcessSetOrder()`, which executes the pipeline set in `CartModifierFormHandler.setOrderChainId`. Finally, it calls `OrderManager.updateOrder()`.

See the `modifyOrder()` and `runProcessSetOrder()` methods for more information.

handleSetOrderByRelationshipId

Performs the actual work necessary to save an `Order`. It calls `modifyOrderByRelationshipId()` to validate the user's changes and modify the `Order`. It then calls `runProcessSetOrder()`, which executes the pipeline set in `CartModifierFormHandler.setOrderChainId`. Finally, it calls `OrderManager.updateOrder()`.

See the `modifyOrderByRelationshipId()` and `runProcessSetOrder()` methods for more information.

handleMoveToPurchaseInfo

Performs the actual work necessary to save an `Order`. Unlike `handleSetOrder()` and `handleSetOrderByRelationshipId()`, it also verifies that the `Order` is ready for checkout.

The handle method calls `modifyOrder()` to validate the user's changes and modify the `Order`. It then calls `runProcessMoveToPurchaseInfo()`, which executes the pipeline set in `CartModifierFormHandler.moveToPurchaseInfoChainId`. Finally, it calls `OrderManager.updateOrder()`.

See the `modifyOrder()` and `runProcessMoveToPurchaseInfo()` methods for more information.

handleMoveToPurchaseInfoByRelId

Performs the actual work necessary to save an `Order`. Unlike `handleSetOrder()` and `handleSetOrderByRelationshipId()`, it also verifies that the `Order` is ready for checkout.

The handle method calls `modifyOrderByRelationshipId()` to validate the user's changes and modify the `Order`. It then calls `runProcessMoveToPurchaseInfo()`, which executes the pipeline set in `CartModifierFormHandler.moveToPurchaseInfoChainId`. Finally, it calls `OrderManager.updateOrder()`.

See the `modifyOrderByRelationshipId()` and `runProcessMoveToPurchaseInfo()` methods for more information.

addItemToOrder

Invoked by `handleAddItemToOrder()`. The method calls `mergeItemInputForAdd()`. If all input values are valid, the method then calls `doAddItemToOrder()`.

See the `mergeItemInputForAdd()` and `doAddItemToOrder()` methods for more information.

mergeItemInputForAdd

Invoked by `addItemToOrder()` to unify the way input values are made available to `doAddItemToOrder()` and to validate input values.

The method first calls `CartModifierFormHandler.getItems()`. If the returned value is `null`, the method constructs an `items` array whose size matches the size of the return value from `CartModifierFormHandler.getCatalogRefIds()`. The method copies the values from `getCatalogRefIds()` into the `items` array elements. Then the method copies the values returned by the following `CartModifierFormHandler.getXXX` methods into the `items` array elements: `quantity`, `productId` or `productIds`, `value`, `commerceItemType`, `giftlistId`, and `giftlistItemId`.

If the initial `CartModifierFormHandler.getItems()` call retrieves a non-null value, the method copies the value returned by `CartModifierFormHandler.getCommerceItemType()` to every `items` array element whose `commerceItemType` subproperty was `null`. The method also calls `CartModifierFormHandler.mergeValueDictionaries()` to combine the Dictionary returned by `CartModifierFormHandler.getValueDictionary()` with each `items` array element's Dictionary.

doAddItemToOrder

Invoked by `addItemToOrder()`. The method retrieves the list of items to add by calling `CartModifierFormHandler.getItems()`, and calls `CartModifierFormHandler.getCatalogKey()` to determine which catalog to use. Then, for each item to add to the `Order`, the method uses the `PurchaseProcessHelper` class to do the following:

- Creates a `CommerceItem` using the `commerceItemType`, `catalogRefId`, `productId` and `quantity` found in the current array element from `getItems()`.
- Adds the created `CommerceItem` to the order. Note that the "new" item may represent an increase in the quantity of an existing item if the item added is already in the order.
- Copies custom values from the current item's `value` Dictionary to the new `CommerceItem`.
- Calls the `PurchaseProcessHelper.getShippingGroupForItem()` method to get a shipping group of the appropriate type. The type can be determined in one of three ways.
  - Passed in to `PurchaseProcessHelper` from the `CartModifierFormHandler.getItems()[ ].shippingGroupType`. The passed-in type is used along with the item's gift information (if any) to determine the correct shipping group to which the item should be added.
  - Use the first shipping group of the passed-in type (if that information is available) or the first shipping group on the order, regardless of the item type. If this is the desired behavior (perhaps you only sell goods with one shipping group type, and the default is always the correct type), set the `addItemToDefaultShippingGroup` property of the `/atg/commerce/order/purchase/PurchaseProcessHelper` component to `true`. This property is set to `true` by default.
  - `PurchaseProcessHelper` can determine the correct group from the item type (based on the SKU's `fulfiller` property value) and gift information. To use this behavior, set the `addItemToDefaultShippingGroup` property of the `/atg/commerce/order/purchase/PurchaseProcessHelper` component to `false`.
- Calls `PurchaseProcessHelper.addItemToShippingGroup()`, which calls `CommerceItemManager.addItemQuantityToShippingGroup()` which in turn takes the given quantity of the `CommerceItem` and the given `ShippingGroup` and creates a `ShippingGroupCommerceItemRelationship` of type `SHIPPINGQUANTITY`.

  For information on the `SHIPPINGQUANTITY` type of `ShippingGroupCommerceItemRelationship`, see Relationship Types (page 243) in the Using Relationship Objects (page 243) section of the *Working With Purchase Process Objects* (page 223) chapter.
- Calls `createConfigurableSubitems()`, which is an empty method that can be overridden as needed by sites that use configurable commerce items.

- Calls `processGiftAddition()`, which checks if the item that was added to the order is a gift (that is, the item's input `giftlistId` or `giftlistItemId` property is non-`null`). If the item is a gift, `processGiftAddition()` calls `GiftListManager.addGiftToOrder()` to perform additional gift list processing.

After the above steps have been taken for all the new items, `addItemToOrder()` calls `runProcessRepriceOrder()`, which reprices the `Order` by executing the pipeline set in `CartModifierFormHandler.repriceOrderChainId`. Then, for each new item `addItemToOrder()` calls `runProcessAddItemToOrder()`, which executes the pipeline set in `CartModifierFormHandler.addItemToOrderChainId`. Finally, the method fires a scenario event of type `ItemAddedToOrder` for each new item.

See the `runProcessRepriceOrder()` and `runProcessAddItemToOrder()` methods for more information.

modifyOrder

Invoked by `handleSetOrder()` and `handleMoveToPurchaseInfo()`.

The `modifyOrder()` method modifies the `Order` based on the changes made in the request. It iterates over each `CommerceItem` in the `Order`. For each `CommerceItem`, it retrieves the current quantity by calling `getQuantity()` and passing in the `catalogRefId` (SKU ID) of the item.

Next, the method checks if the `catalogRefId` of the current item is in the `removalCatalogRefIds` list. If it is, then the quantity of the current item is set to zero.

Then, the quantity of the current item is assessed and one of two actions occurs:

- If the quantity is greater than zero, the method sets the quantity in the `CommerceItem` and the corresponding `ShippingGroupCommerceItemRelationship`.
- If the quantity is less than or equal to zero, then it removes the `CommerceItem` and any associated `Relationship` objects from the `Order`.

modifyOrderByRelationshipId

Invoked by `handleSetOrderByRelationshipId()` and `handleMoveToPurchaseInfoByRelId()`.

The `modifyOrderByRelationshipId()` method updates the `Order` based on the changes made in the request and the existing `ShippingGroupCommerceItemRelationships` in the `Order`. It iterates over each `ShippingGroupCommerceItemRelationship` in the `Order`. For each `ShippingCommerceItemRelationship`, it first checks to make sure the `Relationship` is of type SHIPPINGQUANTITY. If it is not, then an exception is thrown. Then, the method retrieves the current quantity of the `ShippingGroupCommerceItemRelationship` by calling `getQuantity()` and passing in the `ShippingGroupCommerceItemRelationship` ID.

Next, the method checks if the ID of the `ShippingGroupCommerceItemRelationship` is in the `removalRelationshipIds` list. If it is, then the quantity of the current `ShippingGroupCommerceItemRelationship` is set to zero.

Then, the quantity of the current `ShippingGroupCommerceItemRelationship` is assessed and one of two actions occurs:

- If the quantity is greater than zero, then the quantity of the `ShippingGroupCommerceItemRelationship` and the quantity of the associated `CommerceItem` are adjusted appropriately.
- If the quantity is less than or equal to zero, then the `ShippingGroupCommerceItemRelationship` is removed from the `Order` and the quantity of the associated `CommerceItem` is adjusted appropriately.

For more information on the SHIPPINGQUANTITY type of ShippingGroupCommerceItemRelationship, see Relationship Types (page 243) in the Using Relationship Objects (page 243) section of the *Working With Purchase Process Objects* (page 223) chapter.

handleRemoveItemFromOrder

Removes items from the Order by CommerceItem ID. This handle method calls deleteItems() to delete the items from the Order and then calls OrderManager.updateOrder().

See the deleteItems() method for more information.

handleRemoveItemFromOrderByRelationshipId

Removes items from the Order by ShippingGroupCommerceItemRelationship ID. This handle method calls deleteItemsByRelationshipId() to delete the items from the Order and then calls OrderManager.updateOrder().

See the deleteItemsByRelationshipId() method for more information.

deleteItems

Deletes from the Order all CommerceItems whose IDs are in the removalCommerceIds property. The method also removes all associated ShippingGroupCommerceItemRelationships and calls runProcessRepriceOrder(), which reprices the Order by executing the pipeline set in CartModifierFormHandler.repriceOrderChainId. Finally, the method fires a scenario event of type ItemRemovedFromOrder.

See the runProcessRepriceOrder() method for more information.

deleteItemsByRelationshipId

This method deletes CommerceItems from the Order by ShippingGroupCommerceItemRelationship ID. It iterates through the IDs in the removalShippingGroupCommerceItemRelIds property. For each ID, it first ensures that the Relationship type is of type SHIPPINGQUANTITY (logging an error if it is not), and then it removes the HandlingInstructions associated with the ShippingGroup. Next, one of two conditions can exist:

- If the quantity in the ShippingGroupCommerceItemRelationship is greater than or equal to the quantity in the CommerceItem, then the CommerceItem and all associated Relationships are removed from the Order. The method then calls runProcessRepriceOrder(), which reprices the Order by executing the pipeline set in CartModifierFormHandler.repriceOrderChainId. Finally, the method fires a scenario event of type ItemRemovedFromOrder. (See the runProcessRepriceOrder() method in this table for more information.)

- If the quantity in the ShippingGroupCommerceItemRelationship is less than the quantity in the CommerceItem, but the CommerceItem has Relationships to other ShippingGroups, then the quantity in the CommerceItem is reduced and the given ShippingGroupCommerceItemRelationship removed.

runProcessAddItemToOrder

Invoked by the handleAddItemToOrder() method. This method runs the pipeline set in CartModifierFormHandler.addItemToOrderChainId. By default, this property is set to addItemToOrder.

Note: By default, the addItemToOrder pipeline is commented out of commercepipeline.xml, the commerce pipeline configuration file. It is provided for extension purposes, should you need to include additional functionality, such as scenario events.

runProcessSetOrder

Invoked by `handleSetOrder()` and `handleSetOrderByRelationshipId()`. This method runs the pipeline set in `CartModifierFormHandler.setOrderChainId`. By default, this property is set to `setOrder`.

Note: By default, the `setOrder` pipeline is commented out of `commercepipeline.xml`, the commerce pipeline configuration file. It is provided for extension purposes, should you need to include additional functionality, such as scenario events.

runProcessRepriceOrder

Runs the pipeline to execute whenever the order needs to be repriced. The pipeline to run is set in `CartModifierFormHandler.repriceOrderChainId`. By default, this property is set to `repriceOrder`. By default, this method executes an ORDER_TOTAL pricing operation. (For more information about pricing operations, see the *Repricing Shopping Carts* section of the *ATG Commerce Guide to Setting Up a Store*.)

runProcessMoveToPurchaseInfo

Invoked by `handleMoveToPurchaseInfo()` and `handleMoveToPurchaseInfoByRelId()`. This method runs the pipeline set in `CartModifierFormHandler.moveToPurchaseInfoChainId`. By default, this property is set to `moveToPurchaseInfo`.

The `moveToPurchaseInfo` pipeline, in turn, executes the `validateForCheckout` pipeline, which verifies that the `Order` is ready for checkout. For more information on both pipelines, see Appendix F, *Pipeline Chains* (page 699).

Modifying the Current Order

To modify an `Order`, you must supply either a `CatalogRefId` of a `CommerceItem` or a `ShippingGroupCommerceItemRelationship` ID. It is recommended that you modify an `Order` by `ShippingGroupCommerceItemRelationship` ID, especially if you intend to support complex product-SKU relationships, such as multiple `CommerceItems` with the same `catalogRefId` (SKU ID) or multiple shipping groups. For example, a customer could order 5 of a given item and choose to ship a quantity of 3 to a home address and the remaining 2 to a work address. In this example, to remove the items being shipped to the work address, you would modify (and ultimately remove) the `ShippingGroupCommerceItemRelationship` instead of modifying the `CommerceItem`.

The following subsections describes both order modification methods:

- Modifying an Order by catalogRefId (page 274)
- Modifying an Order by ShippingGroupCommerceItemRelationship ID (page 275)

Modifying an Order by catalogRefId

Modifying orders by `catalogRefId` works for very simple sites. Because it does not provide the granularity necessary to delete just a part of a `CommerceItem`, it is not recommended for sites with complex features, such as multiple `CommerceItems` with the same `catalogRefId` or multiple shipping groups.

You can use the following `CartModifierFormHandler` methods to modify an `Order` by `catalogRefId`:

- `handleSetOrder()`
- `handleRemoveItemFromOrder()`

- handleMoveToPurchaseInfo()

Refer to Understanding the CartModifierFormHandler (page 269) for more information on these handle methods.

To change the quantities of items in an Order using the catalogRefIds of CommerceItems, call the CartModifierFormHandler.handleSetOrder() method for each CommerceItem whose quantity you want to change and pass in the catalogRefId and quantity for the CommerceItem. This is illustrated in the following JSP code:

```
<dsp:input bean='beanName' value='<dsp:valueof param="CommerceItem.quantity"/>'
type="text" name='<dsp:valueof param="CommerceItem.catalogRefId"/>'/>
```

Note that if no quantity is found for a CommerceItem, then the CommerceItem is removed from the Order.

To remove items from an Order using the catalogRefIds of CommerceItems, edit the JSPs that invoke the CartModifierFormHandler handle methods that delete items from the Order. Populate the form handler's removalCatalogRefIds array with the catalogRefIds of the CommerceItems to be removed. For example, you can populate the array using following JSP code:

```
<dsp:input bean="CartModifierFormHandler.removalCatalogRefIds"
paramvalue="CommerceItem.catalogRefId" type="checkbox"/>
```

Modifying an Order by ShippingGroupCommerceItemRelationship ID

If your sites support complex product-SKU relationships (for example, multiple CommerceItems with the same catalogRefId) or multiple shipping groups, then it is recommended that you modify an Order using the IDs of the ShippingGroupCommerceItemRelationship objects in the Order. Doing so makes the changes at the CommerceItem-to-ShippingGroup level.

You can use the following CartModifierFormHandler methods to modify an Order by ShippingGroupCommerceItemRelationship ID:

- handleSetOrderByRelationshipId()
- handleRemoveItemFromOrderByRelationshipId()
- handleMoveToPurchaseInfoByRelId()

Refer to Understanding the CartModifierFormHandler (page 269) for more information on these handle methods.

To change the quantity of an item in the Order, pass the new quantity into the ShippingpingGroupCommerceItemRelationship, as shown in the following JSP example:

```
<dsp:input value='<dsp:valueof param="SgCiRelationship.quantity"/>'
type="text" name='<dsp:valueof param="SgCiRelationship.Id"/>'>
```

To delete an item from the Order, pass the ID of the associated ShippingGroupCommerceItemRelationship into the form handler's removalRelationshipIds property, as shown in the following JSP example:

```
<dsp:input bean="CartModifierFormHandler.removalRelationshipIds"
```

```
paramvalue="SgCiRelationship.Id" type="checkbox"/>
```

Repricing Orders

As described in Modifying Orders (page 269) and Checking Out Orders (page 280) in this chapter, two form handlers in the Oracle ATG Web Commerce purchase process have handle methods that you can use to reprice an `Order`:

- `CartModifierFormHandler`, which is used to modify orders by adding and removing items and changing item quantities.
- `ExpressCheckoutFormHandler`, which manages and expedites the pre-checkout processing of orders.

However, you'll need to reprice orders via some other mechanism if customers can make order changes that affect order price through other form handlers that do not reprice orders (for example, by making shipping changes via the form handlers that create and manage shipping groups), or if the orders are modified through some other means in ways that affect order price, such as the delivery of a promotion via a scenario.

If your sites have any pages where you need to reprice an `Order`, but you cannot do so through a form action and corresponding handle method, use the `RepriceOrderDroplet` servlet bean. By default, the servlet bean is configured to invoke the `repriceAndUpdateOrder` pipeline, which reprices the `Order` by calling the `repriceOrder` pipeline and then updates the `Order` by calling `OrderManager.updateOrder()`.

The `RepriceOrderDroplet` servlet bean is an instance of `atg.commerce.order.purchase.RepriceOrder`, which extends `atg.service.pipeline.servlet.PipelineChainInvocation`. Oracle ATG Web Commerce provides an instance of `RepriceOrder`, which is located in Nucleus at `/atg/commerce/order/purchase/RepriceOrderDroplet`.

The `RepriceOrder` class provides the required objects for executing a repricing pipeline as convenient properties. Typically, execution of a repricing pipeline requires the `Order`, the `Profile`, the `OrderManager`, and the user's `PricingModelHolder`. `RepriceOrder` is conveniently configured to reference these objects, which means that a page developer doesn't need to supply them as input parameters every time the `RepriceOrderDroplet` servlet bean is invoked. Consequently, the only **required** parameter that must be supplied is the pricing operation to execute. Acceptable pricing operations are defined in the `atg.commerce.pricing.PricingConstants` interface; they are the following:

Pricing Operation	Pricing Constant
ORDER_TOTAL	PricingConstants.OP_REPRICE_ORDER_TOTAL
ORDER_SUBTOTAL	PricingConstants.OP_REPRICE_ORDER_SUBTOTAL
ORDER_SUBTOTAL_SHIPPING	PricingConstants.OP_REPRICE_ORDER_SUBTOTAL_SHIPPING
ORDER_SUBTOTAL_TAX	PricingConstants.OP_REPRICE_ORDER_SUBTOTAL_TAX
ITEMS	PricingConstants.OP_REPRICE_ITEMS
SHIPPING	PricingConstants.OP_REPRICE_SHIPPING

Pricing Operation	Pricing Constant
ORDER	PricingConstants.OP_REPRICE_ORDER
TAX	PricingConstants.OP_REPRICE_TAX
NO_REPRICE	PricingConstants.OP_NO_REPRICE

The following code sample is taken from `RepriceOrderDroplet.properties` and indicates its default configuration:

```
$class=atg.commerce.order.purchase.RepriceOrder
$scope=request

defaultPipelineManager=/atg/commerce/PipelineManager
defaultChainId=repriceAndUpdateOrder
order^=/atg/commerce/ShoppingCart.current
profile=/atg/userprofiling/Profile
orderManager=/atg/commerce/order/OrderManager
userPricingModels=/atg/commerce/pricing/UserPricingModels
```

This default configuration enables a page developer to include the `RepriceOrderDroplet` servlet bean on any shopping cart page that requires the repricing and updating of `Orders` with the following JSP code:

```
<dsp:droplet name="RepriceOrderDroplet">
  <dsp:param value="ORDER_SUBTOTAL" name="pricingOp"/>
</dsp:droplet>
```

For information on all of the input, output, and open parameters of `RepriceOrderDroplet`, see the *RepriceOrder* reference entry in *Appendix: ATG Commerce Servlet Beans* of the *ATG Commerce Guide to Setting Up a Store*. For information on the `OrderManager.updateOrder()` method and the `updateOrder` pipeline, see Updating an Order with the OrderManager (page 278) in this chapter. For more information about pipelines, the `PipelineManager`, and the transactional modes and transitions of the processors in the `repriceOrder` pipeline, see Appendix F, *Pipeline Chains* (page 699).

Saving Orders

The `SaveOrderFormHandler` (class `atg.commerce.order.purchase.SaveOrderFormHandler`) saves the user's current `Order` and adds the `Order` to the `ShoppingCart`'s list of saved orders. Additionally, it constructs a new, empty `Order` and sets it as the user's current `Order`. Oracle ATG Web Commerce includes an instance of `SaveOrderFormHandler`, which is located in Nucleus at `/atg/commerce/order/purchase/SaveOrderFormHandler`.

If you are writing custom code, to avoid the possibility of deadlocks, be sure to synchronize on the Order before beginning the transaction.

The following table describes the important methods in `SaveOrderFormHandler`.

Method	Description
handleSaveOrder	This handle method first calls the empty preSaveOrder() method, then calls the saveOrder() method to save the order, and finally calls the empty postSaveOrder() method.
saveOrder	This method first sets the current Order's description based on the provided String description. If no description is provided, the method sets the description using the date and time as represented by the user's locale. Next, the method saves the Order to the repository by calling the OrderManager.updateOrder() method. Finally, the method adds the Order to the list of saved orders in the ShoppingCart.saved property, and constructs a new, empty Order that is set as the user's current order. For more information on OrderManager.updateOrder(), see Updating an Order with the OrderManager (page 278) below.

Updating an Order with the OrderManager

The actual saving of an Order object occurs by calling the updateOrder() method in the OrderManager. The updateOrder() method calls into the PipelineManager to execute the updateOrder pipeline. Each processor in the pipeline saves a different type of commerce object.

The following table describes the individual processors in the updateOrder pipeline. They are listed in order of execution.

PipelineLink name	Description
updateOrderObject	Saves an Order object's properties to the repository. The Order is supplied in the optional user parameter of the PipelineManager. The class that implements this functionality is atg.commerce.order.processor.ProcSaveOrderObject.
updateCommerceItemObjects	Saves the CommerceItem properties for the items in the Order. The class that implements this functionality is atg.commerce.order.processor.ProcSaveCommerceItemObjects.
updateShippingGroupObjects	Saves the ShippingGroup properties for the shipping groups in the Order. The class that implements this functionality is atg.commerce.order.processor.ProcSaveShippingGroupObjects.
updateHandlingInstructionObjects	Saves the HandlingInstruction properties for the handling instructions in the Order. The class that implements this functionality is atg.commerce.order.processor.ProcSaveHandlingInstructionObjects.

PipelineLink name	Description
updatePaymentGroupObjects	Saves the PaymentGroup properties for the payment groups in the Order. The class that implements this functionality is atg.commerce.order.processor.ProcSavePaymentGroupObjects.
updatePaymentStatusObjects	Saves the PaymentStatus properties for the PaymentStatus objects in all the payment groups in the Order. These are the authorizationStatus, debitStatus, and creditStatus properties in the PaymentGroup class. The class that implements this functionality is atg.commerce.order.processor.ProcSavePaymentStatusObjects.
updateRelationshipObjects	Saves the Relationship properties for the Relationships in the Order. The class that implements this functionality is atg.commerce.order.processor.ProcSaveRelationshipObjects.
updatePriceInfoObjects	Saves the PriceInfo properties for the PriceInfo in the Order. The properties saved are in Order (priceInfo and taxPriceInfo), ShippingGroup (priceInfo), and CommerceItems (priceInfo). The class that implements this functionality is atg.commerce.order.processor.ProcSavePaymentGroupObjects.
setLastModifiedTime	Sets the lastModifiedTime property in the Order object if any changes were made to the Order. If no changes were made, then the lastModifiedTime is not changed. The class that implements this functionality is atg.commerce.order.processor.ProcSetLastModifiedTime.

For more information about pipelines, the PipelineManager, and the transactional modes and transitions of the processors in the updateOrder pipeline, see Appendix F, *Pipeline Chains* (page 699).

Canceling Orders

The CancelOrderFormHandler (class atg.commerce.order.purchase.CancelOrderFormHandler) cancels the user's current Order, which deletes the Order from the ShoppingCart. Oracle ATG Web Commerce includes an instance of CancelOrderFormHandler, which is located in Nucleus at /atg/commerce/order/purchase/CancelOrderFormHandler.

The following table describes the important methods in CancelOrderFormHandler.

Method	Description
handleCancelOrder	This handle method calls either the deleteOrder() method or the preserveOrder() method (depending on whether the Order can be deleted).
deleteOrder	If the state of the current Order is one of the configured states in the CancelOrderFormHandler.deleteStates property, then the deleteOrder() method deletes the current Order from the user's ShoppingCart.
preserveOrder	If the state of the current Order isn't one of the configured states in the CancelOrderFormHandler.deleteStates property, then the preserveOrder() method simply sends a ModifyOrder GenericRemove notification message to Fulfillment for any action deemed appropriate.

Checking Out Orders

The order checkout process can vary depending on the requirements and complexities of your sites. This section describes the checkout process for both simple and complex sites and includes the following sections:

- Preparing a Simple Order for Checkout (page 280)
  Describes the use of ExpressCheckoutFormHandler, which manages and expedites the pre-checkout processing of orders. Intended for sites that support only a single HardgoodShippingGroup and CreditCard.
- Preparing a Complex Order for Checkout (page 282)
  Describes the various form handlers that manage the pre-checkout processing of orders. Intended for sites that support any number or type of shipping group, or any number or type of payment group.
- Checking Out an Order (page 294)
  Describes the processing of an Order after the customer has supplied all necessary information for the Order and has submitted it for checkout.

Preparing a Simple Order for Checkout

If your sites support the use of only a single HardgoodShippingGroup and a single CreditCard for a given Order, you can manage and expedite the pre-checkout process for Orders using the ExpressCheckoutFormHandler (class atg.commerce.order.purchase.ExpressCheckoutFormHandler). ExpressCheckoutFormHandler supports the use of a single Profile-derived HardgoodShippingGroup and a single Profile-derived CreditCard.

However, if your sites support any number or type of shipping group, or any number or type of payment group, then you must use the form handlers described in Preparing a Complex Order for Checkout (page 282). Note that the form handlers described in that section also work with simple Orders that have a single HardgoodShippingGroup and a single CreditCard.

Oracle ATG Web Commerce provides an instance of ExpressCheckoutFormHandler, which is located in Nucleus at /atg/commerce/order/purchase/ExpressCheckoutFormHandler. The following table describes its important methods:

Method	Description
handleExpressCheckout	This handle method first invokes the runRepricingProcess() method to reprice the Order, then calls OrderManager.updateOrder() to save the Order in its present state to the Order Repository, and finally calls commitOrder() to submit the Order for checkout. For more information on OrderManager.updateOrder() and the updateOrder pipeline that it executes, see the Updating an Order with the OrderManager (page 278) subsection of *Saving Orders* in this chapter.
runRepricingProcess	Reprices the Order by running the pipeline specified in ExpressCheckoutFormHandler.repriceOrderChainId. By default, this property is set to repriceOrder. For more information on the repriceOrder pipeline, see Appendix F, *Pipeline Chains* (page 699).
commitOrder	This method first ensures that the user isn't trying to double-submit the Order by checking if the ID of the current Order is equal to the ID of the user's last Order (in ShoppingCart.last). If the IDs are not equal, then the current Order can be submitted for checkout. The method then calls the OrderManager.processOrder() method, which executes the processOrder pipeline (See Checking Out an Order (page 294) later in this chapter.). Finally, the method sets the submitted Order as the user's last Order (in ShoppingCart.last), and it constructs a new, empty Order and sets it as the user's current Order (in ShoppingCart.current).

The following boolean properties of the ExpressCheckoutFormHandler govern its behavior:

Property Name	Description
paymentGroupNeeded	If true, then a CreditCard payment group is automatically taken from the Profile. If false, then the user can supply the CreditCard information in a form through the ExpressCheckoutFormHandler.paymentGroup property.
shippingGroupNeeded	If true, then a HardgoodShippingGroup is automatically taken from the Profile. If false, then the user can supply the HardgoodShippingGroup information in a form through the ExpressCheckoutFormHandler.shippingGroup property.
commitOrder	If true, then the ExpressCheckoutFormHandler.handleExpressCheckout() method submits the Order for checkout. You can set this property to false if you want to display a confirmation page before committing the Order. By default, this property is set to false.

Note: Recall that, as with all shopping cart-related form handlers, empty preXXX and postXXX methods are provided so you can extend `ExpressCheckoutFormHandler`, as necessary. To implement a system that requires a more complex checkout process, see Preparing a Complex Order for Checkout (page 282).

See Checking Out an Order (page 294) for detailed information on the order checkout process.

Preparing a Complex Order for Checkout

Oracle ATG Web Commerce provides several form handlers to support a checkout process that uses any number or type of shipping group and payment group. If your sites have this type of complex checkout process, then you should use the form handlers described in this section instead of `ExpressCheckoutFormHandler`. (For more information on `ExpressCheckoutFormHandler`, see Preparing a Simple Order for Checkout (page 280).)

The form handlers described in this section manage different subprocesses in the pre-checkout process, which makes it easier for you to extend them when necessary. Separate form handlers exist to support the following tasks:

Creating Shipping Groups

Oracle ATG Web Commerce provides two implementations of the `CreateShippingGroupFormHandler` interface to support the form-driven creation of hard good and electronic shipping groups. These form handler classes create the `ShippingGroups` and optionally add them to the `ShippingGroupMapContainer`. Once the shipping groups are added to the `ShippingGroupMapContainer`, the user can then select from among them for use in the current `Order`.

The two default implementations of `CreateShippingGroupFormHandler` are:

- **CreateHardgoodShippingGroupFormHandler**

  This form handler class creates a `HardgoodShippingGroup` and exposes it via a `getHardgoodShippingGroup()` method, which makes it possible for users to edit its properties directly via JSP forms. Commerce provides an instance of `atg.commerce.order.purchase.CreateHardgoodShippingGroupFormHandler`; it is located in Nucleus at `/atg/commerce/order/purchase/CreateHardgoodShippingGroup FormHandler`.

  To create the `HardgoodShippingGroup`, the `handleNewHardgoodShippingGroup()` method invokes the `createHardgoodShippingGroup()` method, which actually creates the shipping group. The form handler's `hardgoodShippingGroupType` property determines the type of shipping group to create; by default, this property is set to `hardgoodShippingGroup`. The form handler's `hardgoodShippingGroupName` property determines the name of the new shipping group, as referenced in the `ShippingGroupMapContainer`.

  `CreateHardgoodShippingGroupFormHandler` is configured to use `/atg/commerce/util/AddressValidator` to validate the shipping address before creating the shipping group. The default configuration is:

  - `validateFirstName=true`

- validateLastName=true
- validateAddress1=true
- validateCity=true
- validateCounty=false
- validateState=true
- validatePostalCode=true
- validateCountry=true
- validateEmail=false
- validatePhoneNumber=false
- validateFaxNumber=false

This validation can be performed every time the shipping group is updated, via the UpdateHardgoodShippingGroupFormHandler class.

Finally, the form handler's addToContainer property determines whether the new shipping group is added to the ShippingGroupMapContainer and made the default shipping group; by default, this property is set to True. Once the HardgoodShippingGroup is added to the ShippingGroupMapContainer, the user can use it when checking out the Order.

After creating the HardgoodShippingGroup, you can use the UpdateHardgoodShippingGroupFormHandler to handle shipping address changes. UpdateHardgoodShippingGroupFormHandler can update this information in any or all of three places, based on these properties:

- updateContainer—Updates the HardgoodShippingGroup in the ShippingGroupMapContainer.
- updateProfile—Updates the shipping address in the profile.
- updateOrder—Updates the HardgoodShippingGroup in the order.

- **CreateElectronicShippingGroupFormHandler**

  This form handler class creates an ElectronicShippingGroup and exposes it via a getElectronicShippingGroup() method, which makes it possible for users to edit its properties directly via JSP forms. Oracle ATG Web Commerce provides an instance of atg.commerce.order.purchase.CreateElectronicShippingGroupFormHandler; it is located in Nucleus at /atg/commerce/order/purchase/CreateElectronicShippingGroupFormHandler.

  To create the ElectronicShippingGroup, the handleNewElectronicShippingGroup() method invokes the createElectronicShippingGroup() method, which actually creates the shipping group and sets the shipping group's emailAddress property. The form handler's electronicShippingGroupType property determines the type of shipping group to create; by default, this property is set to electronicShippingGroup. The form handler's electronicShippingGroupName property determines the name of the new shipping group, as referenced in the ShippingGroupMapContainer. Finally, the form handler's addToContainer property determines whether the new shipping group is added to the ShippingGroupMapContainer and made the default shipping group; by default, this property is set to True. Once the ElectronicShippingGroup is added to the ShippingGroupMapContainer, the user can use it when checking out the Order.

After creating the ElectronichippingGroup, you can use the UpdateShippingGroupFormHandler to handle updates to the shipping group. UpdateShippingGroupFormHandler can update shipping group information in the container and/or the order, based on these properties:

- updateContainer—Updates the ElectronicShippingGroup in the ShippingGroupMapContainer.
- updateOrder—Updates the ElectronicShippingGroup in the order.

You can also create Profile-derived ShippingGroups and add them to the ShippingGroupMapContainer by using the ShippingGroupDroplet servlet bean (class atg.commerce.order.purchase.ShippingGroupDroplet). The input parameters passed into ShippingGroupDroplet determine what types of ShippingGroups are created (hard good, electronic, or both) and whether the ShippingGroupMapContainer is cleared before they are created. For a detailed list of these input parameters, as well as output parameters, open parameters, and a code example, see the *Adding Shipping Information to Shopping Carts* section of the *Implementing Order Retrieval* chapter of the *ATG Commerce Guide to Setting Up a Store*.

To initialize the ShippingGroup objects, the service method of ShippingGroupDroplet calls initializeUsersShippingMethods(), which initializes one or more ShippingGroups for the current user and adds them to the ShippingGroupMapContainer. For each entry in ShippingGroupDroplet.shippingGroupTypes (which is supplied via an input parameter), its corresponding ShippingGroupInitializer is obtained from the ServiceMap in ShippingGroupDroplet.shippingGroupInitializers (keyed by ShippingGroup type). The initializeShippingGroups() method of the ShippingGroupInitializer is then used to initialize the ShippingGroup and add it to the ShippingGroupMapContainer.

Note that Oracle ATG Web Commerce provides two implementations of the ShippingGroupInitializer interface, namely HardgoodShippingGroupInitializer and ElectronicShippingGroupInitializer. The former creates HardgoodShippingGroups based on their existence in the user's Profile, and the latter creates ElectronicShippingGroups based on the existence of an e-mail address in the user's Profile.

To use this framework with a new ShippingGroup type that you create, first, write a new ShippingGroupInitializer implementation. Its initializeShippingGroups() method should gather the user's ShippingGroups by type and add them to the ShippingGroupMapContainer referenced by the ShippingGroupFormHandler. For example, the ElectronicShippingGroupInitializer queries the Profile's email property and applies the result to a new ElectronicShippingGroup, which is subsequently added to the ShippingGroupMapContainer. Second, register a Nucleus component for the new ShippingGroupInitializer implementation and add it to the ServiceMap in ShippingGroupDroplet.shippingGroupInitializers, which is keyed by ShippingGroup type. Finally, include the new ShippingGroup type in the ShippingGroupDroplet.shippingGroupTypes parameter on those site pages where the new ShippingGroup type is utilized.

Associating Shipping Groups with an Order and Its Items

When the user has supplied the shipping information for an Order, the ShippingGroupFormHandler can be used to create and manage the associations between the ShippingGroups and the items in the Order. Oracle ATG Web Commerce provides a request-scoped instance of atg.commerce.order.purchase.ShippingGroupFormHandler, which is located in Nucleus at /atg/commerce/order/purchase/ShippingGroupFormHandler.

The ShippingGroupFormHandler works in conjunction with the ShippingGroupDroplet to manipulate the relationships between CommerceItems and ShippingGroups in the order. Handling Instructions and their relationships to CommerceItems and ShippingGroups are also manipulated based on the relationship between each CommerceItem and the ShippingGroups.

By default, this form handler invokes the `validateShippingInfo` pipeline chain to validate `ShippingGroup` information. To skip validation, set the `validateShippingGroups` property to `false`. To learn more about the pipeline chain, see the validateShippingInfo Pipeline Chain (page 720) section.

It should be noted that `ShippingGroupFormHandler` does **not** reprice the given `Order`. Consequently, if you enable customers to make order changes that affect order price through this form handler, you should then reprice the given `Order` using the `RepriceOrderDroplet` servlet bean before displaying its price to the customer. For more information on `RepriceOrderDroplet`, see *Repricing Orders* section of the *ATG Commerce Guide to Setting Up a Store*.

The `ShippingGroupFormHandler` is composed of the following containers:

- `atg.commerce.order.purchase.ShippingGroupMapContainer`, which defines a `Map` of user-assigned `ShippingGroup` names to `ShippingGroups`. This container stores the user's potential `ShippingGroups` for the `Order`.

- `atg.commerce.order.purchase.CommerceItemShippingInfoContainer`, which defines a `Map` of `CommerceItems` to `CommerceItemShippingInfo` Lists. This container stores the user's `CommerceItemShippingInfo` objects for the `CommerceItems` in the `Order`.

Additionally, the `ShippingGroupFormHandler` uses the following helper classes:

- `atg.commerce.order.purchase.CommerceItemShippingInfo`, which represents the association between a `CommerceItem` and its shipping information and includes properties such as `quantity`, `splitQuantity`, and `relationshipType`. These objects store the information needed to create `ShippingGroupCommerceItemRelationships` for the `Order`.

- `atg.commerce.order.purchase.HandlingInstructionInfo`, which is used to associate handling instruction information with each `CommerceItemShippingInfo` object. `HandlingInstructionInfo` is stored in a `handingInstructionInfos` list property of the `CommerceItemShippingInfo` object.

- `atg.commerce.order.purchase.ShippingGroupDroplet`, which contains a reference to both the `ShippingGroupMapContainer` and `CommerceItemShippingInfoContainer`. The `ShippingGroupDroplet` servlet bean is used to initialize `ShippingGroup` objects and `CommerceItemShippingInfo` objects for use by the `ShippingGroupFormHandler`. The resulting collections of `ShippingGroups` and `CommerceItemShippingInfos` are exposed via the output parameters of the servlet bean. For more information on using the `ShippingGroupDroplet` to initialize `ShippingGroup` objects, see the previous section, Creating Shipping Groups (page 282). For more information on initializing `CommerceItemShippingInfo` objects, see below.

- `atg.commerce.order.purchase.CommerceItemShippingInfoTools`, which contains helper methods for creating, modifying, removing and applying the `CommerceItemShippingInfo` and `ShippingGroup` objects in the `CommerceItemShippingInfoContainer` and `ShippingGroupMapContainer` containers respectively. `CommerceItemShippingGroupTools` includes the `includeGifts` flag, which determines how gift items are handled when split across shipping groups.

With these helper classes and containers, the `ShippingGroupFormHandler` adds the necessary `ShippingGroups` to the `Order`, establishes their relationships to the `CommerceItems`, performs validation, and updates the `Order`. The following table describes the handle methods used in these processes:

Method	Description of Functionality
`handleSplitShippingInfos`	This handle method splits the quantities of `CommerceItems` across several `CommerceItemShippingInfo` objects. The handle method calls `splitShippingInfos()`, which retrieves the list of `CommerceItemShippingInfo` objects from the `CommerceItemShippingInfoContainer`. The method then iterates through the list. For each `CommerceItemShippingInfo`, it retrieves the `quantity` and the `splitQuantity`. If the `splitQuantity` is greater than zero and not greater than the `quantity`, then the method calls `splitCommerceIdentifierShippingInfoByQuantity()`. In turn, `splitCommerceIdentifierShippingInfoByQuantity()` creates a new `CommerceIdentifierShippingInfo` object, adjusts the properties of both the new and existing objects, and adds the new object to the `CommerceItemShippingInfoContainer`.
`handleSpecifyDefault ShippingGroup`	This handle method is used to let the user specify a default `ShippingGroup` to use for shipping. The method calls `specifyDefaultShippingGroup()`, which sets the `defaultShippingGroupName` in the `ShippingGroupMapContainer`. Setting the default `ShippingGroup` can facilitate simpler applications that permit only one `ShippingGroup` per `Order`, as well as advanced applications that apply a default `ShippingGroup` to any remaining items not explicitly covered by other `ShippingGroups`.

Method	Description of Functionality
handleApplyShippingGroups	This handle method adds the ShippingGroups to the Order. It is used when the customer has supplied the necessary shipping information for the Order and is ready to proceed with the next checkout phase. The handle method calls applyShippingGroups(), which first calls ShippingGroupManager.removeAllShippingGroupsFromOrder() to remove any existing ShippingGroups from the Order. This ensures a clean Order. The applyShippingGroups() method then calls applyCommerceItemShippingInfo(), which applies all CommerceItemShippingInfo objects to the Order. The applyCommerceItemShippingInfo() method iterates through the list of CommerceItemShippingInfo objects in the CommerceItemShippingInfoContainer. For each CommerceItemShippingInfo object, the associated ShippingGroup is retrieved and added to the Order (if it isn't already in the Order). Then the method retrieves the Relationship type of the current CommerceItemShippingInfo object and calls either CommerceItemManager.addItemQuantityToShippingGroup() or CommerceItemManager.addRemainingItemQuantityToShipping(), depending on the relationshipType (SHIPPINGQUANTITY OR SHIPPINGQUANTITYREMAINING). This adds the appropriate quantity of the CommerceItem to the ShippingGroup. Next, if the form handler's applyDefaultShippingGroup property is True, then the applyShippingGroups() method checks for a default shipping group in the ShippingGroupMapContainer. If one exists, then the remaining quantity of all CommerceItems in the Order is added to the default shipping group. Then, the handle method calls runProcessValidateShippingGroups() to validate the ShippingGroups in the Order. This executes the shipping validation pipeline specified in ShippingGroupFormHandler.validateShippingGroupsChainId; by default, this property is set to validateShippingInfo. For information on the validateShippingInfo pipeline, see Appendix F, *Pipeline Chains* (page 699). Finally, the handle method calls OrderManager.updateOrder() to save the Order in its present state to the Order Repository. For more information on OrderManager.updateOrder() and the updateOrder pipeline that it executes, see the Updating an Order with the OrderManager (page 278) subsection of *Saving Orders* in this chapter.

As previously mentioned, the ShippingGroupDroplet servlet bean is used to initialize CommerceItemShippingInfo objects and add them to the CommerceItemShippingInfoContainer, so they can be used by the ShippingGroupFormHandler. To initialize the CommerceItemShippingInfo objects, the service method calls initializeCommerceItemShippingInfos() which, by default, creates and initializes a CommerceItemShippingInfo for each CommerceItem in the Order and adds them to the CommerceItemShippingInfoContainer. Each new CommerceItemShippingInfo references the default ShippingGroup in the ShippingGroupMapContainer.

The paragraph above describes the default behavior of the ShippingGroupDroplet. The droplet, however, has several input parameters that control whether and how CommerceItemShippingInfo objects are created. An additional parameter controls whether the CommerceItemShippingInfoContainer is cleared before the objects are created. For a detailed list of these input parameters, as well as ShippingGroupDroplet output parameters, open parameters, and a code example, see the *Adding Shipping Information to Shopping Carts* section of the *Implementing Order Retrieval* chapter of the *ATG Commerce Guide to Setting Up a Store*.

The HandlingInstructionInfo class makes it possible to split, merge, and apply handling information along with the CommerceItemShippingInfo objects with which it is associated. When the ShippingGroupDroplet initializes a container based on the current contents of an Order, HandlingInstructionInfo objects are automatically generated for any handling instructions currently in the Order. These HandlingInstructionInfo objects are then associated with the appropriate CommerceItemShippingInfo objects based on the CommerceItem referenced in the handling instruction.

Gift items in an order are identified by a special handling instruction (GiftHandlingInstruction), and therefore the ShippingGroupDroplet and ShippingGroupFormHandler automatically handle splitting and merging of gift items in the order.

If the CommerceItemShippingInfoTools includeGifts property is set to false (the default), it has the following effects:

- CommerceItemShippingInfo objects don't include the quantity of the items designated as gifts. For example, if an item is quantity 3, and 1 is a gift, a CommerceItemShippingInfo will be created for a quantity of 2.
- GiftHandlingInstructions are not included in the HandlingInstructionInfos associated with the CommerceItemShippingInfos
- ShippingGroups that contain only gifts are not added to the ShippingGroupMapContainer

The result is that none of the gift-related objects in the Order are added to the CommerceItemShippingInfoContainer and ShippingGroupMapContainer when initializing based on the Order, and after applying, they remain unchanged in the Order.

Creating Payment Groups

Oracle ATG Web Commerce provides two implementations of the CreatePaymentGroupFormHandler interface to support the form-driven creation of credit card and invoice payment groups. These form handler classes create the payment groups and optionally add them to the PaymentGroupMapContainer. Once the payment groups are added to the PaymentGroupMapContainer, the user can then select from among them for use in the current Order.

The default implementations of the CreatePaymentGroupFormHandler are:

- CreateCreditCardFormHandler

  This form handler creates a CreditCard payment group and exposes it via a getCreditCard() method, which makes it possible for users to edit its properties directly via JSP forms. Commerce provides an instance of atg.commerce.order.purchase.CreateCreditCardFormHandler; it is located in Nucleus at /atg/commerce/order/purchase/CreateCreditCardFormHandler.

To create the `CreditCard` payment group, the `handleNewCreditCard()` method invokes the `createCreditCard()` method, which actually creates the payment group. The form handler's `creditCardType` property determines the type of `CreditCard` payment group to create; by default, this property is set to `creditCard`. The form handler's `creditCardName` property determines the name of the new payment group, as referenced in the `PaymentGroupMapContainer`. The `generateNickname` property allows Commerce to automatically generate a name for credit cards if the user does not provide one. The generated name uses the credit card type and the last four digits of the number (for example, Visa - 3674).

Note that if `generateNickname` is false but `copyToProfile` is set to true, a copy of the card is made and added to the profile. A nickname is generated for the copy if one is not passed in, but no nickname is generated for the original card.

The form handler's `addToContainer` property determines whether the new payment group is added to the `PaymentGroupMapContainer` and made the default payment group; by default, this property is set to True. (Once the payment group is added to the `PaymentGroupMapContainer`, the user can use it when checking out the `Order`.)

`CreateCreditCardFormHandler` can optionally validate credit card information using the credit card's verification number. To validate credit cards, set the `validateCreditCard` property to `true`. Finally, the form handler's `copyToProfile` property determines whether the payment group is copied to the `Profile`; by default, this property is set to `true`.

After creating the credit card information, you can use the `UpdateCreditCardFormHandler` to deal with any changes the user makes to their credit card information. `UpdateCreditCardFormHandler` can update this information in any or all of three places, based on these properties:

- `updateContainer`—Update the credit card in the `PaymentGroupMapContainer`.
- `updateProfile`—Update the credit card in the profile.
- `updateOrder`—Update the credit card in the order.

- `Cer`

  This form handler creates an `InvoiceRequest` payment group and exposes it via a `getInvoiceRequest()` method, which makes it possible for users to edit its properties directly via JSP forms. Commerce provides an instance of `atg.commerce.order.purchase.CreateInvoiceRequestFormHandler`; it is located in Nucleus at `/atg/commerce/order/purchase/CreateInvoiceRequestFormHandler`.

  To create the `InvoiceRequest` payment group, the `handleNewInvoiceRequest()` method first invokes the `checkRequiredProperties()` method. By default, this method checks that a `poNumber` for the invoice has been specified and throws an exception if one has not been provided. The handle method then calls `createInvoiceRequest()`, which actually creates the payment group.

  The form handler's `invoiceRequestType` property determines the type of `InvoiceRequest` payment group to create; by default, this property is set to `invoiceRequest`. The form handler's `addToContainer` property determines whether the new payment group is added to the `PaymentGroupMapContainer` and made the default payment group; by default, this property is set to True. (Once the payment group is added to the `PaymentGroupMapContainer`, the user can use it when checking out the `Order`.) The form handler's `billingAddressPropertyName` determines the billing address `Profile` property to copy into the `InvoiceRequest`; by default, this property is set to `defaultBillingAddress`. Finally, the form handler's `invoiceRequestProperties` property determines what additional `Profile` properties to dynamically add to the `InvoiceRequest`.

You can also create `Profile`-derived `PaymentGroups` and add them to the `PaymentGroupMapContainer` by using the `PaymentGroupDroplet` servlet bean (class

atg.commerce.order.purchase.PaymentGroupDroplet). The input parameters passed into PaymentGroupDroplet determine what types of PaymentGroups are created (credit card, store credit, gift certificate) and whether the PaymentGroupMapContainer is cleared before they are created. For a detailed list of these input parameters, as well as its output parameters, open parameters, and a code example, see the *Adding Payment Information to Shopping Carts* section of the *Implementing Shopping Carts* chapter of the *ATG Commerce Guide to Setting Up a Store*.

To initialize the PaymentGroup objects, the service method of PaymentGroupDroplet calls initializeUserPaymentMethods(), which initializes one or more PaymentGroups for the current user and adds them to the PaymentGroupMapContainer. For each entry in PaymentGroupDroplet.paymentGroupTypes (which is supplied via an input parameter), its corresponding PaymentGroupInitializer is obtained from the ServiceMap in PaymentGroupDroplet.paymentGroupInitializers (keyed by PaymentGroup type). The initializePaymentGroups() method of the PaymentGroupInitializer is then used to initialize the PaymentGroup and add it to the PaymentGroupMapContainer.

Note that Commerce provides four implementations of the PaymentGroupInitializer interface. They are:

- CreditCardInitializer
- GiftCertificateInitializer
- StoreCreditInitializer
- InvoiceRequestInitializer

To use this framework with a new PaymentGroup type that you create, first, write a new PaymentGroupInitializer implementation. Its initializePaymentGroups() method should gather the user's PaymentGroups by type and add them to the PaymentGroupMapContainer referenced by the PaymentGroupFormHandler. For example, the StoreCreditInitializer queries the Claimable Repository for the user's StoreCredit PaymentGroups, instantiates objects for them, and then adds them to the PaymentGroupMapContainer. Second, register a Nucleus component for the new PaymentGroupInitializer implementation and add it to the ServiceMap in PaymentGroupDroplet.paymentGroupInitializers, which is keyed by PaymentGroup type. Finally, include the new PaymentGroup type in the PaymentGroupDroplet.paymentGroupTypes parameter on those site pages where the new PaymentGroup type is utilized.

Associating Payment Groups with an Order and Its Items

When the user has supplied the payment information for an Order, the PaymentGroupFormHandler can used to create and manage the associations between the PaymentGroups and the various parts of the Order. Any Order that has been successfully processed by the PaymentGroupFormHandler is ready for the next phase of the purchase process, which is typically order confirmation. Oracle ATG Web Commerce provides a request-scoped instance of atg.commerce.order.purchase.PaymentGroupFormHandler, which is located in Nucleus at /atg/commerce/order/purchase/PaymentGroupFormHandler.

The PaymentGroupFormHandler adds the PaymentGroups to the Order, adds the CommerceItems, ShippingGroups, tax, cost amount and cost remaining information to the PaymentGroups, validates the PaymentGroup information, and finally saves the Order in its present state to the Order Repository. If you'd prefer for items to be priced according to a pricelist rather than the default behavior provided by the pricing engine, set the priceListId property to the appropriate pricelist. When it is finished, the Order is ready to proceed to the next step in the purchase process, which typically is Order checkout. (See Submitting an Order for Checkout (page 294).)

The PaymentGroupFormHandler is composed of the following containers:

- atg.commerce.order.purchase.PaymentGroupMapContainer, which defines a Map of user-assigned PaymentGroup names to PaymentGroups. This container stores the user' potential PaymentGroups for the Order.

- atg.commerce.order.purchase.CommerceIdentifierPaymentInfoContainer, which defines a Map of CommerceIdentifiers to CommerceIdentifierPaymentInfo Lists. This container stores the user's CommerceIdentifierPaymentInfo objects for the Order.

Additionally, the PaymentGroupFormHandler uses the following helper classes:

- atg.commerce.order.purchase.CommerceIdentifierPaymentInfo, which represents the association between a CommerceIdentifier and its payment information and includes properties that allow the cost of a given quantity or even a single item to be spread across multiple payment groups. These objects store the information need to create payment Relationships for the Order.

- atg.commerce.order.purchase.PaymentGroupDroplet, which implements both the PaymentGroupMapContainer and CommerceIdentifierPaymentInfoContainer interfaces. The PaymentGroupDroplet servlet bean is used to initialize PaymentGroup objects and CommerceIdentifierPaymentInfo objects for use by the PaymentGroupFormHandler. The resulting collections of PaymentGroups and CommerceIdentifierPaymentInfo objects are exposed via the output parameters of the servlet bean. (For more information on using PaymentGroupDroplet to initialize PaymentGroup objects, see Creating Payment Groups (page 288). For more information on initializing CommerceIdentifierPaymentInfo objects, see below in this section.)

With these helper classes and containers, the PaymentGroupFormHandler adds the necessary PaymentGroups to the Order, validates them, and updates the Order. The following table describes the handle methods used in these processes:

Method	Description of Functionality
handleSplitPaymentInfos	This handle method is used when the user wants to split a particular CommerceIdentifierPaymentInfo by amount across multiple PaymentGroups. The handle method calls splitPaymentInfos(), which retrieves the list of CommeceIdentifierPaymentInfo objects from the CommerceIdentifierPaymentInfoContainer. The splitPaymentInfos() method then iterates through the list and calls splitCommerceIdentifierPaymentInfo() on each object. In turn, splitCommerceIdentifierPaymentInfo() calls splitCommerceIdentifierPaymentInfoByAmount() to split the CommerceIdentifierPaymentInfo object. The method creates a new CommerceIdentifierPaymentInfo object, adjusts the properties of both the existing and new objects, and adds the new object to the CommerceIdentifierPaymentInfoContainer. In a form, the user might request to split $50 of an original CommerceIdentifier amount of $100 to a separate payment method. This creates a separate CommerceIdentifierPaymentInfo object, and adjusts the amount of both the original and the new CommerceIdentifierPaymentInfo objects to add up to the original CommerceIdentifier total amount.

Method	Description of Functionality
`handleSpecifyDefault PaymentGroup`	This handle method is used to let the user specify a default `PaymentGroup` to use for payment. The method calls `specifyDefaultPaymentGroup()`, which sets the `defaultPaymentGroupName` in the `PaymentGroupMapContainer`. Setting the default `PaymentGroup` can facilitate simpler applications that permit only one `PaymentGroup` per `Order`, as well as advanced applications that apply a default `PaymentGroup` to any remaining `Order` amount not explicitly covered by other `PaymentGroups`.

Method	Description of Functionality
handleApplyPaymentGroups	This handle method adds the PaymentGroups to the Order. It is used when the user has supplied the necessary payment information for the Order and is ready to proceed with the next checkout phase. The handle method calls applyPaymentGroups(), which first calls PaymentGroupManager.removeAllPaymentGroupsFromOrder() to remove any existing PaymentGroups from the Order. This ensures a clean Order. Next, the applyPaymentGroups() method calls applyCommerceIdentifierPaymentInfo(), which applies the CommerceIdentierPaymentInfo objects to the Order. The applyCommerceIdentifierPaymentInfo() method iterates through the list of CommerceIdentifierPaymentInfo objects in the CommerceIdentifierPaymentInfoContainer. For each CommerceIdentifierPaymentInfo object, the associated PaymentGroup is retrieved and added to the Order (if it isn't already in the Order). Then the method retrieves the Relationship type of the current CommerceIdentifierPaymentInfo object and calls the appropriate method in the appropriate "Manager" to add the amount to the PaymentGroup. For more information, see the Assigning Costs to Payment Groups (page 249) section of the *Working With Purchase Process Objects* (page 223) chapter. Next, if the form handler's applyDefaultPaymentGroup property is True, then the applyPaymentGroups() method checks for a default payment group in the PaymentGroupMapContainer. If one exists, then the remaining order amount is added to the default payment group. Then, applyPaymentGroups() calls PaymentGroupManager.recalculatePaymentGroupAmount() to recalculate the payment groups. Next, the handle method calls runProcessValidatePaymentGroups() to validate the PaymentGroups in the Order. This executes the payment validation pipeline specified in PaymentGroupFormHandler.validatePaymentInformationChainId; by default, this property is to moveToConfirmation. The moveToConfirmation pipeline both prices and validates a given Order. For more information, see Appendix F, *Pipeline Chains* (page 699). Finally, the handle method calls OrderManager.updateOrder() to save the Order in its present state to the Order Repository. For more information on OrderManager.updateOrder() and the updateOrder pipeline that it executes, see the Updating an Order with the OrderManager (page 278) subsection of *Saving Orders* in this chapter.

As previously mentioned, the `PaymentGroupDroplet` servlet bean is used to initialize `CommerceIdentifierPaymentInfo` objects and add them to the `CommerceIdentifierPaymentInfoContainer`, so they can be used by the `PaymentGroupFormHandler`. To initialize the `CommerceIdentifierPaymentInfo` objects, the service method of `PaymentGroupDroplet` calls `initializePaymentInfos()`, which creates and initializes `CommerceIdentifierPaymentInfo` objects for the `Order`, as well as the `Order`'s `CommerceItems`, `ShippingGroups`, and tax. These objects are then added to the `CommerceIdentifierPaymentInfoContainer`.

The input parameters passed into `PaymentGroupDroplet` determine whether the `CommerceIdentifierPaymentInfo` objects are created and whether the `CommerceIdentifierPaymentInfoContainer` is cleared before they are created. For a detailed list of these input parameters, as well as its output parameters, open parameters, and a code example, see the *Adding Payment Information to Shopping Carts* section of the *Implementing Order Retrieval* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Submitting an Order for Checkout

The `CommitOrderFormHandler` (class `atg.commerce.order.purchase.CommitOrderFormHandler`) submits the user's current `Order` for checkout. Oracle ATG Web Commerce includes an instance of `CommitOrderFormHandler`, which is located in Nucleus at `/atg/commerce/order/purchase/CommitOrderFormHandler`.

The form handler's `handleCommitOrder()` method ensures that the user is not trying to double-submit the order by checking if the ID of the current `Order` is equal to the ID of the user's last `Order` (in `ShoppingCart.last`). If the IDs are not equal, then the current `Order` can be submitted. The handle method then calls the `OrderManager.processOrder()` method, which executes the `processOrder` pipeline. (See Checking Out an Order (page 294) below.)

If no errors occur during the validation or checkout of the `Order`, then `handleCommitOrder()` sets the submitted `Order` as the user's last `Order` in `ShoppingCart.last`, and it constructs a new, empty `Order` and sets it as the user's current `Order` in `ShoppingCart.current`.

Checking Out an Order

Order processing occurs when a customer has supplied all the necessary information for the `Order` and has submitted it for checkout. The processing of an `Order` begins with a call to `OrderManager.processOrder()`, which calls into the `PipelineManager` to execute the `processOrder` pipeline. The `processOrder` pipeline first validates the `Order` and then processes it. Note that, by default, Oracle ATG Web Commerce does not process an incomplete `Order`. To allow the processing of incomplete `Orders`, you must modify the pipeline accordingly.

The `PipelineManager` Nucleus component for Commerce is located at `/atg/commerce/PipelineManager`. The related XML configuration file is defined in `<ATG10dir>/DCS/config/atg/commerce/commercepipeline.xml`.

Note: You can use the credit card information listed below to process payments during testing of the order checkout process. The expiration date for all cards can be any date in the future.

- Visa: 4111111111111111
- MasterCard: 5555555555554444
- American Express: 378282246310005
- Discover: 6011111111111117

The following table describes the processors in the `processOrder` pipeline. They are listed in order of execution.

PipelineLink name	Description
`executeValidateForCheckoutChain`	Executes the `validateForCheckout` pipeline (Refer to the next table in this section.) The `atg.commerce.order.processor.ProcExecuteChain` class implements this functionality. Its properties file defines the execution chain to run in the `chainToRun` property. By default, this property is set to `validateForCheckout`. If any errors occur during the execution of this processor, execution of the `processOrder` pipeline stops.
`executeApproveOrderChain`	Executes the `approveOrder` pipeline, which begins the approval process. The `atg.commerce.order.processor.ProcExecuteChain` class implements this functionality. See the *Managing the Order Approval Process* (page 445) chapter for more information.
`stopChainIfOrderRequiresApproval`	Checks the state of the `Order`. If the `Order` requires approval and has not been approved yet, execution of the `processOrder` pipeline stops. If the `Order` required approval and has been approved, the `processOrder` pipeline continues with `executeValidatePostApprovalChain`. If the `Order` did not require approval, the pipeline continues with `executeValidateNoApprovalChain`. The `atg.commerce.order.processor.ProcDispatchOnOrderState` class implements this functionality. See the *Managing the Order Approval Process* (page 445) chapter for more information.
`executeValidatePostApprovalChain`	For `Orders` that have been approved. This processor executes the `validatePostApproval` pipeline, which revalidates information the approver may have changed. Specifically, the pipeline revalidates payment information (all payment groups and cost centers) and checks that all `Order` and shipping costs are accounted for. The `atg.commerce.order.processor.ProcExecuteChain` class implements this functionality. See the *Managing the Order Approval Process* (page 445) chapter for more information.

PipelineLink name	Description
executeValidateNoApprovalChain	For Orders that didn't require approval. This processor executes the validateNoApproval pipeline, which validates information intentionally skipped by the executeValidateForCheckout pipeline processor. Specifically, by default the pipeline validates InvoiceRequests, which are intentionally skipped by the validateForCheckout pipeline until it is determined that the Order requires approval. The atg.commerce.order.processor.ProcExecuteChain class implements this functionality. See the *Managing the Order Approval Process* (page 445) chapter for more information.
checkForExpiredPromotions	Checks that expired promotions are not used in Order that is being checked out. The atg.commerce.order.processor.ProcCheckForExpiredPromotions class implements this functionality.
removeEmptyShippingGroups	Removes any empty (unused) shipping groups from the Order. An empty ShippingGroup is one without any CommerceItemRelationships. The atg.commerce.order.processor.ProcRemoveEmptyShippingGroups class implements this functionality.
removeEmptyPaymentGroups	Removes any empty (unused) payment groups from the Order. An empty PaymentGroup is one without any CommerceItemRelationships, ShippingGroupRelationships, and OrderRelationships. The atg.commerce.order.processor.ProcRemoveEmptyPaymentGroups class implements this functionality.
createImplicitRelationships	Validates the special case of an Order having one ShippingGroup and/or one PaymentGroup and no Relationships. In this situation, the processor actually creates relationships between all the CommerceItems and the ShippingGroup. It also creates a relationship between the PaymentGroup and the Order. The atg.commerce.order.processor.ProcCreateImplicitRelationships class implements this functionality.
setPaymentGroupAmount	Determines and sets the amount to charge in each PaymentGroup based on the Relationships in the Order. The atg.commerce.order.processor.ProcSetPaymentGroupAmount class implements this functionality.

PipelineLink name	Description
`setCostCenterAmount`	Determines and sets the amount to assign to each cost center in the `Order`. The `atg..order.processor.ProcSetCostCenterAmount` class implements this functionality. See the *Implementing Cost Centers* chapter of the *ATG Commerce Guide to Setting Up a Store* for more information.
`moveUsedPromotions`	Moves all promotions that a customer used in the `Order` to the `usedPromotions` list in the customer's profile. If the promotion is a single-use promotion, it is also removed from the `activePromotions` list and added to the customer's `inactivePromotions` list. The `atg.commerce.order.processor.ProcMoveUsedPromotions` class implements this functionality.
`authorizePayment`	Authorizes all `PaymentGroups` (`CreditCard`, or `GiftCertificate`, or `StoreCredit`) in the `Order`. The `atg.commerce.order.processor.ProcAuthorizePayment` class implements this functionality. **Note:** For information on how to prevent the authorization of a user's credit card under certain circumstances, such as when the items in the `Order` don't exist in inventory, see Preventing Payment Authorization for Unfulfilled Orders (page 299) later in this section.
`updateGiftRepository`	Updates the Gift List Repository to reflect the purchase of any gifts in the order. The `atg.commerce.order.processor.ProcUpdateGiftRepository` class implements this functionality
`sendGiftPurchasedMessage`	Sends a message to the messaging system that describes the gifts that were purchased. The message can then be used to execute scenarios. The `atg.commerce.order.processor.ProcSendGiftPurchasedMessage` class implements this functionality.
`addOrderToRepository`	This processor first calls `OrderManager.updateOrder()` to save the `Order` to the Order Repository. (See Updating an Order with the OrderManager (page 278).) The `atg.commerce.order.processor.ProcAddOrderToRepository` class implements this functionality.
`sendPromotionUsedMessage`	Sends a message to the messaging system that describes the promotions that were used in the `Order`. The message can then be used to execute scenarios. The `atg.commerce.order.processor.ProcSendPromotionUsedMessage` class implements this functionality.

PipelineLink name	Description
sendFulfillmentMessage	Sends a message that includes the Order to the fulfillment system. This message indicates to the fulfillment system that it can fulfill the Order. The atg.commerce.order.processor.ProcSendFulfillmentMessage class implements this functionality.

The first processor in the processOrder pipeline, named executeValidateForCheckoutChain, in turn executes the validateForCheckout pipeline. The following table describes the processors in the validateForCheckout pipeline. They are listed in order of execution.

PipelineLink name	Description
ValidateOrderForCheckout	Verifies that the Order contains at least one CommerceItem, ShippingGroup, and PaymentGroup. The atg.commerce.order.processor.ProcValidateOrderForCheckout class implements this functionality.
VerifyOrderAddresses	Verifies the addresses for shipping groups and payment groups. The atg.commerce.order.processor.ProcVerifyOrderAddresses class implements this functionality.
validateShippingGroupsForCheckout	Verifies that all CommerceItems in the Order are assigned to a ShippingGroup. The processor also verifies that all required fields have been supplied to all ShippingGroups in the Order. The atg.commerce.order.processor.ProcValidateShippingGroupsForCheckou class that implements this functionality is.
creditCardModCheck	Verifies that the credit card number and expiration date are valid. The credit card is not authorized by a processing system at this point. (See the previous table, which describes the processors in the processOrder pipeline.) Rather, the processor applies a simple algorithm to the credit card number to determine if it is valid. It then checks the date to determine if the card has expired. The atg.commerce.order.processor.ProcCreditCardModCheck class implements this functionality.
validatePaymentGroupsForCheckout	Verifies that all CommerceItems in the Order can account for their costs, which means that the processor verifies all PaymentGroupCommerceItemRelationships. It also verifies that all required fields have been supplied to all PaymentGroups in the Order. The atg.commerce.order.processor.ProcValidatePaymentGroupsForCheckout class implements this functionality.

PipelineLink name	Description
validateShippingCostsForCheckout	Verifies that all ShippingGroups can account for their costs, which means that the processor verifies all PaymentGroupShippingGroupRelationships. The atg.commerce.order.processor.ProcValidateShippingCostsForC class implements this functionality.
validateOrderCostsForCheckout	Verifies the Order can account for its cost, which means the processor verifies all PaymentGroupOrderRelationships. The atg.commerce.order.processor.ProcValidateOrderCostsForChec class implements this functionality.
validateHandlingInstructionsForCheckout	Verifies that all HandlingInstructions are assigned to a valid ShippingGroup and CommerceItem combination. The atg.commerce.order.processor.ProcValidateHandlingInstructi class implements this functionality.
validateCurrencyCodes	Validates that all the currency codes in the PriceInfo objects are the same. Also validates that all the items, shipping, tax, and order have been priced. The atg.commerce.order.processor.ProcValidateCurrencyCodes class implements this functionality.

For more information about pipelines, the PipelineManager, and the transactional modes and transitions of the processors in the processOrder and validateForCheckout pipelines, see Appendix F, *Pipeline Chains* (page 699).

Preventing Payment Authorization for Unfulfilled Orders

The processOrder pipeline performs several checkout functions, including authorizing a user's credit card for the amount of the order. Under certain circumstances, you may want to prevent the authorization of a credit card, for example, when the order contains items that do not exist in inventory. In this example, the user could be alerted of the lack of inventory and allowed to modify the order before checkout.

You can modify the processOrder pipeline to prevent credit card authorization under certain conditions. To do so, you need to modify the pipeline to include a branch. One processor in the pipeline should check that the items being purchased do exist in inventory. If they do, then the Order should continue through the pipeline for checkout (and authorization of the user's credit card). If the items do not exist in inventory, then the Order should branch to an alternate pipeline that does not authorize the user's credit card. Instead, the pipeline might redirect the user to a page that indicates which items could not be allocated from inventory and allows the user to change the Order. The important concept is that the processOrder pipeline branches so that the user's credit card is not authorized.

As previously mentioned, the PipelineManager Nucleus component for Oracle ATG Web Commerce is located at /atg/commerce/PipelineManager. In Commerce, the related XML configuration file is defined in <ATG10dir>/DCS/config/atg/commerce/commercepipeline.xml.

For information on how to set up a branching pipeline, see the *Processor Chains and the Pipeline Manager* (page 363) chapter.

Processing Payment of Orders

The PaymentManager manages the authorization, debit, and credit of PaymentGroups in an Order, and it tracks the results of those payment operations using PaymentStatus objects.

This section provides information on how the payment of orders is processed in Oracle ATG Web Commerce and describes how to extend the system to add functionality or to support a new payment method. It includes the following sections:

Overview of the Payment Process

When the PaymentManager's authorize/credit/debit method is called, it takes the Order and a single PaymentGroup or List of PaymentGroups as parameters. It then calls its authorize/credit/debit method that takes an additional amount parameter, passing in the amount in the current PaymentGroup. This second authorize/credit/debit method performs the actual payment operation for the current PaymentGroup by looking up the pipeline appropriate for the current PaymentGroup class type and then calling that pipeline.

To obtain the appropriate pipeline to run, the authorize/credit/debit method calls getXXXChainName(PaymentGroup), for example, getCreditChainName(PaymentGroup). In turn, this method calls getChainName(PaymentGroup), which uses the class name of the given PaymentGroup as the key to look up the appropriate pipeline to run in PaymentManager.paymentGroupToChainNameMap. This property stores a map of PaymentGroup class types to the names of the pipelines that perform the payment actions for the payment methods. By default, this property is configured as follows:

```
paymentGroupToChainNameMap=\
  atg.commerce.order.CreditCard=creditCardProcessorChain,\
  atg.commerce.order.GiftCertificate=giftCertificateProcessorChain,\
  atg.commerce.order.StoreCredit=storeCreditProcessorChain
```

Thus, the creditCardProcessorChain pipeline handles authorization, debit, and credit work for the atg.commerce.order.CreditCard class, the giftCertificateProcessorChain pipeline handles authorization, debit, and credit work for the atg.commerce.order.GiftCertificate class, and so on.

By default, each of the pipelines in PaymentManager.paymentGroupToChainNameMap is composed of two processors. The first processor aggregates the necessary information for performing the requested payment action (for example, CREDIT) and creates an XXXInfo object (for example, CreditCardInfo) for use in that action. The second processor performs the actual operation – authorizing, debiting, or crediting the appropriate payment method. Note that while a single pipeline exists to perform authorize, debit, *and* credit actions for a single PaymentGroup type, you can split these actions into separate pipelines if your processing needs for a given payment action are unusual.

Once the appropriate pipeline to run has been obtained (for example, the creditCardProcessorChain pipeline), the authorize/credit/debit method calls into the PaymentPipelineManager to execute the pipeline. It passes in the PaymentManagerPipelineArgs Dictionary object as an argument to the runProcess() method of the pipeline. This Dictionary object contains the information required to perform the transaction, which is as follows:

- `Order`
- `PaymentManager`
- `PaymentGroup`
- Payment amount
- Action (`PaymentManagerAction.AUTHORIZE`, `PaymentManagerAction.CREDIT`, or `PaymentManagerAction.DEBIT`)
- Generic Info object (for example, `CreditCardInfo` and `GiftCertificateInfo`))
- `PaymentStatus`

The `PaymentStatus` object represents the results of the authorize, credit, or debit transaction performed by the pipeline; it contains properties such as `amount`, `errorMessage`, `transactionId`, `transactionSuccess`, and `transactionTimestamp`. When the `PaymentManager`'s `authorize/credit/debit` method is called, the method performs the payment operation and then adds a `PaymentStatus` object to the given `PaymentGroup`. The `PaymentStatus` object is discussed in more detail at the end of this section.

Note: The `PaymentManager` is also used by the Fulfillment system. For information about the Fulfillment system, see the *Configuring the Order Fulfillment Framework* (page 413) chapter.

Extending the Payment Operations of a Payment Method

Sometimes you may find that you need to extend the way a given payment operation works. For example, you may have an unusual credit operation that you want to perform on credit cards.

This section provides information on how to extend the way payment operations work for a given payment method, using the `CreditCard` payment method as an example. The process to extend the payment operations of the `StoreCredit` payment method, the `GiftCertificate` payment method, and the `InvoiceRequest` payment method works in the same fashion.

As described in the previous section, the `PaymentManager.properties` file at `/atg/commerce/payment/PaymentManager` is configured to map `CreditCard` objects to the `creditCardProcessorChain` pipeline. Like all of the default payment pipelines, `creditCardProcessorChain` is composed of two processors. The first processor – `CreateCreditCardInfo` -- aggregates the necessary information for performing the requested payment action (for example, CREDIT) and creates an `XXXInfo` object (for example, `CreditCardInfo`) for use in that action. The second processor – `ProcessCreditCard` --performs the actual operation – authorizing, debiting, or crediting the appropriate payment method.

The `ProcessCreditCard` processor is located in Nucleus at `/atg/commerce/payment/processor/ProcessCreditCard`, and it is instantiated from class `atg.commerce.payment.processor.ProcProcessCreditCard`, which extends `atg.commerce.payment.processor.ProcProcessPaymentGroup`. The `ProcessCreditCard` processor authorizes, debits, and credits a `CreditCard` `PaymentGroup` by calling through to a `CreditCardProcessor` object to perform the actual operations. The specific object used to perform the actual operations is retrieved from `PaymentManager.creditCardProcessor`; this property points to an object instantiated from a class that implements the `atg.payment.creditCard.CreditCardProcessor` interface.

To change the way credit cards are operated on, write a new class that implements the `CreditCardProcessor` interface and provides the additional functionality your sites require, then create and configure an instance of the new class in Nucleus, and finally change the `PaymentManager.creditCardProcessor` property to point to the new component.

Extending the Payment Process to Support a New Payment Method

If your sites require it, you can extend the payment process to support a new payment method. This section provides information on how to create and support a new payment method, using an example called the `StorePoints` payment method. In this example, customers earn store points when they purchase items, and they can later redeem those points on other purchases. The `StorePoints` system is just one example of the many new payment methods you can implement.

The `StorePoints` system is implemented using the following basic steps. Detailed instructions are provided in the referenced subsections that follow.

1. Create the new `StorePoints PaymentGroup`. See Creating a New PaymentGroup (page 303).

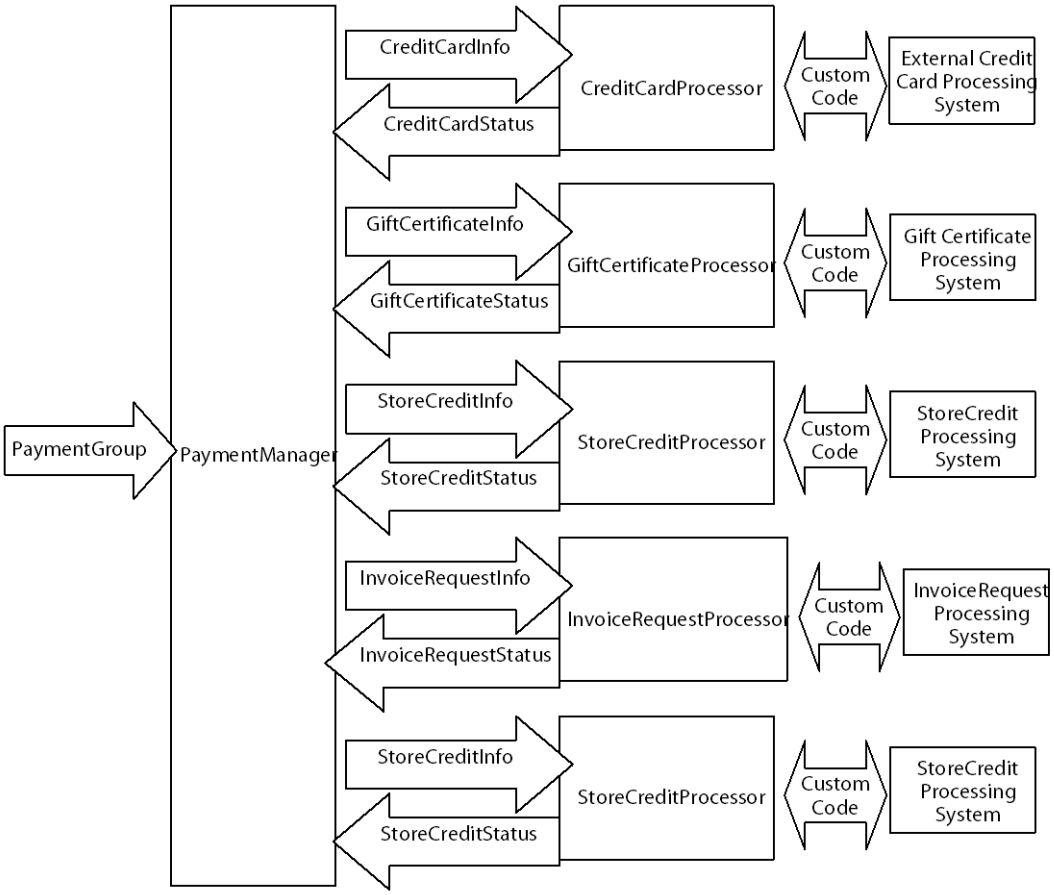
2. Create a new repository item type for the `StorePointsPaymentGroup`. Associate the new item type to the new class by adding an entry to the `beanNameToItemDescriptorMap` property of the `OrderTools` component. Add the new class to the `paymentTypeClassMap`. See Integrating a New Commerce Object: Using an Existing Item Descriptor (page 343).

3. Implement the payment processors involved in operating on the `StorePoints PaymentGroup`. See Implementing Processors for the New PaymentGroup (page 304).

4. Define the pipeline that processes the `StorePoints PaymentGroup` for authorization, debits, and credits, and configure the `PaymentManager` to invoke it when appropriate. See Integrating the New Payment Processors into the PaymentManager (page 312).

5. Extend the `Order` validation process to validate the `StorePoints PaymentGroup` during checkout. See Extending Order Validation to Support New Payment Methods (page 314) in the next section.

The following diagram shows the relationships between the objects involved in processing a payment transaction. In Oracle ATG Web Commerce, by default the `CreditCard`, `GiftCertificate`, `StoreCredit`, and `InvoiceRequest` systems are provided.



Creating a New PaymentGroup

The first step in creating and supporting a new `StorePoints` payment method is to create a new `PaymentGroup` named `StorePoints`. The new `PaymentGroup` allows you to distinguish `StorePoints` from other payment groups and to store data relating to the store points system. The following code sample is an example of the `StorePoints PaymentGroup`.

```
package store.some.package;

import atg.commerce.order.*;

public class StorePoints extends PaymentGroupImpl
{
  public StorePoints() {
  }

  public String getUserId() {
    return (String) getPropertyValue("userId");
  }

  public void setUserId(String pUserId) {
    setPropertyValue("userId", pUserId);
  }
```

```
  public int getNumberOfPoints() {
    return ((Integer) getPropertyValue("numberOfPoints")).intValue();
  }

  public void setNumberOfPoints(int pNumberOfPoints) {
    setPropertyValue("numberOfPoints", new Integer(pNumberOfPoints));
  }
}
```

Implementing Processors for the New PaymentGroup

As previously mentioned, the default payment pipelines are composed of two processors. The first processor aggregates the necessary information for performing the requested payment action (for example, CREDIT) and creates an `XXXInfo` object (for example, `CreditCardInfo`) for use in that action. The second processor actually performs the operation – authorizing, debiting, or crediting the appropriate payment method. For example, the `creditCardProcessorChain` pipeline is composed of the `CreateCreditCardInfo` processor (class `atg.commerce.payment.processor.ProcCreateCreditCardInfo`) and the `ProcessCreditCard` processor (class `atg.commerce.payment.processor.ProcProcessCreditCard`). The `ProcessCreditCard` processor calls through to a `CreditCardProcessor` object to perform the actual operations. The specific object used to perform the actual operations is retrieved from `PaymentManager.creditCardProcessor`, which points to an object instantiated from a class that implements the `atg.payment.creditcard.CreditCardProcessor` interface.

For the `StorePoints PaymentGroup`, you need to implement similar processors -- a pipeline processor to create the `XXXInfo` object for the `StorePoints PaymentGroup`, a second pipeline processor to authorize, debit, and credit the `StorePoints PaymentGroup`, and a processor that implements a `StorePointsProcessor` interface and actually performs the payment operations.

First, write the `StorePointsProcessor` interface that defines the `authorize()`, `debit()`, and `credit()` methods for the `StorePoints PaymentGroup`, as shown in the following code example. Note that the `authorize`, `debit`, and `credit` methods of the `StorePointsProcessor` interface all return a `StorePointsStatus` object, which represents the transaction on the `StorePoints PaymentGroup`. This object is discussed in more detail later in this section.

```
package store.some.package;

import atg.payment.*;

public interface StorePointsProcessor
{
  /**
   * Authorize the amount in StorePoints
   *
   * @param pStorePointsInfo the StorePointsInfo reference which contains
   *        all the authorization data
   * @return a StorePointsStatus object detailing the results of the
   *        authorization
   */
  public StorePointsStatus authorize(StorePointsInfo pStorePointsInfo);

  /**
   * Debit the amount in StorePoints after authorization
   *
   * @param pStorePointsInfo the StorePointsInfo reference which contains
   *        all the debit data
   * @param pStatus the StorePointsStatus object which contains
```

```
   *        information about the transaction. This should be the object
   *        which was returned from authorize().
   * @return a StorePointsStatus object detailing the results of the debit
   */
  public StorePointsStatus debit(StorePointsInfo pStorePointsInfo,
                                 StorePointsStatus pStatus);

  /**
   * Credit the amount in StorePoints after debiting
   *
   * @param pStorePointsInfo the StorePointsInfo reference which contains
   *        all the credit data
   * @param pStatus the StorePointsStatus object which contains
   *        information about the transaction. This should be the object
   *        which was returned from debit().
   * @return a StorePointsStatus object detailing the results of the
   *        credit
   */
  public StorePointsStatus credit(StorePointsInfo pStorePointsInfo,
                                  StorePointsStatus pStatus);

  /**
   * Credit the amount in StorePoints outside the context of an Order
   *
   * @param pStorePointsInfo the StorePointsInfo reference which contains
   *        all the credit data
   * @return a StorePointsStatus object detailing the results of the
   *        credit
   */
  public StorePointsStatus credit(StorePointsInfo pStorePointsInfo);
}
```

Second, write an implementation of the `StorePointsProcessor` interface named `StorePointsProcessorImpl`. `StorePointsProcessorImpl` must work with the resources needed to carry out the transactions. For example, if the customer's points data are stored in a database table, then its methods must operate against that table, reading and writing values to reflect the operation. For a different custom payment method, the implementation must work with whatever 3rd-party resources are needed to carry out the transactions.

The following code sample is taken from the `StorePointsProcessorImpl` class, an example of an implementation of the `StorePointsProcessor` interface. You can assume that `StorePointsProcessorImpl` extends `GenericService` and, therefore, can use standard Oracle ATG Web Commerce logging calls.

Note in the code sample that the authorize, debit, and credit methods of `StorePointsProcessorImpl` all return a `PaymentStatus` object, which represents the results of transaction performed by the pipeline. Recall that a `PaymentStatus` object contains properties such as `amount`, `errorMessage`, `transactionId`, `transactionSuccess`, and `transactionTimestamp`. It is discussed in more detail later in this section.

```
/**
   * This method will obtain the <code>StorePointsInfo</code> object from
   * the pParams parameter and invoke the
   * {@link #authorize<code>authorize</code>} method.
   *
   * @param pParams PaymentManagerPipelineArgs object which contains the
   *  StorePointsInfo object.
   * @return a PaymentStatus object that will detail the authorize details
   * @exception CommerceException if an error occurs
   */
```

```
  public PaymentStatus authorizePaymentGroup(PaymentManagerPipelineArgs
pParams)
    throws CommerceException
  {
    StorePointsInfo spi = null;

    try {
      spi = (StorePointsInfo)pParams.getPaymentInfo();
    }
    catch (ClassCastException cce) {
      if (isLoggingError())
        logError("Expecting class of type StorePointsInfo but got: " +
                  pParams.getPaymentInfo().getClass().getName());

      throw cce;
    }
    return authorize(spi);
  }
/**
   * This method will obtain the <code>StorePointsInfo</code> object from
   * the pParams parameter and invoke the {@link #debit<code>debit</code>}
   * method.
   *
   * @param pParams PaymentManagerPipelineArgs object which contains the
   *  StorePointsInfo and StorePointsStatus objects.
   * @return a PaymentStatus object that will detail the debit details
   * @exception CommerceException if an error occurs
   */
  public PaymentStatus debitPaymentGroup(PaymentManagerPipelineArgs
pParams)
    throws CommerceException
  {
    StorePointsInfo spi = null;

    try {
      spi = (StorePointsInfo)pParams.getPaymentInfo();
    }
    catch (ClassCastException cce) {
      if (isLoggingError())
        logError("Expecting class of type StorePointsInfo but got: " +
                  pParams.getPaymentInfo().getClass().getName());

      throw cce;
    }

    StorePointsStatus authStatus = null;

    PaymentGroup pg = pParams.getPaymentGroup();
    try {
      authStatus = (StorePointsStatus)
pParams.getPaymentManager().getLastAuthorizationStatus(pg);
    }
    catch (ClassCastException cce) {
      if (isLoggingError()) {
        String authStatusClassName =
pParams.getPaymentManager().getLastAuthorizationStatus(pg).getClass().getN
ame();
        logError("Expecting class of type StorePointsStatus but got: " +
                  authStatusClassName);
      }
```

```
      throw cce;
    }

    return debit(spi, authStatus);
  }

  /**
   * This method will obtain the <code>StorePointsInfo</code> object from
   * the pParams parameter and invoke the
   * {@link #credit<code>credit</code>} method.
   *
   * @param pParams PaymentManagerPipelineArgs object which contains the
   *  StorePointsInfo, PaymentGroup and StorePointsStatus object.
   * @return a PaymentStatus object that will detail the credit details
   * @exception CommerceException if an error occurs
   */
  public PaymentStatus creditPaymentGroup(PaymentManagerPipelineArgs
pParams)
    throws CommerceException
  {
    StorePointsInfo spi = null;

    try {
      spi = (StorePointsInfo)pParams.getPaymentInfo();
    }
    catch (ClassCastException cce) {
      if (isLoggingError())
        logError("Expecting class of type StorePointsInfo but got: " +
                 pParams.getPaymentInfo().getClass().getName());
      throw cce;
    }

    StorePointsStatus debitStatus = null;

      PaymentGroup pg = pParams.getPaymentGroup();
    try {
      debitStatus = (StorePointsStatus)
pParams.getPaymentManager().getLastDebitStatus(pg);
    }
    catch (ClassCastException cce) {
      if (isLoggingError()) {
        String debitStatusClassName =
pParams.getPaymentManager().getLastDebitStatus(pg).getClass().getName();
        logError("Expecting class of type StorePointsStatus but got: " +
                 debitStatusClassName);
      }

      throw cce;
    }

    return credit(spi, debitStatus);
  }
```

Third, implement a pipeline processor that performs the payment transactions for the `StorePoints PaymentGroup` by calling through to `StorePointsProcessorImpl`. You might call this pipeline processor class `ProcProcessStorePoints`. Because the implementation will be called within the context of a pipeline, it must also implement the `atg.service.pipeline.PipelineProcessor` interface. Oracle ATG Web Commerce provides an abstract class that implements both the `PipelineProcessor` interface and several other helper methods that determine what action

is requested (authorize, debit, or credit) and then dispatch to the appropriate method call. This abstract class is `atg.commerce.payment.processor.ProcProcessPaymentGroup`. By extending `ProcProcessPaymentGroup`, you only need to define three abstract methods: `authorizePaymentGroup()`, `debitPaymentGroup()` and `creditPaymentGroup()`. These methods should call through to their respective methods in the `StorePointsProcessorImpl` object, passing in the data from the `PaymentManagerPipelineArgs` object that is supplied as a parameter. Additionally, `ProcProcessStorePoints` should include an additional property named `storePointsProcessor` that can be set to the `StorePointsProcessor` object that actually performs the payment operations. In this example, `ProcProcessStorePoints.storePointsProcessor` would be set `StorePointsProcessorImpl`.

Recall from the previous code example of `StorePointsProcessorImpl` that the `StorePoints PaymentGroup` itself is not passed as a parameter to the `StorePointsProcessorImpl` processor. This keeps the payment processors independent of the commerce objects in Commerce. Instead, before the `ProcProcessStorePoints` pipeline processor is invoked, a previous pipeline processor must aggregate the necessary information for performing the requested payment action, create an `XXXInfo` object for use in that action, and finally add the `XXXInfo` object to the `PaymentManagerPipelineArgs Dictionary` object. The `Dictionary` object is then passed as an argument "downstream" to the `ProcProcessStorePoints` pipeline processor and on to the `StorePointsProcessorImpl` processor.

In this `StorePoints` example, the `XXXInfo` object might be called `StorePointsInfo`, and the processor that creates it might be called `ProcCreateStorePointsInfo`. The `StorePointsInfo` object must hold all of the data required by the methods in `StorePointsProcessorImpl`. It might hold a user ID (`Profile` ID) and the number of points for the operation. The following code sample is an example of the `StorePointsInfo` class.

```
package store.some.package;

public class StorePointsInfo
{
  public StorePointsInfo() {
  }

  private String mUserId = null;
  public String getUserId() {
    return mUserId;
  }
  public void setUserId(String pUserId) {
    mUserId = pUserId;
  }

  private int mNumberOfPoints = 0;
  public int getNumberOfPoints() {
    return mNumberOfPoints;
  }
  public void setNumberOfPoints(int pNumberOfPoints) {
    mNumberOfPoints = pNumberOfPoints;
  }
}
```

Next, implement the `ProcCreateStorePointsInfo` processor that must construct the `StorePointsInfo` object and add it to the `PaymentManagerPipelineArgs Dictionary` object. As with the `StorePointsProcessorImpl` class, the `ProcCreateStorePointsInfo` class must implement the `atg.service.pipeline.PipelineProcessor` interface because the implementation will be called within the context of a pipeline. The following code sample is an example of the `ProcCreateStorePointsInfo` class.

```
package store.some.package;
```

```
import atg.nucleus.GenericService;
import atg.service.pipeline.PipelineProcessor;
import atg.service.pipeline.PipelineResult;
import atg.commerce.order.*;
import atg.commerce.payment.*;

/**
 * This pipeline processor element is called to create generic
 * StorePointsInfo objects from instances of the StorePoints
 * payment group. It places them into the pipeline argument dictionary so
 * that downstream pipeline processors can retrieve them by calling
 * <code>PaymentManagerPipelineArgs.getPaymentInfo()</code>.
 *
 * <p>This processor is designed so that the StorePointsInfo class can
 * easily be extended. See
 * {@link #setStorePointsInfoClass "<code>setStorePointsInfoClass</code>"}
 * and
 * {@link #addDataToStorePoints "<code>addDataToStorePointsInfo</code>"}
 * for more information.
 *
 */

public class ProcCreateStorePointsInfo
  extends GenericService
  implements PipelineProcessor
{

  /** The possible return value for this processor. **/
  public static final int SUCCESS = 1;

  //-------------------------------------------------------------------------
  // property: StorePointsInfoClass

  String mStorePointsInfoClass = "store.some.package.StorePointsInfo";

  /**
   * Return the class to instantiate when creating a new StorePointsInfo
   * object.
   **/

  public String getStorePointsInfoClass() {
    return mStorePointsInfoClass;
  }

  /**
   * Specify the class to instantiate when creating a new StorePointsInfo
   * object.  If the <code>StorePointsInfo</code> class is extended to
   * include more information, this property can be changed to reflect the
   * new class.
   **/

  public void setStorePointsInfoClass(String pStorePointsInfoClass) {
    mStorePointsInfoClass = pStorePointsInfoClass;
  }

  //-------------------------------------------------------------------------

  /**
   * This method populates the <code>StorePointsInfo</code> object with
```

```
   * data. If the additional data is required, a subclass of
   * <code>StorePointsInfo</code> can be created with additional
   * properties, the <code>storePointsInfoClass</code> property can be
   * changed to specify the new class, and this method can be overridden
   * to add data for the new properties (or another pipeline processor
   * could be added after this processor to populate the additional
   * properties).
   *
   * @param pOrder
   *    The order being paid for.
   * @param pPaymentGroup
   *    The payment group being processed.
   * @param pAmount
   *    The amount being authorized, debited, or credited
   * @param pParams
   *    The parameter dictionary passed to this pipeline processor
   * @param pStorePointsInfo
   *    An object that holds information understood by the store
   *    points payment processor.
   **/

  protected void addDataToStorePointsInfo(Order pOrder,
       StorePoints pPaymentGroup, double pAmount,
       PaymentManagerPipelineArgs pParams, StorePointsInfo
       pStorePointsInfo)
  {
    pStorePointsInfo.setUserId(pPaymentGroup.getUserId());
    pStorePointsInfo.setNumberOfPoints(pPaymentGroup.getNumberOfPoints());
  }

  //-----------------------------------------------------------------------

  /**
   * Factory method to create a new StorePointsInfo object.  The class
   * that is created is that specified by the
   * <code>storePointsInfoClass</code> property, and must be a subclass
   * of <code>store.some.package.StorePointsInfo</code>
   *
   * @return
   *   An object of the class specified by
   *   <code>storePointsInfoClass</code>
   * @throws Exception
   *   if any instantiation error occurs when creating the info object
   **/

  protected StorePointsInfo getStorePointsInfo()
    throws Exception
  {
    if (isLoggingDebug())
      logDebug("Making a new instance of type: " +
getStorePointsInfoClass());

    StorePointsInfo spi = (StorePointsInfo)
      Class.forName(getStorePointsInfoClass()).newInstance();

    return spi;
  }

  //-----------------------------------------------------------------------
```

```
  /**
   * Generate a StorePointsInfo object of the class specified by
   * <code>StorePointsInfoClass</code>, populate it with data from a
   * <code>StorePoints</code> payment group by calling
   * <code>addDataToStorePointsInfo</code>, and add it to the pipeline
   * argument dictionary so that downstream pipeline processors can access
   * it.
   *
   * @param pParam
   *   Parameter dictionary of type PaymentManagerPipelineArgs.
   * @param pResult
   *   Pipeline result object, not used by this method.
   * @return
   *   An integer value used to determine which pipeline processor is
   *   called next.
   * @throws Exception
   *   If any error occurs creating or populating the store points info
   *   object.
   **/

  public int runProcess(Object pParam, PipelineResult pResult)
    throws Exception
  {
    PaymentManagerPipelineArgs params =
(PaymentManagerPipelineArgs)pParam;
    Order order = params.getOrder();
    StorePoints storePoints = (StorePoints)params.getPaymentGroup();
    double amount = params.getAmount();

    // create and populate store points info class
    StorePointsInfo spi = getStorePointsInfo();
    addDataToStorePointsInfo(order, storePoints, amount, params, spi);

    if (isLoggingDebug())
      logDebug("Putting StorePointsInfo object into pipeline: " +
spi.toString());

    params.setPaymentInfo(spi);

    return SUCCESS;
  }

  //-----------------------------------------------------------------------

  /**
   * Return the possible return values for this processor.  This processor
   * always returns a success code.
   **/

  public int[] getRetCodes() {
    int retCodes[] = {SUCCESS};
    return retCodes;
  }
```

As previously mentioned, the `StorePointsStatus` object represents a transaction on a `StorePoints` `PaymentGroup`. When the `PaymentManager` gets this object, it adds it to one of the `authorizationStatus`, `debitStatus`, or `creditStatus` `List` objects in the `PaymentGroup`. The specific list to which it is added depends on the operation.

Because none of the `StorePointsProcessor` methods throw exceptions, all operations must return an object that implements the `PaymentStatus` interface, as `PaymentStatusImpl` does. Therefore, when you implement `StorePointsStatus`, you should extend `PaymentStatusImpl`, which implements the `atg.payment.PaymentStatus` interface. Follow the steps in Extending the Purchase Process (page 341) to ensure that objects are persisted properly. The following table describes the properties in the `PaymentStatus` interface.

PaymentStatus Property	Type	Description
`transactionId`	`String`	A unique ID for the transaction that is generated by the payment processor.
`amount`	`double`	The amount of the transaction.
`transactionSuccess`	`boolean`	Indicates whether the transaction was successful. True indicates that the transaction succeeded. False indicates that it failed.
`errorMessage`	`String`	A detailed error message about the failure.
`transactionTimestamp`	`Date`	The time that the transaction was executed.

Below is an example of the `StorePointsStatus` class. All properties in this object must have values. The most important property is `transactionSuccess`. If `transactionSuccess` is false, then an exception is thrown with the message in the `errorMessage` property.

```
package store.some.package;

import atg.payment.*;

public class StorePointsStatus extends PaymentStatusImpl
{
  public StorePointsStatus() {
  }

  private String mConfirmationNumber = null;
  public String getConfirmationNumber() {
    return mConfirmationNumber;
  }
  public void setConfirmationNumber(String pConfirmationNumber) {
    mConfirmationNumber = pConfirmationNumber;
  }
}
```

Integrating the New Payment Processors into the PaymentManager

Integrating the new payment processors that you created in step 2 for the `StorePoints PaymentGroup`. This involves two steps:

1. Create the pipeline that creates `StorePointsInfo` objects and processes the `StorePoints PaymentGroup` for authorization, debits, and credits.

2. Configure the `PaymentManager` to invoke the pipeline when an operation is requested on a `StorePoints PaymentGroup`.

See the sections that follow for details.

Creating the Pipeline

To create the pipeline that creates the `StorePointsInfo` objects and performs actions on the `StorePoints PaymentGroup`:

1. Configure a pipeline processor to create the `StorePointsInfo` object. To do this, create a Nucleus component for the `ProcCreateStorePointsInfo` object. To create a Nucleus component located at `/store/payment/processor/CreateStorePointsInfo`, place the following properties file into Nucleus at that path:

   ```
   $class=store.some.package.ProcCreateStorePointsInfo
   $scope=global
   storePointsInfoClass=store.some.package.StorePointsInfo
   ```

2. Configure a pipeline processor to authorize, debit, and credit the `StorePoints` payment method. To do this, create a Nucleus component for the `ProcProcessStorePoints` object. To create a Nucleus component located at `/store/payment/processor/ProcessStorePoints`, place the following properties file into Nucleus at that path:

   ```
   $class=store.some.package.ProcProcessStorePoints
   $scope=global
   storePointsProcessor=/store/payment/StorePointsProcessorImpl
   ```

   Note that the `ProcessStorePoints.storePointsProcessor` property is set to the `StorePointsProcessor` object that actually performs the payment operations. `ProcessStorePoints` calls through to this object to perform the operations. In this example, it would be set to the `StorePointsProcessorImpl` Nucleus component.

3. Create a Nucleus component for the `StorePointsProcessorImpl` object. This is the object that actual performs the payment operations on the `StorePoints` payment method. To create a Nucleus component located at `/store/payment/StorePointsProcessor`, place the following properties file into Nucleus at that path:

   ```
   $class=store.some.package.StorePointsProcessorImpl
   $scope=global
   ```

4. Define the `storePointsProcessorChain` pipeline and add it to the pipelines used by the `/atg/commerce/payment/PaymentPipelineManager`. To do this, create a `paymentpipeline.xml` file in Nucleus at `/atg/commerce/payment/paymentpipeline.xml`. This XML file should define the single pipeline that operates on `StorePoints PaymentGroups`; it will be combined with the existing XML definition of payment pipelines using XML file combination.

   The following is a code example of the `storePointsProcessorChain` pipeline:

```
<pipelinemanager>
<!-- This chain is used to process a store point payment group-->
<!-- This single chain knows how to auth/credit/debit a        -->
<!-- payment group.  It also creates the StorePointInfo object-->

<pipelinechain name="storePointProcessorChain" transaction="TX_REQUIRED"
headlink="createStorePointInfo">
  <pipelinelink name="createStorePointInfo" transaction="TX_MANDATORY">
```

```
    <processor jndi="/store/payment/processor/CreateStorePointInfo"/>
    <transition returnvalue="1" link="processStorePoints"/>
  </pipelinelink>
  <pipelinelink name="processStorePoints" transaction="TX_MANDATORY">
    <processor jndi="/store/payment/processor/ProcessStorePoints"/>
  </pipelinelink>
</pipelinechain>
</pipelinemanager>
```

Configure the PaymentManager to Invoke the StorePointsProcessorChain

To configure the `PaymentManager` to invoke the `storePointsProcessorChain` pipeline when an operation is requested on a `StorePoints` `PaymentGroup`, you need to add a new entry to `PaymentManager.paymentGroupToChainNameMap`. The `paymentGroupToChainNameMap` property stores a mapping of `PaymentGroup` class names to the names of the pipelines to invoke when an operation for those `PaymentGroups` is requested.

To add a new entry to `PaymentManager.paymentGroupToChainNameMap`, layer on a configuration file that makes an additional entry to the `paymentGroupToChainNameMap` property. The new configuration file would be located in Nucleus at `/atg/commerce/payment/PaymentManager` and would look like the following:

```
PaymentGroupToChainNameMap+=Store.some.package.StorePoints=storePointsProc
essorChain
```

Extending Order Validation to Support New Payment Methods

Note: You can also follow this process to extend shipping group validation.

If you have implemented a custom payment method, such as the `StorePoints` payment method described in detail in the previous section, you may want to perform validation on the custom payment method during checkout. For example, for the `StorePoints` payment method you might want to make sure that the number of points specified is greater than zero. You might also want to include other validation logic – for example, you might decide that users cannot apply more than 500 store points to any order, and that store points may not be used to pay for more than 25% of the order price. You can test all of these conditions in a custom validation component for the `StorePoints` payment group.

When a user checks out an `Order`, the Oracle ATG Web Commerce purchase process performs validation checks on all of the payment groups in the `Order` by executing the `validateForCheckout` pipeline, which is defined in `commercepipeline.xml`. The `validateForCheckout` pipeline includes a `ValidatePaymentGroupsForCheckout` processor, which iterates over the payment groups in the `Order` and calls the `validatePaymentGroup` pipeline for each one to verify that the payment group is ready for checkout.

The `validatePaymentGroup` pipeline begins with a processor that examines the type of the current `PaymentGroup` and transfers control to a pipeline processor appropriate to that type. Credit cards are checked by one processor, gift certificates by another, store credits by still another. You can add your own pipeline processor to check the custom payment groups that you create.

Adding validation for your new payment method involves four steps:

Step 1: Implement a validation pipeline processor (page 315).

Step 2: Create an instance of the processor (page 316).

Step 3: Add the custom payment method to the ValidatePaymentGroupByType processor (page 317).

Step 4: Add the custom payment method to the validatePaymentGroup pipeline (page 317).

This subsections that follow describe each step in detail, using the `StorePoints` payment group that you created in the previous section as an example.

Step 1: Implement a validation pipeline processor

The first step in validating your custom payment method is to write a pipeline processor that examines a payment group and determines whether it meets the criteria for use in the `Order`. Recall that your processor must implement the interface `atg.service.pipeline.PipelineProcessor`, which consists of two methods:

- `public int[] getRetCodes();`
- `public int runProcess(Object pParam, PipelineResult pResult) throws Exception;`

A validation processor for `StorePoints` might look similar to the following:

```
package store.checkout;

import atg.service.pipeline.*;
import atg.nucleus.GenericService;
import atg.commerce.order.processor.ValidatePaymentGroupArgs;
import store.payment.StorePoints;

public class ValidateStorePoints
extends GenericService
implements PipelineProcessor
{
      private static int SUCCESS_CODE = 1;
      private static int[] RETURN_CODES = { SUCCESS_CODE };
/**
 * Return the list of possible return values from this
 * processor.  This processor always returns a single value
 * indicating success.  In case of errors, it adds messages
 * to the pipeline result object.
 **/

      public int[] getRetCodes()
{
        return RETURN_CODES;
      }

      /**
 * Perform validation for a StorePoints payment group.
 **/

public int runProcess(Object pParam, PipelineResult pResult)
{
  ValidatePaymentGroupPipelineArgs args;

  // Dynamo guarantees that the pipeline parameter object
  // passed to a payment group validation processor will be
  // of type ValidatePaymentGroupPipelineArgs.

  args = (ValidatePaymentGroupPipelineArgs)pParam;
  PaymentGroup pg = args.getPaymentGroup();

  // Now try casting the payment group to the type we expect
  // and validating the fields.  If the payment group is of
```

```
  // the wrong type, or if anything else goes wrong, add an
  // error to the pipeline result so the order manager will
  // abort the checkout process.

  try
  {
    StorePoints points = (StorePoints)pg;
    int nPoints = points.getNumberOfPoints();
    Order order = args.getOrder();
    double orderPrice = order.getPriceInfo().getTotal();

    // Log some debugging info about the number of points
    // and the total order price.

    if (isLoggingDebug())
      logDebug("Applying " + nPoints + " store points " +
               " to an order totaling " + orderPrice);

    // Are we using more than 500 points or trying to pay
    // for more than 25% of the order?  If so, add an error
    // to the pipeline result before returning.

    if (nPoints <= 0)
      pResult.addError(
        "NoPointsUsed",
        "The number of points should be greater than zero.");
    else if (nPoints > 500)
      pResult.addError(
        "TooManyPointsUsed",
        "A maximum of 500 points can be used per order.");
    else if (nPoints > orderPrice * .25)
      pResult.addError(
        "PointsValueExceeded",
        "Store points cannot pay for more than 25% of an order.");
  }
  catch (ClassCastException cce)
     {
    pResult.addError(
      "ClassNotRecognized",
      "Expected a StorePoints payment group, but got "
           + pg.getClass().getName() + " instead.");
  }
  return SUCCESS_CODE;
}
}
```

Note the use of the `ValidatePaymentGroupPipelineArgs` class in the `runProcess()` method. When the `OrderManager` validates payment groups, it guarantees that the pipeline arguments passed to your processor are an instance of this class. This provides you with a convenient way to retrieve items like the `Order`, the `OrderManager`, the `PaymentGroup`, and the server's `Locale` from the pipeline parameter map.

Step 2: Create an instance of the processor

After implementing your pipeline processor, you must configure an instance of the processor in Nucleus. For the `StorePoints` example, the validation processor might be located at `/store/checkout/ValidateStorePoints`, and it is configured with the following properties file:

```
# Store Points validation processor
```

```
$class=store.checkout.ValidateStorePoints
loggingDebug=true
```

In this simple example the processor doesn't require any additional property settings.

Step 3: Add the custom payment method to the ValidatePaymentGroupByType processor

Recall that payment groups are validated at checkout by invoking the `validatePaymentGroup` pipeline for each payment group in the order. The `validatePaymentGroup` pipeline begins with a pipeline processor called `ValidatePaymentGroupByType`, which examines the type of each payment group and returns an Integer that identifies the payment method. The pipeline then dispatches control to one of a number of different processors based on this return code.

To add support for your payment method, you must add an entry to `ValidatePaymentGroupByType`'s `returnValues` property, which maps your payment method's name (`storePoints` in this example) to a unique return value. You can use any return value as long as it isn't used by any other payment method. This example uses a value of 10. Configure the property by creating a file as follows in `localconfig/atg/commerce/order/processor/ValidatePaymentGroupByType.properties`:

```
# Add a return code for the storePoints payment method
returnValues+=\
      storePoints=10
```

Note that the payment method name, `storePoints`, is the name you added to the `paymentTypeClassMap` in the `OrderTools` component; it is **not** the name of your payment group's implementation class.

Step 4: Add the custom payment method to the validatePaymentGroup pipeline

The final step in adding validation for your `StorePoints` payment method is to reconfigure the `validatePaymentGroup` pipeline so it invokes the `ValidateStorePoints` processor when `ValidatePaymentGroupByType` returns a value of 10. This requires two changes to the pipeline configuration.

First, add a new transition tag to the `dispatchOnPGType` pipeline link in order to specify the pipeline link to use when `ValidatePaymentGroupByType` returns a value of 10. Second, define the new pipeline link and configure it to invoke your `ValidateStorePoints` component (in bold in the example that follows). You can do both steps using XML combination facilities. Modify the pipeline using the following code in `localconfig/atg/commerce/commercepipeline.xml`:

```
<!-- Modify the validatePaymentGroup chain to include  -->
<!-- validation for payment groups of type storePoints -->

<pipelinemanager>
  <pipelinechain name="validatePaymentGroup">
    <pipelinelink name="dispatchOnPGType">
      <transition returnvalue="10" link="validateStorePoints"/>
    </pipelinelink>
    <pipelinelink name="validateStorePoints">
      <processor jndi="/store/checkout/ValidateStorePoints"/>
    </pipelinelink>
  </pipelinechain>
</pipelinemanager>
```

Oracle ATG Web Commerce will now perform validation on your `StorePoints` payment method.

Scheduling Recurring Orders

Sites often require the functionality to create orders to be fulfilled repeatedly on a specific schedule, or to construct and save orders to be placed at a later date. You can use Oracle ATG Web Commerce to support these requirements through the use of scheduled orders.

In Commerce, a scheduled `Order` object (of type `scheduledOrder`) maintains the *schedule* information for the scheduled order, and a template `Order` object maintains the *order* information for the scheduled order. The template `Order` object is a typical `Order` in the `orderRepository`, but it has a state of TEMPLATE. When a scheduled order is placed, the template order is cloned, and the cloned order is checked out and sent to Fulfillment. Consequently, the template `Order` is never processed, but simply serves as a prototype.

Template orders must include enough information, such as all necessary shipping and payment information, to process the cloned `Order` without further user interaction. A previously processed order or even the user's shopping cart (once the shipping and payment information has been specified) can be used to create a template order.

This section describes the Commerce framework that supports scheduled orders and includes the following subsections:

For an example implementation of scheduled orders, see the *Scheduling Orders* section of the *My Account* chapter in the *ATG Business Commerce Reference Application Guide*.

Understanding the scheduledOrder Repository Item

The scheduled `Order` objects, stored in the Order Repository, maintain the schedule information for scheduled orders, as well as extra information as defined by the `scheduledOrder` item descriptor (for example, `name` and `state`).

The `scheduledOrder` item descriptor is defined in `<ATG10dir>//atg/commerce/order/orderRepository.xml`. By default, a `scheduledOrder` repository item contains the following properties:

Property	Description
`name`	The name that the user has assigned to the scheduled `Order`.
`profileId`	The profile ID of the user who created the scheduled `Order`.
`templateOrderId`	The ID of the template `Order` that is cloned whenever the scheduled order is placed.
`state`	The state of the scheduled `Order` (active, inactive, or error).
`clonedOrders`	The list of scheduled orders that have been placed. These `Orders` are clones of the template `Order` that have been checked out.

Property	Description
schedule	A string describing the Order's placement schedule.
nextScheduledRun	The next date and time that the scheduled Order should be placed.
createDate	The date and time that the scheduled Order was created.
startDate	The date and time that start the period within which the scheduled Order can be placed.
endDate	The date and time that end the period within which the scheduled Order can be placed. **Note:** If unset, the scheduled Order is repeatedly placed indefinitely.
id	The ID of the scheduled Order object. A read-only property.
type	The type of repository item. This is set to scheduledOrder.
Version	A value used to protect against data corruption that might be caused if two users attempt to edit this repository item at the same time. The system updates this read-only property automatically.
maxThreads	The current implementation of the scheduled order service can be configured to employ multiple threads when placing orders. By setting the parameter maxThreads to a number greater than 1, you can specify the number of threads that are started to process scheduled orders. If the maxThreads parameter is kept at the default of 1, all orders will be processed in the main thread. The optimal number of threads depends on a variety of factors, including server load, machine and database speed and the complexity of the scheduled order templates.

Submitting Scheduled Orders

The ScheduledOrderService is the back-end service that polls the Order Repository at a periodic interval and submits scheduled Orders according to their schedules.

When an application that includes Oracle ATG Web Commerce deploys, a PlaceScheduledOrders task is scheduled with the /atg/dynamo/service/Scheduler. This scheduled task is run at the interval specified in ScheduledOrderService.schedule. When the scheduled task is run, the Scheduler calls ScheduledOrderService.performScheduledTask(). The ScheduledOrderService then attempts to obtain a global write lock from the server lock manager. If the lock is obtained successfully, then it calls doScheduledTask().

At a general level, the ScheduledOrderService.doScheduledTask() method polls the Order Repository for all scheduled Orders that should be checked out. For each scheduled Order it finds due for checkout, it then clones the template Order associated with the scheduled Order, checks out the cloned Order, and sets the nextScheduledRun property of the scheduled Order to the next date and time it should be checked out. If an error occurs when processing an individual scheduled Order, then a CommerceException is thrown, the state of the scheduled Order is set to Error, and the state of the cloned Order is set to PENDING_MERCHANT_ACTION. However, the remaining scheduled Orders are processed.

The following table describes the various methods of ScheduledOrderService that support this process:

Method	Description
performScheduledTask	Called when the Scheduler finds that the placeScheduledOrders task is scheduled to run. When this method is called, the ScheduledOrderService attempts to obtain a global write lock from the server lock manager. If the lock is obtained successfully, the ScheduledOrderService calls its own doScheduledTask() method. The placeScheduledOrders task is run at the interval defined in ScheduledOrderService.schedule. While the schedule that you set depends on the business needs of your sites, as a general rule, it is recommended that you set it to "every 1 day." For information on how to specify a schedule, see the *Scheduler Services* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*. **Note:** Be aware that the defined interval specified in ScheduledOrderService.schedule begins when an application that includes Oracle ATG Web Commerce deploys. Therefore, if, for example, the Commerce server is redeployed every 11 hours, and if the schedule property of a given scheduled Order is set to "every 12 hours," then the scheduled Order is never placed.
doScheduledTask	Calls processDueScheduledOrders() to process any scheduled Orders that need to be checked out. Other objects can call this method to trigger on demand a poll of the Order Repository for due scheduled Orders.
processDueScheduledOrders	Processes all scheduled Orders that need to be checked out. This method calls pollForNewOrders() to retrieve the list of scheduled Orders that need to be checked out. It then iterates through the array and calls processDueScheduledOrder() on each scheduled Order. (See the methods described later in this table for details.)
pollForNewOrders	Uses the query defined in the ScheduledOrderService.pollQuery property to poll the Order Repository defined in the ScheduledOrderService.orderRepository property with the repository view defined in the itemDescriptorName property (by default, set to scheduledOrder). The method returns an array of scheduled Orders that need to be checked out.

Method	Description
`processDueScheduledOrder`	Processes the current scheduled `Order`. This method calls `placeScheduledOrder()` to check out the scheduled `Order`. If the scheduled `Order` is successfully checked out, then the method calls `getNextScheduledRun()` to set its `nextScheduledRun` property to the next date and time that it should be checked out. If the scheduled `Order` fails to be checked out, then it rolls back to its original state. It will be retrieved again for processing the next time that `pollForNewOrders()` is executed.
`getNextScheduledRun`	Sets the next time that the scheduled `Order` should be checked out.
`placeScheduledOrder`	This method checks out the current scheduled `Order` using the following process: First, the template `Order` is retrieved by calling `ScheduledOrderTools.getTemplateOrder()` and passing in the current scheduled `Order`. Second, `ScheduledOrderTools.UseOrderPriceListsFirst` identifies if price list information should be extracted from an order template. The default is false. Third, the template `Order` is cloned by calling `ScheduledOrderTools.cloneOrder()`. Fourth, if the `ScheduledOrderTools.repriceOnClone` property is set to True, then the cloned order is repriced by calling `ScheduledOrderTools.repriceScheduledOrder()`. Fifth, the cloned `Order` is saved to the Order Repository in its present state by calling `OrderManager.updateOrder()`, which executes the `updateOrder` pipeline. (For more information on the `updateOrder` pipeline, see Updating an Order with the OrderManager (page 278) in this chapter.) Finally, the cloned `Order` is checked out by calling `ScheduledOrderTools.processScheduledOrder()`, which in turn calls `OrderManager.processOrder()` and passes in the cloned `Order` and pipeline to execute. The pipeline to execute is set in `ScheduledOrderTools.processOrderChainId`; by default, this property is set to `processOrder`. When the order has been checked out successfully, it is passed on to Fulfillment. (For more information on the `processOrder` pipeline, see Checking Out an Order (page 294) in this chapter.)

Method	Description
repriceScheduledOrder	Reprices the cloned Order. The repriceScheduledOrder() method calls ScheduledOrderTools.repriceScheduledOrder(), which executes the repricing pipeline specified in ScheduledOrderTools.repriceOrderChainId. By default, this property is set to repriceOrder. (For more information on the repriceOrder pipeline, see Appendix F, *Pipeline Chains* (page 699).)

Oracle ATG Web Commerce provides an instance of class atg.commerce.order.scheduled.ScheduledOrderService, which extends atg.service.scheduler.SingletonSchedulableService. It is located in Nucleus at /atg/commerce/order/scheduled/ScheduledOrderService.

Class atg.service.scheduler.SingletonSchedulableService enables multiple Commerce servers to run the same scheduled service, while guaranteeing that only one instance of the service will perform the scheduled task at any given time. This provides a degree of protection from server failures, since the loss of any single Commerce server will not prevent the scheduled service from running on some other Commerce server.

To use a SingletonSchedulableService like ScheduledOrderService on multiple Commerce instances, you must run a server lock manager and point all client lock managers at it. For more information on SingletonSchedulableService, see the *Scheduler Services* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

Creating, Modifying, and Deleting Scheduled Orders

The ScheduledOrderHandler is responsible for creating, updating, deleting, activating, and deactivating scheduled orders. Oracle ATG Web Commerce provides an instance of atg.commerce.order.scheduled.ScheduledOrderHandler, which extends atg.repository.servlet.RepositoryFormHandler. It is located in Nucleus at /atg/commerce/order/scheduled/ScheduledOrderFormHandler.

A scheduled Order has some complex properties that are incapable of being mapped directly from the user interface to the value Dictionary defined in the RepositoryFormHandler (of which ScheduledOrderHandler is a subclass). Simple properties, such as name and state, can be handled by the superclass RepositoryFormHandler. However, other complex properties map to a large number of user input fields. For example, the startDate and endDate properties in the value Dictionary are both mapped to Year, Month, Date, and Hour input fields. Similarly, the schedule property maps to a large number of input fields. Each property of a scheduled Order that maps to more then one form element or needs special processing is represented by an instance of the abstract class atg..order.scheduled.ComplexScheduledOrderProperty or one of its subclasses. The ComplexScheduledOrderProperty class has two methods that facilitate the mapping between the property in the value Dictionary and the corresponding user input fields in a form, namely remapValueFromScheduledOrder() and remapValueFromUserInputField().

The ScheduledOrderFormHandler.complexScheduledOrderProperties property is a Map that specifies the complex properties in the scheduled Order. The keys to the Map are the names of the complex properties, and the values are the names of the classes that represent those properties, as shown in bold by the following ScheduledOrderFormHandler.properties file:

```
$class=atg.commerce.order.scheduled.ScheduledOrderHandler
```

```
$scope=session

#From RepositoryFormHandler
itemDescriptorName=scheduledOrder
repository=/atg/commerce/order/OrderRepository
requireIdOnCreate=false
clearValueOnSet=true

#From ScheduledOrderFormHandler
localeService=/atg/userprofiling/LocaleService
profile=/atg/userprofiling/Profile
orderManager=/atg/commerce/order/OrderManager
transactionManager=/atg/dynamo/transaction/TransactionManager
scheduledOrderTools=ScheduledOrderTools
complexScheduledOrderProperties=\
  calendarSchedule=atg.commerce.order.scheduled.CalendarSchedu
     leProperty,\
  startDate=atg.commerce.order.scheduled.DateProperty,\
  endDate=atg.commerce.order.scheduled.DateProperty,\
  templateOrderId=atg.commerce.order.scheduled.TemplateOrderProperty
```

Note that the `templateOrderId` property is represented by the `TemplateOrderProperty` class, which extends `ComplexScheduledOrderProperty`. When a user designates an existing `Order` as a template `Order`, the existing `Order` is copied, and the new template `Order` is assigned a new ID. The `templateOrderId` property contains a reference to the repository ID of the new template `Order`. When the user later views the scheduled `Order`, this property loads the associated template `Order` represented by the `templateOrderId`.

Once a scheduled order template has been modified, `ScheduledOrderLookup` can be used to provide information on scheduled orders for a scheduled order ID (`itemId`), template ID (`templateId`) or profile ID (`profileId`). Output parameters include `scheduledOrders` and `count`.

Scheduling information can be displayed in different formats. Using a scheduled order item or ID, `ScheduledOrderInfo` can provide a reference to scheduled objects, the scheduled order item and a readable string representation of the schedule. Output parameters to `ScheduledOrderInfo` are `scheduledOrderItem`, `readableSchedule` and `schedule`.

The following table describes the important handle methods of `ScheduledOrderFormHandler`:

Method	Description
`handleRepositoryId`	Called when the user wants to review or update a scheduled `Order`. This method is called before the user interface is rendered to the user. The method retrieves all of the property values for the scheduled `Order` with the ID set in `ScheduledOrderFormHandler.repositoryId`. It then populates the value Dictionary with the properties and finally remaps all the complex properties from the Order Repository to the user interface.
`handleVerify`	Called when the user wants to review the input data for the scheduled `Order` before it is created in the Order Repository. This method validates the submitted values, throwing a form exception if one is invalid.
`handleCreate`	Creates a scheduled `Order` in the Order Repository.
`handleUpdate`	Updates an existing scheduled `Order` in the Order Repository.

Method	Description
handleDelete	Deletes an existing scheduled Order from the Order Repository.
handleRemove	*Inactivates* an existing scheduled Order in the Order Repository by changing the state of the Order from ACTIVE to INACTIVE.
handleRestore	*Activates* an existing scheduled Order in the Order Repository by changing the state of the Order from INACTIVE to ACTIVE.

Note that the ScheduledOrderFormHandler uses ScheduledOrderTools to fire events for all of the actions that are associated with these handle methods.

Using the scheduledOrderFormHandler is very similar to using the RepositoryFormHandler. (For more information on using the RepositoryFormHandler, see the *Using Repository Form Handlers* chapter of the *ATG Page Developer's Guide*.) Simple properties like name, state, and nextScheduledRun can all be set in the following manner:

```
<dsp:form action="setName.jsp">
   new name : <dsp:input bean="ScheduledOrderHandler.value.name"
type="text"/><br>
   <dsp:input bean="ScheduledOrderHandler.update" value="update name"
type="submit"/>
</dsp:form>
```

Complex properties of the scheduled Order are set according to the configuration of ScheduledOrderFormHandler.complexScheduledOrderProperties property. As shown in the ScheduleOrderFormHandler.properties file above, the startDate and endDate complex properties of a scheduled Order are represented by the DateProperty class. The following JSP example illustrates how to change these properties, using the month in the startDate as an example:

```
<dsp:form action="setStartDateMonth.jsp">
   New start month : <dsp:input
bean="ScheduledOrderFormHandler.value.startDate.month" type="text"/> <br>
   <dsp:input bean="ScheduledOrderFormHandler.update" value="update month"
type="submit"/>
</dsp:form>
```

You can modify the schedule property of a scheduled Order in a similar manner. Two classes in package atg.commerce.order.scheduled can represent the schedule property of a scheduled Order:

- atg.commerce.order.scheduled.CalendarScheduleProperty

  If used, the schedule property is represented by a CalendarSchedule, and the schedule property is mapped to the user input form fields used by the CalendarSchedule.

  A CalendarSchedule specifies a task that occurs according to units of the calendar and clock (for example, at 2:30 AM on the 1st and 15th of every month).

- atg.commerce.order.scheduled.PeriodicScheduleProperty

  If used, the schedule property is represented by a PeriodicSchedule, and the schedule property is mapped to the user input form fields used by the PeriodicSchedule.

A `PeriodicSchedule` specifies a task that occurs at regular intervals (for example, every 10 seconds in 20 minutes without catch up).

The following JSP example illustrates how to change the `schedule` property. In this example, the frequency of a schedule whose `scheduledMode` is Monthly is updated:

```
<dsp:form action="setSchedule.jsp">
…

<dsp:select
bean="ScheduledOrderFormHandler.complexScheduledOrderMap.calendarSchedule.
userInputFields.scheduleMode" size="1" name="select">
  <dsp:option value="onceMonthly"/>once a month.
  <dsp:option value="biMonthly"/>every two months.
  <dsp:option value="quarterly"/>every quarter.
</dsp:select>

…

<dsp:input bean="ScheduledOrderFormHandler.update" value="Update"
type="submit"/>
</dsp:form>
```

For more information on `CalendarScheduleProperty` and `PeriodicScheduleProperty`, see the *ATG Platform API Reference*. For more information on `CalendarSchedule` and `PeriodicSchedule`, see the *Scheduler Services* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

Using Scheduled Orders with Registered Sites

If you have registered sites for use with Oracle ATG Web Commerce's multisite feature, then scheduled orders can be "site-aware" just like other Commerce orders. Scheduled orders are different from regular orders placed by users, however, because there is no actual user interacting with a site when the order is placed. Instead, the `ScheduledOrderFormHandler` saves the `siteId` as part of the order template.

The `siteId` is copied from the initial order's `siteId` property, even if the value is null, and overrides any `siteId` set in the page. The ID is then retrieved by the `ScheduledOrderService` during order processing.

The `ScheduledOrderTools` component includes one property to control how scheduled orders are processed for registered sites:

- `useSitePriceLists` – If `true`, retrieves the price list from the scheduled order. If `false`, the price list is determined based on the profile. The default value is `true`.

Setting Restrictions on Orders

In some situations you may want to prevent an order from being placed. For example, a given item might be prohibited from sale in certain locations, or you may want to ensure that customers order a minimum quantity

of a given item, or you may want to designate some items as requiring approval for purchase. You can use Oracle ATG Web Commerce to specify criteria that orders must meet if they are to be placed, thereby setting restrictions on certain kinds of orders.

This section describes the order restriction system in Commerce and includes the following subsections:

Understanding the Order Restriction Classes (page 326)

Implementing Order Restrictions (page 328)

Understanding the Order Restriction Classes

You set restrictions on `Orders` by specifying the criteria or "rules" that an `Order` must meet if it is to be placed. The functionality for this `Order` restriction system relies on the following Oracle ATG Web Commerce classes in package `atg.commerce.expression`:

- Rule (page 326), which represents the rule.
- ExpressionParser (page 326), which parses the expression (rule).
- RuleEvaluator (page 326), which evaluates the expression to either to True or False.
- ProcPropertyRestriction (page 327), which evaluates the rule using the `ExpressionParser` and `RuleEvaluator`.

Rule

The `atg.commerce.expression.Rule` class represents a rule. A `Rule` object contains `RuleExpressions` (operands) and an operator, which together are evaluated to either True or False.

ExpressionParser

The `atg.commerce.expression.ExpressionParser` class is used to parse expressions. `ExpressionParser` supports the following operators: =, >, <, >=, <=, !=, `contains`, and `containsKey`. It does **not** support & (and) or | (or). The `ExpressionParser.parseExpression()` method takes a string containing an expression to parse, such as the following:

```
Order.priceInfo.amount > Profile.maxAmountAllowed
Profile.approvalRequired = true
Order.id = null
Order.specialInstructions.shippingInfo.size != 1000
```

After parsing the expression, the `parseExpression()` method returns a `Rule` object, which is then passed to the RuleEvaluator (page 326) for evaluation.

Oracle ATG Web Commerce provides a globally-scoped instance of `ExpressionParser`, which is located in Nucleus at `/atg/commerce/util/`.

RuleEvaluator

The `atg.commerce.expression.RuleEvaluator` class is used to evaluate a given rule. After the ExpressionParser (page 326) parses an expression, it returns a `Rule` object and passes it to the `RuleEvaluator`. The `RuleEvaluator.evaluateRule()` method then evaluates the rule to True or False.

The RuleEvaluator class supports all primitive data types in Java. These are long, double, int, short, float, boolean, char, and String. If two incompatible data types are evaluated, such as a double and boolean, then an EvaluationExpression is thrown.

Oracle ATG Web Commerce provides a globally-scoped instance of RuleEvaluator, which is located in Nucleus at /atg/commerce/util/.

ProcPropertyRestriction

The atg.commerce.expression.ProcPropertyRestriction class resolves all references in the rule set in ProcPropertyRestriction.ruleExpression using the ExpressionParser (page 326) and then evaluates the rule using the RuleEvaluator (page 326). The processor then returns a value based on whether the rule evaluates to True or False. The specific value returned is determined by its returnValueForTrueEvaluation and returnValueForFalseEvaluation properties.

Additionally, if the expression evaluates to True and the addErrorToResultOnTrueEval property is set to True, then the value in the errorMessage property is added to the PipelineResult object, keyed by the string in the pipelineResultErrorMessageKey property. (See the table below for more information on these properties.)

The ProcPropertyRestriction processor has the following properties:

Property	Description
ruleExpression	The expression that is passed to the ExpressionParser, such as Order.priceInfo.amount > 1000.0. This is the rule against which the processor evaluates the Order.
expressionParser	The ExpressionParser Nucleus component. If null, the ProcPropertyRestriction processor creates a new instance of atg.commerce.expression.ExpressionParser for its use.
ruleEvaluator	The RuleEvaluator Nucleus component. If null, the ProcPropertyRestriction processor creates a new instance of atg.commerce.expression.RuleEvaluator for its use.
returnValueForFalseEvaluation	The integer to return when the expression evaluates to False.
returnValueForTrueEvaluation	The integer to return when the expression evaluates to True.
addErrorToResultOnTrueEval	A boolean property that controls whether the errorMessage is added to the PipelineResult object when the expression evaluates to True.
errorMessage	The error message to add to the PipelineResult object when the expression evaluates to True.
pipelineResultErrorMessageKey	The key to use when adding the errorMessage to the PipelineResult object.

Implementing Order Restrictions

You can set restrictions on `Orders` by adding a `ProcPropertyRestriction` processor to any pipeline, for example, the `validateForCheckout` pipeline.

To do so, create an instance of `atg.commerce.expression.ProcPropertyRestriction`, defining the rule by which `Orders` are to be evaluated in `ProcPropertyRestriction.ruleExpression` and setting the remaining properties as necessary. (See the table in the previous section for more information on the properties of `ProcPropertyRestriction`.) Then add the processor at any point in any pipeline.

For example, you might check `Orders` against certain restrictions before checkout by adding a link to the `validateForCheckout` pipeline in `<ATG10dir>/DCS/src/config/atg/commerce/commercepipeline.xml`. To insert a new link, add a new element to the XML file that references `atg.commerce.expression.ProcPropertyRestrictions`, as follows:

```
<pipelinechain name="validateForCheckout" transaction="TX_REQUIRED"
   headlink="validateOrderForCheckout"
   classname="atg.service.pipeline.PipelineChain"
   resultclassname="atg.service.pipeline.PipelineResultImpl">
   <pipelinelink name="propertyRestrictions" transaction="TX_MANDATORY">
      <processor
          jndi="/atg/commerce/expression/ProcPropertyRestrictions"/>

. . .

</pipelinechain>
```

For more information on pipelines and how to extend them, see the *Processor Chains and the Pipeline Manager* (page 363) chapter.

Tracking the Shopping Process

The Oracle ATG Web Commerce platform includes a business tracking feature that lets you define a business process as a series of stages, track activity within the business process, and report on the activity for a specified time frame. In Oracle ATG Web Commerce, we've defined the lifecycle of an order (through the point the customer checks out) as a business process, allowing you to track stages in the shopping process from browsing, to adding items to the shopping cart, to completing shipping and billing information, to final check out. You can use the information gathered by tracking the stages of the shopping process to understand better how your customers react to their experience with your sites. For example, if reports show that many customers are abandoning their purchases at the stage where you display the shipping price, that may be an indication of poor or confusing application page design, or undesirable pricing schemes. You can also use shopping process tracking to modify a customer's experience. For example, you could offer special promotions to customers who stall at a particular point in the shopping process.

For more information about business process tracking, including information about how to create new business processes, see the *Defining and Tracking Business Processes* chapter in the *ATG Personalization Programming Guide*

Shopping Process Stages

Oracle ATG Web Commerce has one business process configured out-of-the-box, the shopping process. The shopping process is made up of a series of stages that a customer follows, from browsing for products to, if all goes well, purchasing and checking out. The stages of the shopping process in Commerce are defined as:

```
Browsed
AddedToCart
ShippingInfoComplete
ShippingPriceDisplayed
RequestedBillingInfo
BillingInfoCompleted
CartSummaryViewed
```

These stages are defined in the `stageNames` property of the `/atg/commerce/bp/ShoppingProcessConfiguration` component.

The `ShoppingProcessConfiguration` component also specifies the duplication mode for the shopping process. The duplication mode determines what happens if an order reaches a business stage for the second or subsequent time. By default, the duplication mode of the `ShoppingProcessConfiguration` component is NO_DUPLICATES, which means an order is marked as having reached a new stage in the shopping process only the first time it reaches that stage. No change is made if the order reaches the same stage again.

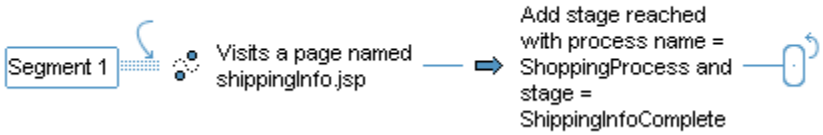
Every commerce site is different. The business process stages that are defined by default in Commerce may not fit the needs of your sites. You can define whatever business process stages you want by setting the `stageNames` property of the `/atg/commerce/bp/ShoppingProcessConfiguration` component. Whether you use the default business process stages or define your own, you need to track them by adding servlet beans to your checkout pages or defining scenarios to mark when a stage is reached, as described in the rest of this section.

Working with Shopping Process Stages

Oracle ATG Web Commerce includes page-based and scenario-based tools that let you add, remove, and check for business process stages. For the shopping process, these include the following servlet beans and scenario elements:

Task	Servlet Bean	Scenario Element
Add a shopping process stage	`AddShoppingProcessStageDroplet`	Add Stage Reached
Remove a shopping process stage	`RemoveShoppingProcessStageDroplet`	Remove Stage(s) Reached
Check if a shopping process stage has been reached	`HasShoppingProcessStageDroplet`	Has Reached Stage
Check the most recent shopping process stage that has been reached	`MostRecentShoppingProcessStage Droplet`	Most Recent Stage Reached

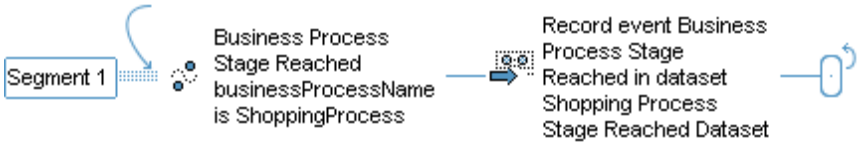
These servlet beans are instances of classes in the `atg.markers.bp.droplets` package, each with the default process name set to `ShoppingProcess`. See *Appendix: ATG Commerce Servlet Beans* in the *ATG Commerce Guide to Setting Up a Store* for reference information about these servlet beans. You can use the servlet beans to the checkout pages of a Commerce application to add, remove, and check shopping process stages. You can as an alternative use the corresponding scenario elements in a scenario. For example, you could create a scenario like this:



The business stage scenario elements are described in the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

Shopping Process Recorder

Data about stages reached in the shopping process is recorded by the `shoppingprocess` scenario recorder:



This recorder makes a record in the Shopping Process Stage Reached Dataset whenever an order reaches a new stage in the shopping process. Note that by default the shopping process is configured with the NO_DUPLICATES setting, which means that we only track the first time an order reaches a stage in the shopping process.

Turning Off Recording of Shopping Process Tracking

If you don't want to generate reports on the shopping process, you can disable shopping process events by setting the `generateEvents` property of the `/atg/commerce/bp/ShoppingProcessConfiguration` component to `false`.

Troubleshooting Order Problems

If you modify the functionality of the `OrderManager` or its related components, you should make sure to follow these guidelines:

- When making changes to an `Order`, you must call the `updateOrder()` method. (For more information on the `updateOrder()` method, see Updating an Order with the OrderManager (page 278) in the *Saving Orders* section of this chapter.)

- `updateOrder()` must always be called within a transaction. (For more information on transactions, see the *Transaction Management* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.)
- If errors occur when a user logs in, set the `persistOrders` property of the `ShoppingCart` component to true.

The `handleMoveToPurchaseInfoByRelId` method and all other handle methods of `atg.commerce.order.purchase.CartModifierFormHandler` provide good examples of how and when to call `getOrderManager.updateOrder()`.

Handling Returned Items

In Oracle ATG Web Commerce, you can use the `CommerceItemManager` (class `atg.commerce.order.CommerceItemManager`) to manage the return of items in an `Order`. Call `CommerceItemManager.returnCommerceItem()` to mark a given `CommerceItem` as returned. This method does the following:

- Reduces the `quantity` property in the `CommerceItem` by the quantity returned.
- Reduces the `quantity` property in the `ShippingGroupCommerceItemRelationship` by the quantity returned.
- Increases the `returnedQuantity` property in the `ShippingGroupCommerceItemRelationship` by the quantity returned.

After the item is returned, the `quantity` of the `CommerceItem` reflects the final quantity that was purchased. The `quantity` property of the `ShippingGroupCommerceItemRelationship` reflects the quantity that was shipped, and its `returnedQuantity` reflects that quantity of the `CommerceItem` that was returned.

The store credit system in Oracle ATG Web Commerce can also manage the return of items. The Claimable Repository contains a `storeCreditClaimable` item that includes the following properties:

- `amount` (double) – the original amount of the credit
- `amountAuthorized` (double) – the amount of credit authorized for use
- `amountRemaining` (double) – the current amount of the store credit
- `ownerId` (String) – the ID of the user or organization for which the credit was issued
- `lastUsed` (date) – the date the credit was last accessed

To modify an existing store credit account (for example, to increase the amount of remaining credit), call the `ClaimableManager.updateClaimableStoreCredit()` method. To create a new store credit account for a user or organization, call the `ClaimableManager.createClaimableStoreCredit()` method (`atg.commerce.claimable.ClaimableManager`) to create the new store credit and then call the `ClaimableManager.initializeClaimableStoreCredit()` method to set its initial values.

Note that only one store credit account can exist for a given user or organization.

For more information on the Claimable Repository, including gift certificates, see the *Configuring Commerce Services* (page 71) chapter.

Managing Transactions in Oracle ATG Web Commerce

Most of the Oracle ATG Web Commerce form handlers extend `atg.commerce.order.purchase.PurchaseProcessFormHandler`. This form handler, which is a subclass of `atg.droplet.GenericFormHander`, is an abstract class which implements a transaction management pattern that should be followed by any custom form handlers. This transaction management pattern is implemented through the form handler's `beforeSet`, `afterSet`, and handler methods provided as part of the Commerce form handlers.

beforeSet Method

This method is called once before any form handler property is set or handler method is called. It implements the following transactional steps:

1. If the form handler's `useLocksAroundTransactions` property is `true` (the default), obtain a transaction lock before the transaction is created.

   This prevents a user from modifying an order in multiple concurrent threads. The lock name used defaults to the current profile ID. For more information, see `atg.commerce.util.TransactionLockFactory`. (Note that use of locking has a small performance impact.)

2. Check for an existing transaction and, if no transaction exists, create one.

Handler Methods

The handler methods implement the following transactions steps:

1. Synchronize on the `Order` object.

2. Execute logic for modifying the `Order` object.

   For example, the `CartModifierFormHandler` subclass has a `handleAddItemToOrder` method that executes the logic of adding an item to an order.

3. Call the `OrderManager` object's `updateOrder` method to save the order data to the repository.

4. End the synchronization.

afterSet Method

This method is called once after all form handler processing is completed. It implements the following transactional steps:

1. Commit or roll back any transaction that was created in the `beforeSet` method.

   If the transaction was already in place before the `beforeSet` method was called, the `afterSet` method does **not** end the transaction automatically; this is the application's responsibility.

2. If a transaction lock was acquired in the `beforeSet` method, release the lock.

If you're extending an Oracle ATG Web Commerce form handler and your code makes its own decisions about errors, you can mark a transaction for rollback by calling the `setTransactionToRollbackOnly` method.

For more information on `PurchaseProcessFormHandler` and its subclasses, you can examine the source files at `<ATG10dir>\DCS\src\Java\atg\commerce\order\purchase` and refer to the *ATG Platform API Reference*.

Extending the Oracle ATG Web Commerce Form Handlers

If you write a form handler that modifies the `Order` object, you should implement the transaction-handling pattern described above. The easiest way to do this is to extend either `PurchaseProcessFormHandler` or a subclass of `PurchaseProcessFormHandler`. Your form handler will then inherit the `beforeSet` and `afterSet` methods, so you won't need to replicate their portion of the transaction logic. However, any new handler methods you write will need to implement the transaction logic described in the Handler Methods (page 332) section.

Note that the handleXXX methods of the Oracle ATG Web Commerce form handlers invoke preXXX and postXXX methods before and after any computation is performed. For example, when a customer adds an item to their shopping cart using `CartModifierFormHandler`, the submit button on the form submits to the `handleAddItemToOrder` method. Before any computation is done, the `handleAddItemToOrder` method invokes the `preAddItemToOrder` method. Additionally, after all computation is complete but before returning any information, the `handleAddItemToOrder` method invokes the `postAddItemToOrder` method.

By default these preXXX and postXXX methods have no functionality. They are provided so you can easily extend the form handlers to support additional functionality. However, be aware that these methods do not take any input parameters.

16 Customizing the Purchase Process Externals

This chapter contains information on Oracle ATG Web Commerce features that operate outside the actual process of creating and placing orders. It also contains information on how to extend the existing purchase framework.

Application Messaging

Oracle ATG Web Commerce provides a messaging system that lets you configure multiple message types. You can then send the messages to slots used by an application (see the *ATG Personalization Programming Guide*), or send them as JMS messages to be used by scenarios. Out of the box, this system is used by Gift with Purchase promotions (see Gift with Purchase Promotions (page 181) in this guide) and by the stacking rules feature (see the *ATG Merchandising Guide for Business Users*). You can extend the provided framework for your custom components.

Application Messaging Using Slots

To use slot messaging, set the `PricingTools.sendUserMessages` property to `true`. The default is `false`.

The `PricingTools.messageTools` component includes a `sendUserMessage()` method that resolves the `MessageTools` component and sends the message to it.

```
/*
* @param pMessage UserMessage to send
* @param pMessageTools MessageTools component to use (optional)
* @param pExtraParameters Map of extra parameters
*/
public void sendUserMessage(UserMessage pMessage, MessageTools
pMessageTools, Map pExtraParameters)
```

The `MessageTools` class allows you to configure a single slot using the `userMessagingSlot` property. `MultiSlotMessageTools` allows you to configure multiple slots to store messages, as shown in this example:

```
/**
   * Adds message to one or more user messaging slots for storage until the UI is
ready to view the messages.
   * <br>If the message identifier maps to any slots via the
<code>slotMessages</code> mapping then
   * <br>we send the message to those slots otherwise
   * <br>If the message group maps to any slots via the
<code>slotMessageGroups</code> mapping then
   * <br>we send the message to those slots otherwise
   *  the superclass implementation of this method is invoked.
   *
   * @param pMessage The user message to store in the slots
   */
  @Override
  public void addMessage(UserMessage pMessage)
```

Configure the `slotMessages` Map property of the `MyMultiSlotMessageTools` component with the slot paths and strings that identify user messages. When a message is added, it is sent to the slots configured in the map.

Alternatively, configure the `slotMessagesGroups` property, which maps slot components to a user message group string. If a user group mapping is added, any messages with that message group are sent to the slots configured in the map.

If you want to remove messages from slots, set the `slotsToClean` property of `MyMultiSlotMessageTools` with the names of the slots from which to remove messages. Message removal takes place at the start of the pricing operation.

You can configure a default slot in the `userMessagingSlot` property, or explicitly send a message to the default slot by adding a mapping with this key:

```
[default]
```

The following example Map includes these slots:

- GWPSlot for Gift with Purchase qualification messages
- GWPFailureSlot for Gift with Purchase failure notifications
- A default message slot

```
# Slot messages property
slotMessages=\
  [default]=GWPQualified\
  /app/GWPSlot=GWPQualified\
  /app/GWPFailureSlot=GWPPartialFailure,GWPFullFailure
```

This example does not use groups. Note that in this example the failure messages have an explicitly configured slot and do not go to the default slot.

The `UserMessage` class has the following properties:

Property	Data Type	Description
`identifier`	String	The unique message identifier
`messageGroup`	String	A group to which this message belongs, such as `GiftWithPurchaseMessages`.
`summary`	String	A string description for the message.
`priority`	int	An integer priority, Validation=10, Form Error (not validation)=5, Error (default)=0, Confirmation=0, Warning=-5, Information=-10
`type`	String	Message type: error (default), confirmation, warning, information
`params`	Object[]	Optional message specific array of object parameters that relate to the message.

Application Messaging Using JMS Messages

To turn on application JMS messaging, set the `sendPricingMessages` property of the `PricingTools` component to `True`.

Oracle ATG Web Commerce includes a `PricingMessage` JMS message class that can wrap a `UserMessage`. The `PricingMessage` class extends `CommerceMessageImpl` class to create a simple JMS class with a `userMessage` property containing the application message. The `type` property is derived from the user message identifier by prefixing `atg.commerce.pricing.` For example,

```
atg.commerce.pricing.GWPQualified
```

Default application messages are providing for the Gift with Purchase and stacking rules features.

All of the following properties must be set to true in order to use JMS messaging:

- `GWPManager.jmsEnabled`—The default is `false`.
- `PricingTools.sendUserMessages`—The default is `false`.
- `PricingTools.sendPricingMessages`—The default is `false`.
- `PricingTools.sendGWPMessages`—The default is `true`.

Application Messaging for Gift with Purchase Promotions

By default, Commerce includes messages used by the Gift with Purchase promotion feature. To enable messages related to this feature, set the following property to true:

```
PricingTools.sendGWPMessages
```

By default, Commerce includes a `GiftWithPurchaseMessages` message group that is used to send messages for that feature.

The default slot for Gift with Purchase messages is `/atg/commerce/pricing/NoCleanBeforePricingSlot`. Messages are removed from this slot as they are retrieved.

Application Messaging for Stacking Rules

By default, Commerce includes messages used by the stacking rules feature. To enable messages related to this feature, set the following property to true:

```
PricingTools.sendStackingRuleMessages
```

By default, Commerce includes a `StackingRuleMessages` message group that is used to send messages for that feature.

The default slot for stacking rule messages is `/atg/commerce/pricing/CleanBeforePricingSlot`. Messages are not removed from this slot as they are retrieved. However, by default, the slot is configured in the `MultiSlotMessageTools.slotsToClean` property, so messages are cleared automatically at the start of each pricing operation.

Purchase Process Event Messages

The event messages generated by the purchase process are sent at various points in the purchase process. These include messages that are sent when a product or category is browsed, when an item is added or removed from the order and when an order has been submitted for fulfillment (checkout complete).

Note: This section is for reference. These messages are generated automatically during the purchase process.

Event	Description
`ItemAddedToOrder`	Sent when an item is added to an order. Includes order ID, commerce item ID, site ID, and JMS message type
`ItemRemovedFromOrder`	Sent when an item is removed from an order. Includes the order ID, commerce item ID and the JMS message type
`SubmitOrder`	Sent when an order has been submitted for fulfillment (checkout complete). Includes the serialized order, the JMS message type

Event	Description
`ViewItem`	Sent when a product or a category is browsed, includes the repository name, the item type, the repository id and the serialized repository item object.

Oracle ATG Web Commerce factions include giving promotions to customers when they add a particular item to their order or browse a particular product or category. Additionally, confirmation e-mail is sent through a scenario that is listening for the `Submitorder` message.

Integrating with Purchase Process Services

This section contains the following information:

Purchase Process Integration Points

There are several points in the purchase process where specialized components can be integrated into the system. To integrate a component, configure a property of a Nucleus component to reference an object that implements an interface (as described below).

The following list describes the integration points in the purchase process:

- `PaymentManager.creditCardProcessor`

  Use this property of the `PaymentManager` to integrate with a credit card processing system. The credit card processing system must implement the `atg.payment.creditcard.CreditCardProcessor` interface. By default, the `PaymentManager` is configured to use a dummy processing system, `atg.commerce.payment.DummyCreditCardProcessor`.

  The `PaymentManager` is located in Nucleus at `/atg/commerce/payment/`.

- `PaymentManager.giftCertificateProcessor`

  Use this property of the `PaymentManager` to integrate with a gift certificate processing system. The gift certificate processing system must implement the `atg.payment.giftcertificate.GiftCertificateProcessor` interface. By default, the `PaymentManager` is configured to use the Oracle ATG Web Commerce gift certificate processing system, `atg.commerce.payment.GiftCertificateProcessorImpl`.

  The `PaymentManager` is located in Nucleus at `/atg/commerce/payment/`.

- `VerifyOrderAddresses.addressVerificationProcessor`

  Use this property of the `ProcVerifyOrderAddresses` object to integrate with an address verification system. The address verification system must implement the `atg.payment.avs.AddressVerificationProcessor` interface. By default, the `VerifyOrderAddresses`

component, which is located in Nucleus at `/atg/commerce/order/processor/`, is configured to use a dummy processing system, `atg.commerce.payment.DummyAddressVerificationProcessor`.

Adding Credit Card Types to Oracle ATG Web Commerce

This section describes how to extend Oracle ATG Web Commerce and a payment system integration to use the additional credit card types that the payment system might accept.

By default, Commerce considers only common credit cards valid. These cards include Visa, MasterCard, etc. Many payment systems handle many other credit and debit cards, such as Switch/Solo. Many of these other cards have more validation parameters than the standard cards. If your commerce site needs to accept these cards, you can extend Commerce to handle these card types.

The following sections describe the three parts to extending Commerce to include new credit card types:

1. Extending the CreditCard Class (page 340)
2. Extending the CreditCardInfo Class (page 341)
3. Extending the Payment System Integration (page 341)

Extending the CreditCard Class

The following steps describe how to extend the `CreditCard` class and modify Oracle ATG Web Commerce to use the new class. For general information on extending a class and modifying the Commerce purchase process, see the Extending the Purchase Process (page 341) section of this chapter.

1. Create a subclass of `atg.commerce.order.CreditCard` and include any new properties you need for the credit card type. Add get/set methods for each of these properties. The get and set methods need to use `super.getPropertyValue()` and `super.setPropertyValue()`, so that the underlying repository item is updated correctly.

   For example, the following code sample creates a property for the issue number of the credit card:

   ```
   //-------------------------------------------------------
   // property:IssueNumber
   //-------------------------------------------------------
   public void setIssueNumber(String pIssueNumber) {
   setPropertyValue("issueNumber",
   (pIssueNumber == null ? pIssueNumber :
   StringUtils.removeWhiteSpace(pIssueNumber)));
   }
   /**
   * The issue number of this credit card
   * @beaninfo description: The issue number of this credit card
   **/
   public String getIssueNumber() {
   return (String) getPropertyValue("issueNumber");
   }
   ```

2. Add columns to the `dcspp_credit_card` table to store your new properties for the `CreditCard` subclass.
3. Extend `orderrepository.xml` to add the new properties in your `CreditCard` subclass to the existing `creditCard` item descriptor.
4. Modify `/atg/commerce/order/OrderTools` to make Commerce use your new `CreditCard` subclass instead of the default class. For example:

```
beanNameToItemDescriptorMap+=\
my.class.dir.myCreditCard=creditCard
paymentTypeClassMap+=\
creditCard=my.class.dir.myCreditCard
```

5. Modify `/atg/commerce/payment/PaymentManger.paymentGroupToChainNameMap` to contain a pointer to your new class:

   ```
   paymentGroupToChainNameMap+=\
   my.class.dir.myCreditCard=creditCardProcessorChain
   ```

6. Edit the following properties of `/atg/commerce/payment/ExtendableCreditCardTools` to include appropriate values for your new `CreditCard`:

   - `cardCodesMap`
   - `cardLengthsMap`
   - `cardPrefixesMap`
   - `cardTypesMap`

Extending the CreditCardInfo Class

The following steps describe how to extend the `CreditCardInfo` class to accommodate the new credit card type.

1. Create a subclass of `atg.payment.creditcard.GenericCreditCardInfo`, with any necessary new properties. Refer to the payment system's documentation for information on what properties it needs to process the new credit card type.
2. Modify `/atg/commerce/payment/processor/CreateCreditCardInfo.creditCardInfoClass` to point to the new subclass.
3. Create a new class that extends `atg.commerce.payment.processor.ProcCreateCreditCardInfo`. In this class, extend the `addDataToCreditCardInfo` method to call the superclass, followed by code that adds your new properties (added in step 1) to the `CreditCardInfo` object.
4. Modify the class of `CreditCreditCardInfo.properties` to point to your new subclass.

Extending the Payment System Integration

The final part of the process of adding a new credit card type is to extend the credit card processor for your payment system to use your new card type's properties in its validation mechanisms. The payment system integration will have an implementation of `atg.payment.creditcard.CreditCardProcessor`.

In addition, the $class line in the properties file for the credit card processor must be changed to use your new subclass.

Extending the Purchase Process

Extending the purchase process is necessary when you want to store purchasing information that is not included in the out-of-the-box functionality of Oracle ATG Web Commerce. For example, if you want to allow

customers to specify a box size for their purchases, you could extend the purchase process to store that information.

You extend the purchase process by first subclassing an existing object in the commerce object hierarchy to add new properties and then integrating that new class into Commerce. See the following sections for details:

Adding a Subclass with Simple Data Type Properties (page 342)

Adding a Subclass with Complex Data Type Properties (page 347)

Manipulating Extended Objects (page 361)

Note: For information on extending the Commerce payment process to support a new payment method or additional operations for an existing payment method, see the Processing Payment of Orders (page 300) section in the *Configuring Purchase Process Services* chapter.

Adding a Subclass with Simple Data Type Properties

You can extend the commerce object hierarchy by subclassing an existing object and adding new properties. When you add simple data type properties such a strings, you don't need to write any code to save the properties to or load the properties from the Order Repository. Using introspection, the processors in the `updateOrder` and `loadOrder` pipelines handle this automatically.

As an example, the following code creates a new class called `MyCommerceItemImpl`. It extends `CommerceItemImpl` and adds a new `String` property called `shortDescription`.

```
import atg.commerce.order.CommerceItemImpl;

public class MyCommerceItemImpl extends CommerceItemImpl {
  public MyCommerceItemImpl() {
  }

  public String getShortDescription() {
    return (String) getPropertyValue("shortDescription");
  }

  public void setShortDescription(String pShortDescription) {
    setPropertyValue("shortDescription", pShortDescription);
  }
}
```

In the code example above, note the calls to `getPropertyValue()` and `setPropertyValue()`. These methods retrieve and set the values of properties directly on the repository item objects; they are part of the `atg.commerce.order.ChangedProperties` interface. In a commerce object that supports the `ChangedProperties` interface, every `get()` method needs to call the `getPropertyValue()` method, and similarly every `set()` method needs to call the `setPropertyValue()` method. In Oracle ATG Web Commerce, all commerce objects implement the `ChangedProperties` interface except for `atg.commerce.order.AuxiliaryData` and all subclasses of `atg.commerce.pricing.AmountInfo`.

The `ChangedProperties` interface enhances performance when saving an `Order` to the Order Repository. In the example above, the call to `setPropertyValue("shortDescription", pShortDescription)` in the `setShortDescription()` method causes the `shortDescription` repository item property to be set directly when the method is called. This approach reduces the amount of processing when `OrderManager.updateOrder()` is called to save the `Order` to the repository. Performance is enhanced because you set the values directly to the repository item and only save the properties that have actually been

changed in the class. The call to `getPropertyValue("shortDescription")` retrieves the property directly from the repository item and eliminates the need to create a member variable in the class to store the value.

With the `MyCommerceItemImpl` subclass created, you now need to integrate the new commerce object into Oracle ATG Web Commerce. You can do so using one of two approaches:

- (**Recommended**) If you want your extensions in all objects, add the new properties to an existing item descriptor and then map the new object to that item descriptor. This approach is recommended because it eliminates the need to change property values that contain item descriptor names throughout Commerce. For more information, see Integrating a New Commerce Object: Using an Existing Item Descriptor (page 343), which continues the example of `MyCommerceItemImpl`.
- Create a new item descriptor subtype that includes the new properties and map the new object to it. For more information, see Integrating a New Commerce Object: Using a New Item Descriptor (page 345), which continues the example of `MyCommerceItemImpl`.

Note: You can also extend the commerce object hierarchy by subclassing `AuxiliaryData`, which holds auxiliary data for a `CommerceItem`. If you do so, you can integrate the new class into Oracle ATG Web Commerce using either process described in this section. (Recall that `AuxiliaryData` does not implement the `ChangedProperties` interface.) However, you should take the following additional steps:

- Override the `createAuxiliaryData()` method in the subclass so that it creates an instance of that new class, as follows:

```
protected void createAuxiliaryData() {
if (mAuxiliaryData == null) {
mAuxiliaryData = new MyAuxiliaryData(getRepositoryItem());
}
```

- Ensure the new properties of the subclass are defined within the XML definition for the new commerce object.
- Ensure the database columns that store the new properties of the subclass go into the table that represents the new commerce object.

Integrating a New Commerce Object: Using an Existing Item Descriptor

To integrate `MyCommerceItemImpl` into Oracle ATG Web Commerce using an existing item descriptor, follow the steps described in this section.

Extend the Order Repository Definition File

Extend the Order Repository definition file, `orderrepository.xml`, to add the new properties in `MyCommerceItemImpl` to the existing `commerceItem` item descriptor. In this example, the new property to add is the `shortDescription` property.

The `orderrepository.xml` file is found in the CONFIGPATH at `/atg/commerce/order/orderrepository.xml`. To extend the file, create a new `orderrepository.xml` file at `/atg/commerce/order/` in your `localconfig` directory. The new file should define the `shortDescription` property for the `commerceItem` item descriptor. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `orderrepository.xml` files in the CONFIGPATH into a single composite XML file. (For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.)

The `orderrepository.xml` file that you create might look as follows:

```
<gsa-template xml-combine="append">
```

```
  <item-descriptor name="commerceItem">
    <table name="dcspp_my_item" id-column-name="commerce_item_id">
      <property name="shortDescription" column-name="short_description"
                                data-type="string"/>
    </table>
  </item-descriptor>

</gsa-template>
```

The first line in the above XML example begins the GSA template and instructs the XML combiner to append the contents of the tags in this file to the contents of the tags in the file with which it is combined.

The next section defines the `shortDescription` property of a `commerceItem` repository item, as well as the database table and column that store that property.

For more information on setting up a repository and defining item descriptors, see the *ATG Platform API Reference*.

Modify the Order Repository Database Schema

In step 1, you defined the new `shortDescription` property of the `commerceItem` item descriptor, specifying the database table and column that store that property. Now you need to modify accordingly the Order Repository database schema.

The following DDL statement creates the database table and columns specified in the `orderrepository.xml` file that you created in step 1.

```
CREATE TABLE dcspp_my_item (
     commerce_item_id          VARCHAR(40)          NOT NULL
REFERENCES dcspp_item(commerce_item_id),
     short_description         VARCHAR(254)         NULL,
     PRIMARY KEY(commerce_item_id)
);
```

Modify the OrderTools Configuration File

The `OrderTools` component controls many aspects of the purchase process, such as mapping between commerce object types and class names, defining the default commerce object types, and mapping between commerce objects and item descriptors. You need to modify the `OrderTools` configuration file to support the new `MyCommerceItemImpl` class.

To modify the `OrderTools` configuration file, layer on a configuration file by creating an `OrderTools.properties` file at `/atg/commerce/order/` in your `localconfig` directory. The `OrderTools.properties` file might look as follows:

```
beanNameToItemDescriptorMap-=\
          atg.commerce.order.CommerceItemImpl=commerceItem

beanNameToItemDescriptorMap+=\
          my.class.dir.MyCommerceItemImpl=commerceItem

commerceItemTypeClassMap+=\
          default=my.class.dir.MyCommerceItemImpl
```

The `beanNameToItemDescriptorMap` property maps Order Repository item descriptors to Bean names. In Oracle ATG Web Commerce, the processors that save and load an `Order` look for an item descriptor that is

mapped to the corresponding commerce object class; the `beanNameToItemDescriptorMap` property contains this mapping. The configuration file above first removes the out-of-the-box configuration, then remaps the existing `commerceItem` item descriptor to the new Bean class, `MyCommerceItemImpl`. The `my.class.dir` prefix specifies some Java package in which the class exists.

Because you can have more than one type of `CommerceItem` object, the `commerceItemTypeClassMap` property maps `CommerceItem` types to class names. This mapping is used by the `createCommerceItem()` method in the `CommerceItemManager`; by passing it a type parameter (such as the string "default"), the method constructs and returns an instance of the corresponding class. When one of the `createCommerceItem()` methods that does not take a type parameter is called, the method constructs and returns an instance of the type specified in `OrderTools.defaultCommerceItemType`. By default, the `defaultCommerceItemType` property is set to the type `default`, which, in turn, is mapped to the new `MyCommerceItemImpl` class in the `commerceItemTypeClassMap` property in the configuration file above. The `my.class.dir` prefix indicates some Java package in which the class exists.

Integrating a New Commerce Object: Using a New Item Descriptor

To integrate `MyCommerceItemImpl` into Oracle ATG Web Commerce using a new item descriptor subtype, follow the steps in the following sections. This approach is most useful when you have an alternative subclass of the runtime object (as opposed to replacing the runtime object). If you have two object classes, you might add a new entry instead of reconfiguring the `default` entry in the `commerceItemTypeClassMap`.

Step 1 of 4 - Extend the Order Repository Definition File

Extend the Order Repository definition file, `orderrepository.xml`, to create a new item descriptor subtype that supports the new properties in `MyCommerceItemImpl`.

The `orderrepository.xml` file is found in the CONFIGPATH at `/atg/commerce/order/orderrepository.xml`. To extend the file, create a new `orderrepository.xml` file at `/atg/commerce/order/` in your `localconfig` directory. The new file should define the new item descriptor subtype. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `orderrepository.xml` files in the CONFIGPATH into a single composite XML file. (For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.)

The following `orderrepository.xml` file defines a new item descriptor named `myCommerceItem`. As a subtype of the `commerceItem` item descriptor, `myCommerceItem` inherits all of the properties of `commerceItem`. Additionally, it defines one new property, `shortDescription`.

```
<gsa-template xml-combine="append">

  <item-descriptor name="commerceItem">
    <table name="dcspp_item">
      <property name="type">
        <option value="myCommerceItem" code="1"/>
      </property>
    </table>
  </item-descriptor>

  <item-descriptor name="myCommerceItem" super-type="commerceItem"
                              sub-type-value="myCommerceItem">
    <table name="dcspp_my_item" id-column-name="commerce_item_id">
      <property name="shortDescription" column-name="short_description"
                              data-type="string"/>
    </table>
  </item-descriptor>
```

```
</gsa-template>
```

The first line in the above XML example begins the GSA template and instructs the XML combiner to append the contents of the tags in this file to the contents of the tags in the file with which it is combined.

The next section defines `myCommerceItem` as a subtype of the `commerceItem` item descriptor. You do this by adding a new string value for `myCommerceItem` to the `type` enumerated property of `commerceItem`. In this case, the new type is called `myCommerceItem`, and its corresponding integer value is 1. The base `orderrepository.xml` file contains the other options for the `type` property of `commerceItem`.

The last section of the XML file defines the `myCommerceItem` item descriptor, specifying `commerceItem` as the `super-type` (or parent item descriptor) and `myCommerceItem` as the `sub-type-value`. The section then specifies the properties of a `myCommerceItem` repository item, as well as the database table and columns that store those properties. In this case, a single property, `shortDescription`, is specified. However, recall that `myCommerceItem` inherits all of the properties of `commerceItem`, its parent item descriptor.

For more information on setting up a repository and defining item descriptors, see the *ATG Repository Guide*.

Step 2 of 4 – Modify the Order Repository Database Schema

In step 1, you created the new `myCommerceItem` item descriptor, defining both its properties and the database table and columns that store those properties. Now you need to modify accordingly the Order Repository database schema.

The following DDL statement creates the database table and columns specified in the `orderrepository.xml` file that you created in step 1.

```
CREATE TABLE dcspp_my_item (
     commerce_item_id          VARCHAR(40)          NOT NULL
REFERENCES dcspp_item(commerce_item_id),
     short_description         VARCHAR(254)         NULL,
     PRIMARY KEY(commerce_item_id)
);
```

Step 3 of 4 – Modify the OrderTools Configuration File

The `OrderTools` component controls many aspects of the purchase process, such as mapping between commerce object types and class names, defining the default commerce object types, and mapping between commerce objects and item descriptors. You need to modify the `OrderTools` configuration file to support the new `MyCommerceItemImpl` class and `myCommerceItem` item descriptor.

To modify the `OrderTools` configuration file, layer on a configuration file by creating an `OrderTools.properties` file at `/atg/commerce/order/` in your `localconfig` directory. The `OrderTools.properties` file might look as follows:

```
beanNameToItemDescriptorMap+=\
          my.class.dir.MyCommerceItemImpl=myCommerceItem

commerceItemTypeClassMap+=\
          default=my.class.dir.MyCommerceItemImpl
```

The `beanNameToItemDescriptorMap` property maps Order Repository item descriptors to Bean names. In Oracle ATG Web Commerce, the processors that save and load an `Order` look for an item descriptor that is mapped to the corresponding commerce object class; the `beanNameToItemDescriptorMap` property contains

this mapping. The configuration file above adds a new entry, mapping the `myCommerceItem` item descriptor that you created in step 1 to the `MyCommerceItemImpl` class. The `my.class.dir` prefix specifies some Java package in which the class exists.

Because you can have more than one type of `CommerceItem` object, the `commerceItemTypeClassMap` property maps `CommerceItem` types to class names. This mapping is used by the `createCommerceItem()` method in the `CommerceItemManager`; by passing it a type parameter (such as the string "default"), the method constructs and returns an instance of the corresponding class. When one of the `createCommerceItem()` methods that does not take a type parameter is called, the method constructs and returns an instance of the type specified in `OrderTools.defaultCommerceItemType`. By default, the `defaultCommerceItemType` property is set to the type `default`, which, in turn, is mapped to the new `MyCommerceItemImpl` class in the `commerceItemTypeClassMap` property in the configuration file above. The `my.class.dir` prefix indicates some Java package in which the class exists.

Step 4 of 4 – Extend the ID Spaces Definition File

Note: Because the example provided throughout this section involves an item descriptor subtype rather than a root item descriptor, this step **is not** required for the example. It is provided here for information when defining root item descriptors.

When an ID is requested for a new repository item, it is requested from the appropriate `IdSpace` for that repository item. The item descriptor's `id-space-name` attribute specifies which `IdSpace` supplies repository IDs for items of that item type. By default, all items use the item descriptor's name as the ID space unless their item type inherits from another item type. In the latter case, the items use the ID space name of the root item descriptor in the super-type hierarchy.

If the new item descriptor that you've defined is a root item descriptor, you need to extend the ID spaces definition file, `idspaces.xml`, in order to define an ID space for that item descriptor. The Oracle ATG Web Commerce `IdGenerator` guarantees that IDs within a named ID space are unique, and each root item descriptor defines the characteristics of its ID space in the `idspaces.xml` definition file.

Because the example used throughout this section involves the `myCommerceItem` item descriptor subtype, which is a subtype of the `commerceItem` item descriptor, it doesn't require a defined ID space. However, if `myCommerceItem` were a root item descriptor, you would define an ID space for it by creating a new `idspaces.xml` file at `/atg/dynamo/service/` in your `localconfig` directory. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `idspaces.xml` files in the CONFIGPATH into a single composite XML file. (For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.) The `idspaces.xml` file might look as follows:

```
<id-spaces xml-combine="append">
  <id-space name="myCommerceItem" seed="1" batch-size="10000"
                                          prefix="mci"/>
</id-spaces>
```

For more information on defining ID spaces and its impact on repository item IDs, see the *ID Generators* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

Adding a Subclass with Complex Data Type Properties

You can extend the commerce object hierarchy by subclassing an existing commerce object and adding new properties whose data types are complex objects. Unlike with adding new simple data type properties (see Adding a Subclass with Simple Data Type Properties (page 342)), adding new complex data type properties requires that you write code to save and load the object's properties.

As an example, the following code creates a new class called `OrderData`. It extends `CommerceIdentifierImpl` and adds a new `String` property called `miscInformation`. A subsequent code example creates a new class called `MyOrder`, which extends `OrderImpl` and adds a new `OrderData` property named `orderData`.

Note that the `OrderData` class implements the `ChangedProperties` interface, which is explained in detail after the code example. The properties of the `OrderData` object are stored in a corresponding `OrderData` repository item, also explained below.

```
package my_package;

import atg.commerce.order.ChangedProperties;
import atg.commerce.order.CommerceIdentifierImpl;
import java.util.Set;
import java.util.HashSet;
import java.util.Observable;
import atg.repository.MutableRepositoryItem;

public class OrderData extends CommerceIdentifierImpl
             implements ChangedProperties
{
  public OrderData() {
    super();
  }

  // property: miscInformation
  public String getMiscInformation() {
    return (String) getPropertyValue("miscInformation");
  }

  public void setMiscInformation (String pMiscInformation) {
    setPropertyValue("miscInformation", pMiscInformation);
  }

  //
  // Observer implementation
  //
  public void update(Observable o, Object arg) {
    if (arg instanceof String) {
      addChangedProperty((String) arg);
    }
    else {
      throw new RuntimeException("Observable update for " +
         getClass().getName() + " was received with arg type " +
         arg.getClass().getName() + ":" + arg);
    }
  }

  //
  // ChangedProperties implementation
  //

  // property: saveAllProperties
  private boolean mSaveAllProperties = false;

  public boolean getSaveAllProperties() {
    return mSaveAllProperties;
  }

  public void setSaveAllProperties(boolean pSaveAllProperties) {
```

```
  mSaveAllProperties = pSaveAllProperties;
}

// property: changed
private boolean mChanged = false;

public boolean isChanged() {
  return (mChanged || (mChangedProperties != null
                && ! getChangedProperties().isEmpty()));
}

public void setChanged(boolean pChanged) {
  mChanged = pChanged;
}

// property: changedProperties
private HashSet mChangedProperties = new HashSet(7);

public Set getChangedProperties() {
  return mChangedProperties;
}

public void addChangedProperty(String pPropertyName) {
  mChangedProperties.add(pPropertyName);
}

public void clearChangedProperties() {
  mChangedProperties.clear();
}

// property: repositoryItem
private MutableRepositoryItem mRepositoryItem = null;

public MutableRepositoryItem getRepositoryItem() {
  return mRepositoryItem;
}

public void setRepositoryItem(MutableRepositoryItem pRepositoryItem) {
  mRepositoryItem = pRepositoryItem;
}

// setPropertyValue/getPropertyValue methods
public Object getPropertyValue(String pPropertyName) {
  MutableRepositoryItem mutItem = getRepositoryItem();
  if (mutItem == null)
    throw new RuntimeException("Null repository item: " + getId());

  return mutItem.getPropertyValue(pPropertyName);
}

public void setPropertyValue(String pPropertyName,
                   Object pPropertyValue)
{
  MutableRepositoryItem mutItem = getRepositoryItem();
  if (mutItem == null)
    throw new RuntimeException("Null repository item: " + getId());

  mutItem.setPropertyValue(pPropertyName, pPropertyValue);
  setChanged(true);
}
```

```
}
```

As previously mentioned, the code above creates a new `OrderData` class that extends `CommerceIdentifierImpl`, implements the `ChangedProperties` interface, and adds a new `String` property called `miscInformation`.

The `CommerceIdentifierImpl` class is an abstract class that contains a single property called `id`; it is the base class for all commerce object classes. The class contains the `getId()` and `setId()` methods and implements the `CommerceIdentifier` interface, which contains only the `getId()` method. The purpose of the `id` property is to store the repository ID for the given commerce object. The `CommerceIdentifier` interface provides a standard way for Oracle ATG Web Commerce systems to access the repository IDs of items.

The `ChangedProperties` interface enhances performance when saving the object by allowing the object's property values to be set directly to the repository item. The interface contains the properties described in the following table.

Property	Description
`changed`	A `boolean` property that returns true if the object has changed since the last update and returns false if it has not.
`changedProperties`	A `Set` that contains the names of all changed properties. The property is implemented as a `Set` to include each property only once.
`repositoryItem`	Contains the object that refers to the repository item. Having the object contain the repository item eliminates the need to look up the item in the repository when saving it.
`saveAllProperties`	A `boolean` property that marks all properties as changed. This causes all properties to be saved to the repository, regardless of whether or not they have changed.

With the `OrderData` class created, the next step is to add the `OrderData` property to the `Order`. The following code creates a new class called `MyOrder`, which extends `OrderImpl` and adds a new `OrderData` property called `orderData`.

```
package my_package;

import atg.commerce.order.OrderImpl;
import atg.repository.MutableRepositoryItem;

public class MyOrder extends OrderImpl {
  public MyOrder() {
  }

  // property: orderData
  private OrderData mOrderData = null;

  public OrderData getOrderData() {
    if (mOrderData == null) {
      mOrderData = new OrderData();
```

```
      setRepositoryItem((MutableRepositoryItem) getPropertyValue("orderData"));
    }
    return mOrderData;
  }

  public void setOrderData(OrderData pOrderData) {
    mOrderData = pOrderData;
    setPropertyValue("orderData", pOrderData.getRepositoryItem());
  }
}
```

With the `OrderData` and `MyOrder` classes created, you now need to integrate the new commerce objects into Oracle ATG Web Commerce. To do so, perform the steps described in the sections that follow.

Extend the Order Repository Definition File

First, extend the Order Repository definition file, `orderrepository.xml`, to create new item and property descriptors that support the new properties in `OrderData` and `MyOrder`.

The `orderrepository.xml` file is found in the CONFIGPATH at `/atg/commerce/order/orderrepository.xml`. To extend it, add a new `orderrepository.xml` file at `/atg/commerce/order/` in your `localconfig` directory. The new file should define the new item descriptors. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `orderrepository.xml` files in the CONFIGPATH into a single composite XML file. (For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.)

The following `orderrepository.xml` file defines an item descriptor named `myOrder`. As a subtype of the `order` item descriptor, `myOrder` inherits all of the properties of `order`. Additionally, it defines one new property, `orderData`. The definition file also defines a root item descriptor named `orderData`, which supports the `miscInformation` property in `OrderData`.

```
<gsa-template xml-combine="append">

  <item-descriptor name="order">
     <table name="dcspp_my_order" id-column-name="order_id">
      <property name="orderData" column-name="order_data"
                                 item-type="orderData"/>
    </table>
  </item-descriptor>

  <item-descriptor name="orderData" sub-type-property="type"
                                 version-property="version">
    <table name="dcspp_order_data" type="primary"
                                 id-column-name="order_data_id">
       <property name="version" column-name="version" data-type="int"
                                 writable="false"/>
      <property name="miscInformation" column-name="misc_information"
                                 data-type="string"/>
    </table>
  </item-descriptor>

</gsa-template>
```

The first line in the above XML example begins the GSA template and instructs the XML combiner to append the contents of the tags in this file to the contents of the tags in the file with which it is combined.

The next section defines the `orderData` property of the order repository item, as well as the database table and column that store that property.

The last section of the XML file defines `orderData` as a root item descriptor. The section then specifies the properties of an `orderdata` repository item, as well as the database table and columns that store those properties. (Note that each property of a repository item is stored in a database column that has the same name as the property, unless otherwise specified using the `column-name` attribute.) In this case, the following properties are specified: `version` and `miscInformation`. The `version` property is defined as `readonly` (`writable="false"` in the XML) because it is used primarily by the repository. Both properties are stored in the `dcspp_my_order` database table. The `dcspp_my_order` table is declared a primary table, meaning it defines a column that stores repository IDs. In this case, that column is `order_data_id`.

Modify the Order Repository Database Schema

In step 1, you created the new `orderData` item descriptor and defined its properties, a new order property, and the database tables and columns that store those properties. Now you need to modify the Order Repository database schema.

The following DDL statements create the database tables and columns specified in the `orderrepository.xml` file that you created in step 1.

```
CREATE TABLE dcspp_my_order (
     order_id          VARCHAR(40)          NOT NULL
REFERENCES dcspp_order(order_id),
     order_data          VARCHAR(40)          NULL,
     PRIMARY KEY(order_id)
);

CREATE TABLE dcspp_order_data (
     order_data_id         VARCHAR(40)           NOT NULL,
     version               integer               NOT NULL,
     misc_information      VARCHAR(254)          NULL,
     PRIMARY KEY(order_data_id)
);
```

Subclass OrderTools and SimpleOrderManager to Create the New Object

When an `Order` is created, the new `OrderData` object must also be created and added to the `Order` object. This functionality requires the following two steps:

1. Subclass `atg.commerce.order.OrderTools` and add a new `createOrderData()` method that instantiates an `OrderData` object and creates an `OrderData` repository item in the Order Repository.

2. Subclass `atg.commerce.order.SimpleOrderManager` (which extends `atg.commerce.order.OrderManager`) and override its `createOrder()` method (there are six `createOrder()` methods; override the one with the most input parameters). The new `createOrder()` method should first call the `createOrder()` method of the superclass to create the `Order` object, then call the `createOrderData()` method that you created in step 1 to create the `OrderData` object, and finally add the `OrderData` object to the `Order`. (For more information on the `SimpleOrderManager`, see the Using the SimpleOrderManager (page 242) section of the *Working With Purchase Process Objects* (page 223) chapter.)

The following code example subclasses `OrderTools` and adds a new `createOrderData()` method.

```
package my_package;
```

```
import atg.commerce.*;
import atg.commerce.order.*;
import atg.repository.*;

public class MyOrderTools extends OrderTools
{
  public MyOrderTools() {
  }

  public OrderData createOrderData() throws ObjectCreationException
  {
    // instantiate the orderData object
    OrderData orderData = new OrderData();
    if (orderData instanceof ChangedProperties)
      ((ChangedProperties) orderData).setSaveAllProperties(true);

    // create the OrderData in the repository and set its id to the
    // repository's id
    try {
      MutableRepository mutRep = (MutableRepository) getOrderRepository();
      MutableRepositoryItem mutItem = mutRep.createItem("orderData");
      orderData.setId(mutItem.getRepositoryId());
      if (orderData instanceof ChangedProperties)
        ((ChangedProperties) orderData).setRepositoryItem(mutItem);
    }
    catch (RepositoryException e) {
      throw new ObjectCreationException(e);
    }

    return orderData;
  }
}
```

The following code example subclasses `SimpleOrderManager`, overriding its `createOrder()` method in order to call the `createOrderData()` method in the `OrderTools` object.

```
package my_package;

import atg.commerce.*;
import atg.commerce.order.*;
import atg.commerce.pricing.*;

public class MyOrderManager extends SimpleOrderManager
{
  public MyOrderManager() {
  }

  public Order createOrder(String pProfileId, String pOrderId,
         OrderPriceInfo pOrderPriceInfo, TaxPriceInfo pTaxPriceInfo,
         ShippingPriceInfo pShippingPriceInfo, String pOrderType)
         throws CommerceException
  {
    MyOrder order = (MyOrder)super.createOrder(
                        pProfileId, pOrderId, pOrderPriceInfo,
                        pTaxPriceInfo, pShippingPriceInfo, pOrderType);

    OrderData orderData = ((MyOrderTools)getOrderTools()).createOrderData();
    order.setOrderData(orderData);
```

```
    return order;
  }
}
```

Modify the OrderTools and OrderManager Configuration Files

In step 3, you subclassed `OrderTools` and `SimpleOrderManager` to create the new `OrderData` object and add it to the `Order`. Now you need to configure instances of these new classes in Nucleus.

First, configure an instance of `MyOrderManager` in Nucleus by modifying the existing `OrderManager` configuration file. To do so, layer on a configuration file by creating an `OrderManager.properties` file at `/atg/commerce/order/` in your `localconfig` directory. The `OrderManager.properties` file should look as follows (Note that no properties need to be configured.):

```
$class=my_package.MyOrderManager
```

Second, configure an instance of `MyOrderTools` in Nucleus by modifying the existing `OrderTools` configuration file. The `OrderTools` component controls many aspects of the purchase process, such as mapping between commerce object types and class names, defining the default commerce object types, and mapping between commerce objects and item descriptors. You need to modify the `OrderTools` configuration file to support the new commerce objects and item descriptors that you have created.

To modify the `OrderTools` configuration file, layer on a configuration file by creating an `OrderTools.properties` file at `/atg/commerce/order/` in your `localconfig` directory. The `OrderTools.properties` file should look as follows:

```
$class=my_package.MyOrderTools

beanNameToItemDescriptorMap-=atg.commerce.order.OrderImpl=order

beanNameToItemDescriptorMap+=\
        my.class.dir.OrderData=orderData,\
        my.class.dir.MyOrder=Order

orderTypeClassMap+=\
        default=my.class.dir.MyOrder
```

The `beanNameToItemDescriptorMap` property maps Order Repository item descriptors to Bean names. In Oracle ATG Web Commerce, the processors that save and load an `Order` look for an item descriptor that is mapped to the corresponding commerce object class; the `beanNameToItemDescriptorMap` property contains this mapping. The configuration file above first removes the out-of-the-box configuration, then remaps the existing order item descriptor to the new Bean class, `MyOrder`. The configuration file also adds a new entry, mapping the `orderData` item descriptor that you created in step 1 to the corresponding class. The `my.class.dir` prefix specifies the Java package in which the classes exist.

Because you can have more than one type of `Order` object, the `orderTypeClassMap` property maps `Order` types to class names. This mapping is used by the `createOrder()` method in the `OrderManager`; by passing it a type parameter (such as the string "default"), the method constructs and returns an instance of the corresponding class. When one of the `createOrder()` methods that does not take a type parameter is called, the method constructs and returns an instance of the type specified in `OrderTools.defaultOrderType`. By default, the `defaultOrderType` property is set to the type `default`, which, in turn, is mapped to the new `MyOrder` class in the `orderTypeClassMap` property in the configuration file above. The `my.class.dir` prefix indicates the Java package in which the class exists.

Add a Processor to Save the New Object

You do not need to write new code to save the `orderData` reference in the `MyOrder` object. The processors in the `updateOrder` pipeline, which perform the actual saving of the `Order` object to the Order Repository, use the Dynamic Beans mechanism to read and write the values of a bean using their property names. However, to save the data in the `OrderData` object itself, you must create a new processor that saves the `OrderData` object to the Order Repository and insert that new processor into the `updateOrder` pipeline. (For more information on the `updateOrder` pipeline, see Saving Orders (page 277) in the *Configuring Purchase Process Services* chapter.)

First, write the Java source code for the new processor. The following Java source file, `ProcSaveOrderDataObject.java`, serves as an example.

```
package my_package;

import atg.repository.*;
import atg.commerce.order.*;
import atg.commerce.CommerceException;
import atg.service.pipeline.*;
import atg.beans.*;
import atg.commerce.order.processor.*;

import java.util.*;

/*
This class extends a class called SavedProperties. SavedProperties provides a set
of properties and a way to retrieve a mapped property. This functionality
corresponds to the savedProperties property and the
propertyDescriptorToBeanPropertyMap property in the properties file which
corresponds the this object.

This class also implements the PipelineProcessor interface. This interface
includes the runProcess() and getRetCodes() methods. All pipeline processors must
implement this interface.
*/
public class ProcSaveOrderDataObject extends SavedProperties
                    implements PipelineProcessor
{
// These are the two valid return codes for this pipeline processor
  private final int SUCCESS = 1;
  private final int FAILURE = 2;

// The constructor
  public ProcSaveOrderDataObject() {
  }

// Returns the set of all valid return codes for this processor.
  public int[] getRetCodes()
  {
    int[] ret = {SUCCESS, FAILURE};
    return ret;
  }

// property: orderDataProperty
// This is the order data property.
  private String mOrderDataProperty = "orderData";

  public String getOrderDataProperty() {
    return mOrderDataProperty;
  }
```

```
  public void setOrderDataProperty(String pOrderDataProperty) {
    mOrderDataProperty = pOrderDataProperty;
  }

  public int runProcess(Object pParam, PipelineResult pResult)
                      throws Exception
  {
/*
The pParam parameter contains the data   required for executing this pipeline
processor. This code extracts the required parameters for this processor.
*/
    HashMap map = (HashMap) pParam;
    Order order = (Order) map.get(PipelineConstants.ORDER);
    OrderManager orderManager = (OrderManager)
                        map.get(PipelineConstants.ORDERMANAGER);
    Repository repository = (Repository)
                        map.get(PipelineConstants.ORDERREPOSITORY);
    OrderTools orderTools = (OrderTools) orderManager.getOrderTools();

    MutableRepository mutRep = null;
    MutableRepositoryItem mutItem = null;
    Object value = null;
    Object[] savedProperties = null;
    String mappedPropName = null;

// Check for null parameters
    if (order == null)
      throw new InvalidParameterException();
    if (repository == null)
      throw new InvalidParameterException();
    if (orderManager == null)
      throw new InvalidParameterException();

/*
Try to cast the repository and make sure that it is a MutableRepository.
In most cases it will be a GSA Repository which is mutable.
*/
    try {
      mutRep = (MutableRepository) repository;
    }
    catch (ClassCastException e) {
      throw e;
    }

/*
This code is taking advantage of the ChangedProperties interface methods. The
first check is checking whether this processor is configured to save all
properties always, or take advantage of ChangedProperties. The
saveChangedPropertiesOnly property is inherited from SavedProperties. If
getSaveChangedPropertiesOnly() returns false, then the local variable
savedProperties is set to the entire list of properties defined in the
SaveOrderDataObject.properties file. If it returns true, then a check is done to
determine whether the orderData object implements ChangedProperties. If so, then
it gets the list of properties whose value has changed, otherwise it sets the list
of properties to save to the savedProperties property.
*/
    CommerceIdentifier orderData = (CommerceIdentifier)
      DynamicBeans.getPropertyValue(order, getOrderDataProperty());
    if (getSaveChangedPropertiesOnly()) {
```

```
      if (orderData instanceof ChangedProperties
         && (! ((ChangedProperties) orderData).getSaveAllProperties()))
       savedProperties =
        ((ChangedProperties) orderData).getChangedProperties().toArray();
      else
        savedProperties = getSavedProperties();
    }
    else {
      savedProperties = getSavedProperties();
    }

/*
Next a check is done to get the repositoryItem property from the object if it has
a repositoryItem property defined. If it does not have the repositoryItem property
or its value is null, then a lookup is done in the repository for the item and set
if it has the property.
*/
    boolean hasProperty = false;
    if (DynamicBeans.getBeanInfo(orderData).hasProperty("repositoryItem"))
    {
      hasProperty = true;
      mutItem = (MutableRepositoryItem)
              DynamicBeans.getPropertyValue(orderData, "repositoryItem");
    }

    if (mutItem == null) {
      mutItem = mutRep.getItemForUpdate(orderData.getId(),
           orderTools.getMappedItemDescriptorName(
           orderData.getClass().getName()));
      if (hasProperty)
        DynamicBeans.setPropertyValue(orderData, "repositoryItem",
                                      mutItem);
    }

/*

/*
Here the repository item is updated to the repository.
This is done here to catch any Concurrency exceptions.
*/
    try {
      mutRep.updateItem(mutItem);
    }
    catch (ConcurrentUpdateException e) {
      throw new CommerceException("Concurrent Update Attempt", e);
    }

/*
Finally, the ChangedProperties Set is cleared and the saveAllProperties property
is set to false. This resets the object for more edits. Then the SUCCESS value is
returned.
*/
    if (orderData instanceof ChangedProperties) {
      ChangedProperties cp = (ChangedProperties) orderData;
      cp.clearChangedProperties();
      cp.setSaveAllProperties(false);
    }

    return SUCCESS;
  }
```

```
}
```

Next, configure an instance of `ProcSaveOrderDataObject` by adding a `SaveOrderDataObject.properties` file to your `localconfig` directory at `/atg/commerce/order/processor/`. The configuration file might look as follows:

```
$class=my_package.ProcSaveOrderDataObject

# This property tells the processor to only save the properties which have
# changed. This requires that when a property is changed that it be marked
# for saving.
saveChangedPropertiesOnly=true

orderDataProperty=orderData
```

Finally, insert the `SaveOrderDataObject` processor into the `updateOrder` pipeline. The `updateOrder` pipeline is defined in the commerce pipeline definition file, `commercepipeline.xml`. In Oracle ATG Web Commerce, this file is located at `<ATG10dir>/DCS/config/atg/commerce/`.

To insert the `SaveOrderDataObject` processor into the `updateOrder` pipeline, extend the `commercepipeline.xml` file by creating a new `commercepipeline.xml` file that defines the new processor and placing it in your `localconfig` directory at `/atg/commerce/`. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `commercepipeline.xml` files in the CONFIGPATH into a single composite XML file. Insert the new processor into the pipeline somewhere after the first processor, `updateOrderObject`, and before the last processor, `setLastModifiedTime`. By convention, you should insert the processor into the pipeline immediately after the processor that references the new object. In this example, the most appropriate place would be immediately after the `updateOrderObject` processor.

For more information on how to add a processor to an existing pipeline, refer to *Processor Chains and the Pipeline Manager* (page 363) chapter. For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

Add a Processor to Load the New Object

You do not need to write new code to load the `orderData` reference in the `Order` object. The `loadOrderObject` processor in the `loadOrder` pipeline knows how to load all of the properties of the `Order` object, regardless of whether the commerce object hierarchy has been extended. However, to load the data into the `OrderData` object itself, you must create a new processor that loads the data from the repository into the `OrderData` object and insert that new processor into the `refreshOrder` pipeline, which performs the actual loading of most of the contained objects in the `Order`. (For more information on loading and refreshing orders, see Loading Orders (page 264) in the *Configuring Purchase Process Services* chapter.)

First, write the Java source code for the new processor. The following Java source file, `ProcLoadOrderDataObject.java` serves as an example.

```
package my_package;

import atg.repository.*;
import atg.commerce.order.*;
import atg.service.pipeline.*;
import atg.beans.*;
import atg.commerce.order.processor.*;

import java.util.*;
```

```
/*
This class extends a class called LoadProperties. LoadProperties provides a set of
properties and a way to retrieve a mapped property. This functionality corresponds
to the loadProperties property and the propertyDescriptorToBeanPropertyMap
property in the properties file which corresponds the this object.

This class also implements the PipelineProcessor interface. This interface
includes the runProcess() and getRetCodes() methods. All pipeline processors must
implement this interface.
*/
public class ProcLoadOrderDataObject extends LoadProperties
            implements PipelineProcessor
{
// These are the two valid return codes for this pipeline processor
  private final int SUCCESS = 1;
  private final int FAILURE = 2;

// The constructor
  public ProcLoadOrderDataObject() {
  }

// Returns the set of all valid return codes for this processor.
  public int[] getRetCodes()
  {
    int[] ret = {SUCCESS, FAILURE};
    return ret;
  }

  // property: orderDataProperty
  private String mOrderDataProperty = null;

  public String getOrderDataProperty() {
    return mOrderDataProperty;
  }

  public void setOrderDataProperty(String pOrderDataProperty) {
    mOrderDataProperty = pOrderDataProperty;
  }

  public int runProcess(Object pParam, PipelineResult pResult)
                       throws Exception
  {
/*
The pParam parameter contains the data required for executing this pipeline
processor. This code extracts the required parameters for this processor.
*/
    HashMap map = (HashMap) pParam;
    Order order = (Order) map.get(PipelineConstants.ORDER);
    MutableRepositoryItem orderItem = (MutableRepositoryItem)
               map.get(PipelineConstants.ORDERREPOSITORYITEM);
    OrderManager orderManager = (OrderManager)
               map.get(PipelineConstants.ORDERMANAGER);
    OrderTools orderTools = orderManager.getOrderTools();

// Check for null parameters
    if (order == null)
      throw new InvalidParameterException();
    if (orderItem == null)
      throw new InvalidParameterException();
```

```
    if (orderManager == null)
      throw new InvalidParameterException();

/*
This section of code first gets the orderData item descriptor from the order
repository item. Next it gets the orderData item descriptor. The third line of
code does a lookup in the OrderTools object and returns the class mapped to the
orderData item descriptor.
*/
    MutableRepositoryItem mutItem = (MutableRepositoryItem)
          orderItem.getPropertyValue(getOrderDataProperty());
    RepositoryItemDescriptor desc = mutItem.getItemDescriptor();
    String className =
          orderTools.getMappedBeanName(desc.getItemDescriptorName());

/*
Next, an instance of OrderData is constructed, its id property set to the
repository item's id, and if the object has a repositoryItem property, the item
descriptor is set to it.
*/
    CommerceIdentifier ci = (CommerceIdentifier)
           Class.forName(className).newInstance();
    DynamicBeans.setPropertyValue(ci, "id", mutItem.getRepositoryId());
    if (DynamicBeans.getBeanInfo(ci).hasProperty("repositoryItem"))
        DynamicBeans.setPropertyValue(ci, "repositoryItem", mutItem);

If the loaded object implements ChangedProperties, then clear the
changedProperties Set. Then set the orderData property in the Order object to ci.
Finally, return SUCCESS.
*/
    if (ci instanceof ChangedProperties)
      ((ChangedProperties) ci).clearChangedProperties();
    DynamicBeans.setPropertyValue(order, getOrderDataProperty(), ci);

    return SUCCESS;
  }
}
```

Next, configure an instance of `ProcLoadOrderDataObject` by adding a `LoadOrderDataObject.properties` file to your `localconfig` directory at `/atg/commerce/order/processor/`. The configuration file might look as follows:

```
$class=my_package.ProcLoadOrderDataObject

# This property maps a OrderRepository property descriptor to an Order
# bean property. By default the processor will look for an OrderRepository
# property descriptor which is the same as the bean property name. If
# there are any properties whose names differ, they can be mapped here.
# The format is repository_property_descriptor=bean_property_name
# The repository_property_descriptor name must be listed above in
# loadProperties.
orderDataProperty=orderData
```

Finally, insert the `LoadOrderDataObject` processor into the `refreshOrder` pipeline. The `refreshOrder` pipeline is defined in the commerce pipeline definition file, `commercepipeline.xml`. In Oracle ATG Web Commerce, this file is located at `<ATG10dir>/DCS/config/atg/commerce/`.

To insert the `LoadOrderDataObject` processor into the `refreshOrder` pipeline, extend the `commercepipeline.xml` file by creating a new `commercepipeline.xml` file that defines the new processor and placing it in your `localconfig` directory at `/atg/commerce/`. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `commercepipeline.xml` files in the CONFIGPATH into a single composite XML file. Insert the new processor into the pipeline somewhere after the first processor, `loadOrderObjectForRefresh`. By convention, you should insert the processor into the pipeline immediately after the processor that references the new object. In this example, the most appropriate place would be immediately after the `loadOrderObjectForRefresh` processor.

As previously mentioned, for more information on how to add a processor to an existing pipeline, refer to the *Processor Chains and the Pipeline Manager* (page 363) chapter. For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming Guide*.

Extend the ID Spaces Definition File

Note: This step is only necessary when a new item descriptor is a root item descriptor. It does not need to be performed when the new item descriptor is a subclass of an item descriptor.

When an ID is requested for a new repository item, it is requested from the appropriate `IdSpace` for that repository item. The item descriptor's `id-space-name` attribute specifies which `IdSpace` supplies repository IDs for items of that item type. By default, all items use the item descriptor's name as the ID space unless their item type inherits from another item type. In the latter case, the items use the ID space name of the root item descriptor in the super-type hierarchy.

If a new item descriptor that you've defined is a root item descriptor, you need to modify the ID spaces definition file, `idspaces.xml`, in order to define an ID space for that item descriptor. Oracle ATG Web Commerce `IdGenerator` guarantees that IDs within a named ID space are unique, and each root item descriptor defines the characteristics of its ID space in the `idspaces.xml` definition file.

In the example used throughout this section, you've defined a single root item descriptor, `orderData`. Consequently, you need to define an ID space for that descriptor. To do so, create a new `idspaces.xml` file at `/atg/dynamo/service/` in your `localconfig` directory. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the `idspaces.xml` files in the CONFIGPATH into a single composite XML file. (For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming Guide*.) The `idspaces.xml` file might look as follows:

```
<id-spaces xml-combine="append">
    <id-space name="orderData" seed="1" batch-size="10000"
                                            prefix="od"/>
</id-spaces>
```

For more information on defining ID spaces and its impact on repository item IDs, see the *ID Generators* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

Manipulating Extended Objects

Regardless of the method you use to extend purchase process objects (using an existing or new item descriptor), you may need to extend some additional areas as well:

- If you extend an implementation of `CommerceItem` or `ShippingGroup`, you may need to make additional changes as described in Merging Orders (page 362) in this chapter.
- If you add custom properties to a `CommerceItem`, you may need to override portions of `CartModifierFormHandler`. This form handler is capable of handling new primitive data type properties

automatically. Other types of extensions require extensions to the form handler. See *Handling Custom Commerce Item Properties* in the *Implementing Shopping Carts* chapter of the *ATG Commerce Guide to Setting Up a Store* for more information.

Merging Orders

The standard process for merging orders involves `OrderManager.mergeOrders()` calling `CommerceItemManager.mergeOrdersCopyCommerceItem()` to copy existing commerce items from a source order to a destination order, and calling `ShippingGroupManager.mergeOrdersCopyShippingGroup` to copy existing shipping groups from the source order to the destination order. In the `ShippingGroupManager`, `mergeOrdersCopyShippingGroup`, in turn, calls either `mergeOrdersCopyHardgoodShippingGroup` or `mergeOrdersCopyElectronicShippingGroup`, depending on the type of the shipping group.

If you are using the multisite features of Oracle ATG Web Commerce, merging orders must also take site information into account when deciding whether two items are identical, and therefore whether to increment the item count or to add a second item to the order. The methods used are `CommerceItemManager.shouldMergeItems()` and `shouldMergeSubItems()`.

Note that if you have extended certain Commerce classes to store additional information, you must change the methods used for merging orders so the additional information is copied to the destination order when merging orders.

- If you've extended `CommerceItem`, you should subclass `CommerceItemManager` and override the `mergeOrdersCopyCommerceItem` method to first call the superclass method to do the basic copy and then copy your extended commerce item data.
- If you've extended `HardgoodShippingGroup`, you should subclass `ShippingGroupManager` and override the `mergeOrdersCopyHardgoodShippingGroup` method to first call the superclass method to do the basic copy and then copy your extended commerce item data.
- If you've extended `ElectronicShippingGroup`, you should subclass `ShippingGroupManager` and override the `mergeOrdersCopyElectronicShippingGroup` method to first call the superclass method to do the basic copy and then copy your extended commerce item data.
- If you've created a new shipping group type that is neither a subclass of `HardgoodShippingGroup` nor `ElectronicShippingGroup`, you should subclass `ShippingGroupManager` and override the `mergeOrdersCopyShippingGroup` method to examine the class of the shipping group. If the shipping group is of the new type you created, then the subclass should copy it to the destination order. Otherwise, the method can call the superclass `mergeOrdersCopyShippingGroup` method to handle the standard shipping group types.

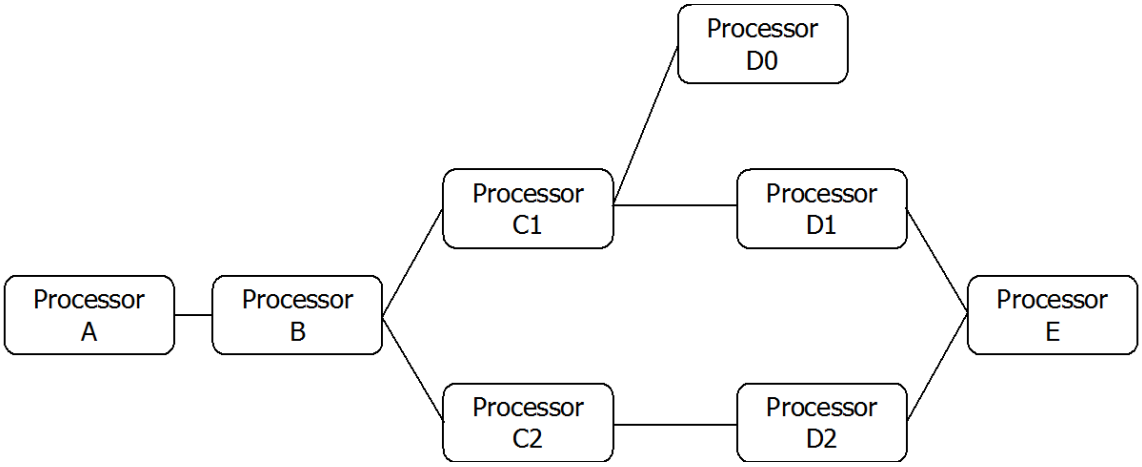
17 Processor Chains and the Pipeline Manager

The Pipeline Manager is a system that executes a series of *processors*, which are linked in processor chains. A processor is a component that executes a piece of functionality and returns a status code. The status code determines which processor in the chain to execute next. The Pipeline Manager enables you to dynamically add and remove processors and chains of processors. The Pipeline Manager does so in a transactionally aware way, supporting a subset of the transactional modes that EJB supports.

This chapter includes the following sections:

Pipeline Manager Overview

The Pipeline Manager controls a series of processors, which make up a processor chain. Each processor in the processor chain is a component that performs a specific function and returns a status code. The status code can determine which processor to execute next. You can imagine a tree structure of processor chains as below:



The processors in the processor chain illustrated above would be executed from left to right. Notice that chains are allowed to split and merge together.

For a more concrete example, suppose you have a commerce site. It might include a processor chain that is invoked when users add an item to their shopping cart. This *Add to Cart* chain might contain the following elements in the given order: `InventoryCheck`, `InventoryReserve`, `FraudDetection`, and `AddToOrder`.

If there was already an existing transaction, when it reaches the Pipeline Manager, it will simply use that transaction for executing the pipeline. If a transaction had not been created, then the Pipeline Manager creates a new one. Next, it calls each processor in sequence from `InventoryCheck` to `AddToOrder` without altering the transaction in any way.

An element in a processor chain can specify that it should be executed in the context of its own transaction. For example, the `InventoryReserve` processor uses its own transaction to access the inventory. When the request reaches `InventoryReserve`, the Pipeline Manager suspends the current transaction, which would be the one that was passed to the pipeline manager, creates a new transaction, and executes the code for `InventoryReserve`.

When the `InventoryReserve` processor completes execution of its transaction, then the Pipeline Manager either commits or rolls back the `InventoryReserve` transaction, and then resumes the original transaction before it executes the `FraudDetection` processor.

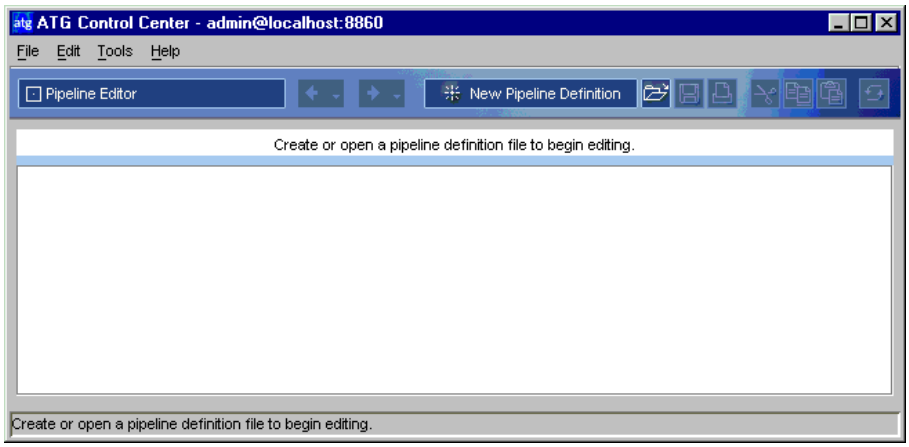
Using the Pipeline Editor

The Pipeline Editor in the ATG Control Center provides a graphical user interface for creating, viewing and editing pipeline chains. This section covers the following topics:

- Accessing the Pipeline Editor (page 365)
- Opening an Existing Pipeline Definition (page 365)
- Creating a New Pipeline Definition (page 366)
- Editing Existing Pipeline Definitions (page 368)

Accessing the Pipeline Editor

The pipeline editor is part of the ATG Control Center. To access the pipeline editor, select Utilities > Pipeline Editor from the navigation menu of the ACC. The Pipeline Editor opens:



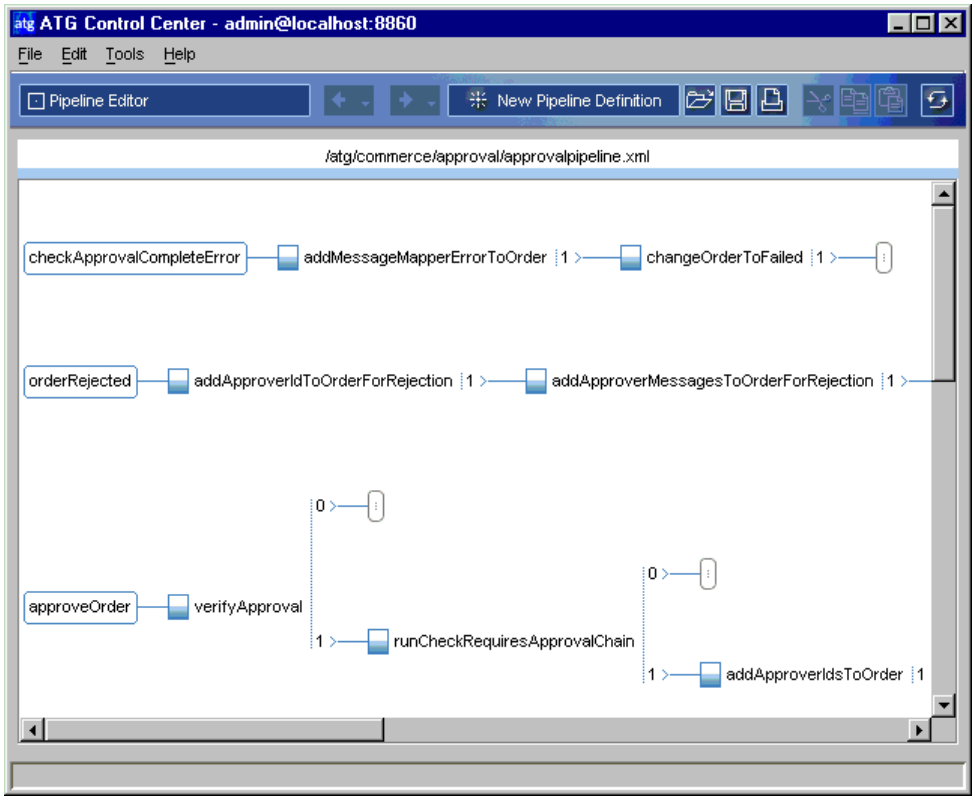
Opening an Existing Pipeline Definition

The following steps describe how to open an existing pipeline definition.

Note: The pipeline manager component must be running for its definition to be loaded. If you have problems loading an existing definition, make sure its manager is running.

1. Click on the Open Pipeline Definition icon.
2. Select the Pipeline Manager that you want to open from the list and click on the Open Pipeline Definition button.

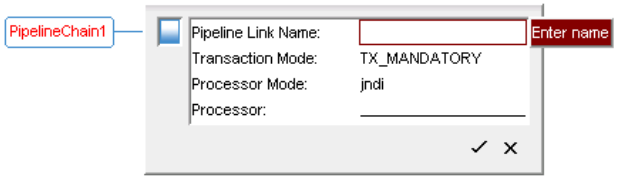
The pipeline definition displays in the ACC.



Creating a New Pipeline Definition

Follow these steps to create a new pipeline definition:

1. Click the New Pipeline Definition button. You see an empty definition with a single empty chain named PipelineChain1.
2. Right-click the Pipeline Chain title icon and select Edit Details.
3. Type a new Pipeline Chain name and select a new Transaction Mode, if necessary.
4. Right-click the line between the new pipeline chain title and termination point and select Create Pipeline Link.



5. Enter a new pipeline link name and change the Transaction Mode and Processor Mode, if necessary.
6. *(JNDI Processor Mode only)* Enter a processor for the link or click Choose Processor to select one from the available processors.
7. Click the check mark to close the pipeline link window. The pipeline editor will automatically display the appropriate number of transitions for the selected processor.

8. Repeat the steps for creating pipeline chains and links for the rest of the pipeline definition. The first pipeline chain node is automatically inserted for the user. To create new chain nodes, select Edit >Insert Pipeline Chain or right-click any execution line and select Insert Pipeline Chain.

9. Save the pipeline definition by selecting one of the following from the File menu:

   - Save all: The complete definition (all chains and their processors) is completely written out to XML and all chains are marked as "xml-combine=replace". This ensures that this definition gets accepted when xml combination happens.

   - Save diffs: Saves only the difference between the latest changes and the XML combined version of the definition file up to the previous layer. This ensures that information is not saved to the current layer that is already included in other layers.

   Note: If you create a new definition and later want to reopen it, you must associate the new XML file with an existing pipeline manager. This is because you can only open definitions associated with pipeline managers. You can create a new pipeline manager and set its definition file to point to your new definition.

Adding New Pipeline Managers and Processors to the ACC

The Pipeline Registry is used to register pipeline manager and pipeline processor components with the Pipeline Editor. If you do not register new components, you will not be able to see and select them in the editor. The Pipeline Registry component is located at `/atg/registry/PipelineRegistry` and its associated XML file is `pipelineRegistry.xml`.

To register a new pipeline manager or processor, you need to specify its Nucleus path in the XML file. You can optionally specify additional properties for processors, including a display name, category, description, and icon. These new properties will be used in the processor picker dialog box (edit a link node and for its Processor property select Choose Processor). The picker lists processors by category. Processors are shown with their icon and the description is displayed in a tool tip.

The following example registers a new manager called `MyPipelineManager` and a new processor called `MyProcessor`:

```
<pipeline-registry-configuration>

 <pipeline-manager-registry xml-combine="append-without-matching">
  <pipeline-manager>
      <nucleus-path>/atg/commerce/MyPipelineManager</nucleus-path>
  </pipeline-manager>
  </pipeline-manager-registry>

  <processor-registry xml-combine="append-without-matching">
   <processor>
      <nucleus-path>/atg/commerce/payment/MyProcessor</nucleus-path>
      <display-name>My Processor</display-name>
      <category>Payment</category>
      <description>A Custom Payment Processor</description>
    </processor>
</processor-registry>

</pipeline-registry-configuration>
```

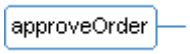
Editing Existing Pipeline Definitions

Pipelines in the ACC are edited in a way similar to the way scenarios are edited. You can perform the following editing functionality on editor nodes.

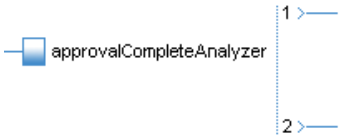
- Copy and paste
- Drag and drop processor elements
- Undo and redo actions

Overview of Pipeline Editor Interface

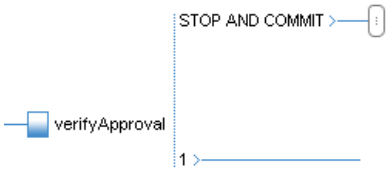
This section describes the icons in the pipeline editor.



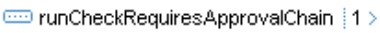
The icon above represents the name of the pipeline chain. You can edit the properties of this node to set the chain's name and transaction mode.



The icon above represents an individual link in a pipeline chain. You can edit the properties of this node to set the link's name, transaction mode, processor mode, and processor name. Once you have specified the processor, the set of possible integer outcomes (as defined by the interface `atg.service.pipeline.PipelineProcessor`) are automatically represented as part of the node with separate execution paths.



The icon above represents a link whose processor returns a special value. There are two predefined return codes with a special meaning: 0 (stop chain execution and commit) and –1 (stop chain execution and rollback). These are defined in `atg.service.pipeline.PipelineProcessor`.



The icon above represents a special link whose processor is an instance of `atg.commerce.order.processor.ProcExecuteChain`. This processor executes a subchain and returns.

The icon above represents a stop in the execution of this path.

The icon above represents a jump back to a prior link in the chain. You can edit the node to change the destination.

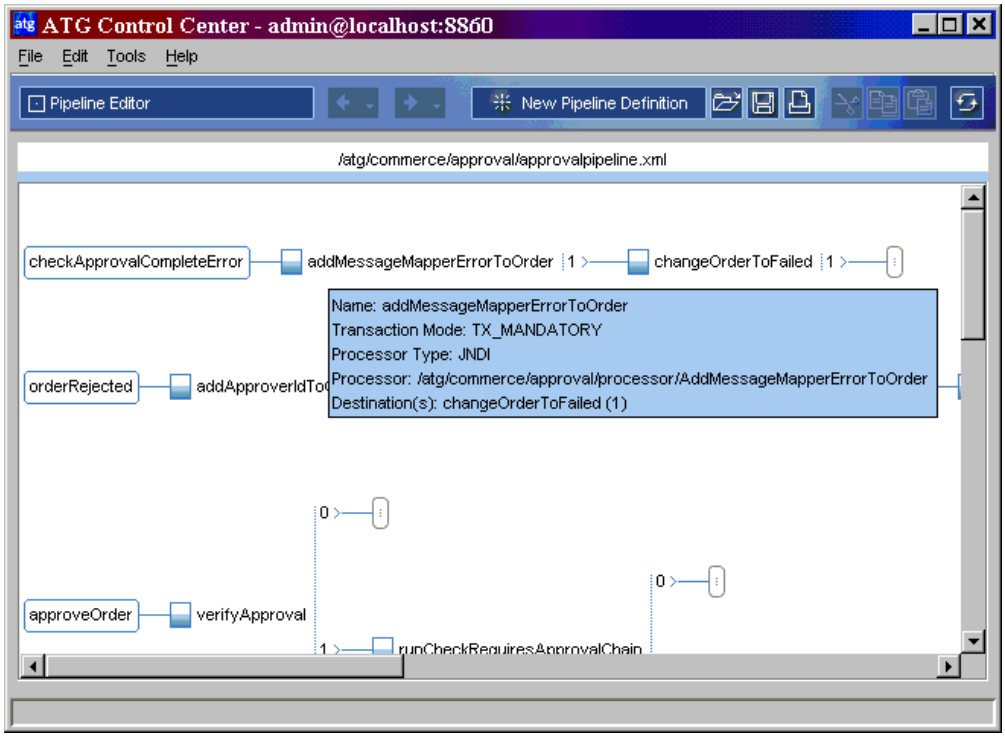
Printing a Pipeline Definition

Follow these steps to print the pipeline chains in the open pipeline definition. You can print any definition in the editor window.

1. Click the Print icon.
2. Select one of the following and click OK:
   - Scale to Page: Prints the entire pipeline definition on one page.
   - No Scaling: Prints the pipeline definition at full size on multiple pages (if necessary).

Activating Verbose Mode

Verbose mode allows you to view detailed information for pipeline nodes by moving your cursor over the node. The information appears in a pop-up window. The same information is available by double-clicking a node or right-clicking and choosing Edit Details. In addition, the verbose tool tip lists the headlink for pipeline chains, and the possible destinations for a pipeline link. A link's destination is the next link to be executed if the current processor returns a certain value.



By default, verbose mode is turned off. You can activate verbose mode by selecting File > Set Verbose Mode On. When verbose mode is active, you can turn it off by selecting File > Set Verbose Mode Off.

Pipeline Debugging

You can access the component editors for pipeline manager and pipeline processor components to turn debugging on and off.

Follow these steps to debug a definition.

1. Select File > Edit pipeline manager component.
2. Select a pipeline manager.
3. In the component editor, set `loggingDebugOnProcessors` to true to turn debugging on for all processors in the definition.

Follow these steps to debug an individual processor.

1. Right-click a pipeline link node.
2. Select Edit Selected Pipeline Processor Component.
3. In the component editor, set `loggingDebug` to true.

Changing the Display Font of the Pipeline Editor

You can change the font that the pipeline editor uses to display the pipeline definitions in the ACC as follows:

1. Select Tools > Preferences.
2. Click the Fonts tab.
3. Select a new font for the Scenario Editor. The Pipeline Editor uses the same font settings.

Reinitializing the Pipeline Manager

When you deploy an application that includes Oracle ATG Web Commerce, the pipeline manager is automatically reinitialized. If you make changes to a pipeline definition and you want the changes to take affect in the session you are currently running, you must reinitialize the pipeline manager.

Follow these steps to reinitialize the pipeline manager.

1. Open an existing pipeline definition.
2. Modify the definition. For example, delete a link in a chain or change a link to execute a different processor.
3. Click Save to save your changes.
4. Click the Reinitialize button to load the new definition into the pipeline manager.

Running a Processor Chain

To execute a processor chain in the `PipelineManager`, make a call to the `runProcess()` method in the `PipelineManager`. This section describes what happens when a processor chain is executed.

The `PipelineManager` first finds the requested chain. If the requested chain is enabled, `PipelineManager` calls `runProcess()` on it and passes it the user data. The `runProcess()` method in the `PipelineChain` manages execution of the processors. If is not enabled, an exception is thrown.

A `PipelineResult` object or subclass of it is constructed. This is determined by what is defined in the `PipelineChain` object.

The `runProcess()` method is called on the head `PipelineLink` object of the processor chain. The `PipelineLink` is responsible for three actions:

- Handling the transaction (if required)
- Executing the processor, and
- Returning the return code to the `PipelineChain`.

To handle the transaction, four methods are called before and after the call to `runProcess()` in the `PipelineProcessor`. These methods are called in the following order:

- `preChainTransaction()`
- `postChainTransaction()`
- `preLinkTransaction()`
- `postLinkTransaction()`

These calls make the appropriate calls and construct any required objects for the execution of the processor in the context of the transaction mode.

The `runProcess()` method is called between these methods and the return value of the call is returned to the `PipelineChain`. The return value of 0, which is mapped to the static variable `PipelineProcessor.STOP_CHAIN_EXECUTION`, tells the `PipelineManager` that execution for this chain should stop normally. Any other return value from `runProcess()` indicates the next processor to execute.

`STOP_CHAIN_EXECUTION` can be returned from any `PipelineProcessor`, regardless of whether it has additional links or not. When `STOP_CHAIN_EXECUTION` is returned, the transactions are handled and the `PipelineResult` object is returned. If a value other than `STOP_CHAIN_EXECUTION` is returned, the `PipelineChain` calls `getNextLink()` on the `PipelineLink` just executed and passes it the return value it received.

This call returns another `PipelineLink` to execute or null. If a `PipelineLink` is returned, then the above process repeats. If null is returned, then an exception is thrown, the transaction is rolled back if needed, and this method exits throwing the appropriate exception object.

Creating a Processor Pipeline

A processor pipeline consists of a Pipeline Manager and a set of processors, which can be organized into processor chains. You can create and delete pipelines in two ways, through an XML file or through the `atg.service.pipeline` API.

You can initialize a Pipeline Manager with a set of processor chains at application deployment using an XML configuration file called a *pipeline definition file*. This provides you with a simple way to construct and manage

the global `PipelineManager` without writing code. It only can be used for initializing the globally scoped `PipelineManager` that is created during deployment. For more information, see the Pipeline Definition Files (page 373) section.

Any non-globally scoped Pipeline Managers need to be created using the API. The API is provided for dynamic creation, editing, and deletion of processor chains and processors. For more information, see the Creating and Editing Processor Chains Programmatically (page 377) section.

The following sections describe how to create a processor pipeline:

- Configuring a Pipeline Manager (page 372)
- Creating Processors (page 372)
- Pipeline Definition Files (page 373)
- Creating and Editing Processor Chains Programmatically (page 377)
- Extending the PipelineChain and PipelineResult Classes (page 380)

Configuring a Pipeline Manager

Each processor chain is controlled by a Pipeline Manager component. If you are using Oracle ATG Web Commerce, a Pipeline Manager instance is located at `/atg/commerce/PipelineManager`. The Pipeline Manager has two properties you may want to configure: `definitionFile` and `chainLockWaitTimeout`.

definitionFile Property

The `definitionFile` property points to the XML file that defines the processor chains that can be run by the Pipeline Manager. For example:

```
definitionFile=/atg/userprofiling/registrationPipeline.xml
```

The value of this property is a file pathname, relative to your CONFIGPATH. This XML file can be anywhere in the CONFIGPATH. The Pipeline Definition Files (page 373) section describes how to create the XML file that defines the processor chains controlled by a Pipeline Manager.

chainLockWaitTimeout Property

The `chainLockWaitTimeout` property determines the amount of time the request handling thread should wait for a processor chain to execute its processing of the request. The value of this property is in milliseconds. The default setting is:

```
chainLockWaitTimeout=15000
```

Creating Processors

Any Java class can act as a processor in a pipeline by implementing the `atg.service.pipeline.PipelineProcessor` interface. The `PipelineProcessor` interface has a single method:

```
int runProcess(Object pParam, PipelineResult pResult)
```

The `runProcess()` method returns an integer status code that the Pipeline Manager uses to determine the next processor to run.

Pipeline Definition Files

The contents of the processor chains controlled by a Pipeline Manager can be determined programmatically, as described in the next section, Creating and Editing Processor Chains Programmatically (page 377), or can be configured by an XML definition file, as specified by the Pipeline Manager's `definitionFile` property. This section describes how to create an XML definition file for a Pipeline Manager.

Configuring processor chains with a pipeline definition file is useful for creating chains that are not edited or creating generic chains that will later be edited dynamically using the API based on some other criteria. This configuration file must be written in XML.

A pipeline definition file can use the following tags:

Tag	Description	Attributes
`PipelineManager`	The top level tag that encloses a definition of a Pipeline Manager.	none
`pipelinechain`	Tag defining a given processor chain in the Pipeline Manager. A Pipeline Manager can include any number of processor chains, each defined by a `<pipelinechain>` tag.	**name** - (required) The name of the processor chain (for example, `AddToCart`). Must be unique. This corresponds to the `id` in the `PipelineChain` object. **headlink** - (required) The first processor in the chain to be executed. **transaction** - The default transactional mode of all the processors in this chain. The valid modes are: TX_REQUIRED TX_REQUIRES_NEW TX_SUPPORTS TX_NOT_SUPPORTED TX_MANDATORY **classname** - The full name of a Java class which is to be instantiated and used as the `PipelineChain` object. The default is `atg.service.pipeline.PipelineChain`. The value must be this class or a subclass of it. **resultclassname** - The full name of a Java class which is to be instantiated and used as the `PipelineResult` object. The default is `atg.service.pipeline.PipelineResult`. The value must implement `PipelineResult`.

Tag	Description	Attributes
`pipelinelink`	Defines a processor within the chain and names it.	**name** - (required) A name for this processor (for example, `CheckInventory`). Must be unique. **transaction** - A transactional mode that overrides the default mode of the chain. The valid modes are: TX_REQUIRED TX_REQUIRES_NEW TX_SUPPORTS TX_NOT_SUPPORTED TX_MANDATORY
`processor`	The name of the `PipelineProcessor` object.	**class** - The processor class to be instantiated or referenced to. If the attribute is of the form `packagename.classname` then a new object is to be created. If it is of the form `jndi://.../...`, then the object is resolved through JNDI and its reference is used as the pipeline element. **jndi** - The processor class to be referenced to. The object is resolved through JNDI and its reference is used as the processor.
`transition`	A reference to the next link to be executed mapped by a return value.	**returnvalue** - (required) An integer string that is used to define the next pipeline element. **link** - (required) The name of a `pipelineprocessor` that will be executed if the return value of the current `pipelineprocessor` matches the `returnvalue` of this link.

Document Type Definition for Pipeline Definition Files

Pipeline definition files use the following XML document type definition:

```
<!ENTITY %transactionmodes
        "(TX_REQUIRED|TX_REQUIRES_NEW|TX_SUPPORTS|TX_NOT_SUPPORTED|TX_MANDATORY)">

<!ELEMENT PipelineManager (pipelinechain*)>

<!ELEMENT pipelinechain (pipelinelink*)>
<!ATTLIST pipelinechain
          name ID #REQUIRED
          headlink IDREF #REQUIRED
          transaction %transactionmodes; "TX_REQUIRED"
          classname CDATA "atg.service.pipeline.PipelineChain"
          resultclassname CDATA "atg.service.pipeline.PipelineResultImpl">

<!ELEMENT pipelinelink (processor,transition*)>
<!ATTLIST pipelinelink
          name ID #REQUIRED
          transaction %transactionmodes; "TX_REQUIRED">
```

```
<!ELEMENT processor EMPTY>
<!ATTLIST processor
          class CDATA #IMPLIED
          jndi CDATA #IMPLIED>

<!ELEMENT transition EMPTY
<!ATTLIST transition
          returnvalue CDATA #REQUIRED
          link IDREF #REQUIRED>
```

Pipeline Definition File Example

The following file, `PipelineManager.xml`, is an example of a pipeline definition file that might be used for initializing a pipeline.

```
<?xml version="1.0"?>
<!DOCTYPE PipelineManager SYSTEM "PipelineManager.dtd">

<PipelineManager>
    <pipelinechain name="AddToCart" transaction="TX_REQUIRED" headlink="proc1">
         <pipelinelink name="proc1">
              <processor class="atg.commerce.addA"/>
              <transition returnvalue="1" link="proc2"/>
              <transition returnvalue="2" link="proc3"/>
         </pipelinelink>
         <pipelinelink name="proc2" transaction="TX_REQUIRES_NEW">
              <processor class="atg.commerce.addB"/>
              <transition returnvalue="1" link="proc4"/>
              <transition returnvalue="2" link="proc5"/>
         </pipelinelink>
         <pipelinelink name="proc3">
              <processor class="atg.commerce.addE"/>
              <transition returnvalue="1" link="proc6"/>
              <transition returnvalue="2" link="proc7"/>
              <transition returnvalue="3" link="proc2"/>
         </pipelinelink>
         <pipelinelink name="proc4">
              <processor class="atg.commerce.addC"/>
         </pipelinelink>
         <pipelinelink name="proc5" transaction="TX_REQUIRES_NEW">
              <processor class="atg.commerce.addD"/>
         </pipelinelink>
         <pipelinelink name="proc6" transaction="TX_NOT_SUPPORTED">
              <processor class="atg.commerce.addF"/>
         </pipelinelink>
         <pipelinelink name="proc7" transaction="TX_SUPPORTS">
              <processor jndi="/dynamo/atg/commerce/addG"/>
         </pipelinelink>
    </pipelinechain>
    <pipelinechain name="RemoveFromCart" transaction="TX_REQUIRED"
          headlink="proc99" classname="atg.service.pipeline.PipelineMonoChain">
         <pipelinelink name="proc99">
              <processor class="atg.commerce.removeA"/>
         </pipelinelink>
    </pipelinechain>
</PipelineManager>
```

Each section of the `PipelineManager.xml` file is described below.

The first line says that this file compiles with the XML version 1.0 specification.

```
<?xml version="1.0"?>
```

The following line says that this is an XML file and that its DTD (document type definition) is in the file `PipelineManager.dtd`.

```
<!DOCTYPE PipelineManager SYSTEM "PipelineManager.dtd">
```

This following line is the start of the `PipelineManager` definition. A `<PipelineManager>` tag encloses all of the processor chain definitions.

```
<PipelineManager>
```

The following line begins the definition for a chain named `AddToCart`. The default transactional mode of the `PipelineLinks` is TX_REQUIRED. The head link is `proc1`, which specifies the name of a `PipelineProcessor` defined later in the file.

```
<pipelinechain name="AddToCart" transaction="TX_REQUIRED"
               headlink="proc1">
```

The next section is the definition of a `PipelineLink` with name `proc1` and `PipelineProcessor` class name `atg.commerce.addA`. The `PipelineManager` initialization routine will construct the object `atg.commerce.addA` and set the `PipelineLink`'s processor reference to this object. This `PipelineLink` has two transitions coming out of it, one with return value 1 which links to `proc2` and one with return value 2 which links to `proc3`. Both `proc2` and `proc3` are defined later in the file.

```
<pipelinelink name="proc1">
     <processor class="atg.commerce.addA"/>
     <transition returnvalue="1" link="proc2"/>
     <transition returnvalue="2" link="proc3"/>
</pipelinelink>
```

The next section defines two additional `PipelineLinks`. It is like the previous section, except that `proc2` has defined an overriding transactional mode, TX_REQUIRES_NEW.

```
<pipelinelink name="proc2" transaction="TX_REQUIRES_NEW">
        <processor class="atg.commerce.addB"/>
        <transition returnvalue="1" link="proc4"/>
        <transition returnvalue="2" link="proc5"/>
</pipelinelink>
<pipelinelink name="proc3">
        <processor class="atg.commerce.addE"/>
        <transition returnvalue="1" link="proc6"/>
        <transition returnvalue="2" link="proc7"/>
        <transition returnvalue="3" link="proc2"/>
```

```
</pipelinelink>
```

This section defines four more `PipelineLink` objects, some with overriding transactional modes. The interesting part of this section is the processor definition for `proc7`. Instead of using a Java class name definition, this processor is defined with a JNDI reference as `jndi="/dynamo/atg/commerce/addG"`. This JNDI reference will be resolved at initialization time and used as the processor for this link. The final line is the closing `</pipelinechain>` tag for the processor chain.

```
<pipelinelink name="proc4">
    <processor class="atg.commerce.addC"/>
</pipelinelink>
<pipelinelink name="proc5" transaction="TX_REQUIRES_NEW">
    <processor class="atg.commerce.addD"/>
</pipelinelink>
<pipelinelink name="proc6" transaction="TX_NOT_SUPPORTED">
    <processor class="atg.commerce.addF"/>
</pipelinelink>
<pipelinelink name="proc7" transaction="TX_SUPPORTS">
    <processor jndi="/dynamo/atg/commerce/addG"/>
</pipelinelink>
</pipelinechain>
```

The last section defines another `PipelineChain` called `RemoveFromCart` with default transaction mode TX_REQUIRED, `headlink` proc99. It specifies that the `classname` to be used for the construction of the `PipelineChain` is called `PipelineMonoChain`. The last line is the closing tag that closes off the `PipelineManager` definition.

```
<pipelinechain name="RemoveFromCart" transaction="TX_REQUIRED" headlink="proc99"
           classname="atg.service.pipeline.PipelineMonoChain">
    <pipelinelink name="proc99">
         <processor class="atg.commerce.removeA"/>
    </pipelinelink>
</pipelinechain>
</PipelineManager>
```

After the XML file is defined, start the Pipeline Manager component to construct the `PipelineManager` and its chains automatically.

Creating and Editing Processor Chains Programmatically

The following section describes various ways to construct or edit a processor chain using the API.

Locking and Synchronization for Editing Processor Chains

When you edit a processor chain, you must first obtain a lock on the chain, using the `lockChain()` method of the `PipelineManager`. When you lock a chain, the `PipelineManager` gets a reference to the calling thread and stores it within the requested chain. If the lock reference of the chain is null, then the lock is granted. If the lock reference is the same as the requesting thread, then the chain is already locked by the caller and execution falls through. For all other cases, an exception is thrown, which means that the chain is already locked by another thread.

Portions of the `lockChain()` and `unlockChain()` methods need to be synchronized, particularly those with sections that check the lock reference and set it.

The lock cannot be granted if there are threads executing in the requested chain. The chain keeps a reference count for all the threads executing within it. If this count is greater than 0, then the call to `lockChain()` blocks until the reference count is 0. When the reference count reaches 0, then the blocked call is notified and the `lockChain()` method continues. To prevent starvation of the `lockChain()` call, once that method is called, other threads attempting to call `runProcess()` will sleep until `unlockChain()` is called. Once this happens, all the threads sleeping on the `runProcess()` call will be notified.

Creating a New Processor Chain from Scratch

To create a new processor chain using the API:

1. Call the `PipelineManager`'s method `createChain()`. The `createChain()` method constructs the appropriate `PipelineChain` object and sets the proper return type into it. It implicitly locks and disables the chain and then adds it to the Pipeline Manager.

2. Edit the chain by making calls to the `createLink()`, `setHeadLink()`, and `addTransition()` methods of the `PipelineManager`.

   - `createLink()` is called to create all the `PipelineLink` objects that will be used in this chain and to link them together.

   - `setHeadLink()` is called to set the first link object into the chain.

   - `addTransition()` is called to create any additional transitions between links.

3. Call `enableChain()` to allow execution and `unlockChain()` to allow other threads to edit the chain. These last two calls must be made in this order. Otherwise, an exception will be thrown because `enableChain()` requires the caller to have a lock on the chain.

Editing an Existing Processor Chain

Follow these steps to edit an existing processor chain.

1. Call `lockChain()`. This call will block until there are no threads executing the chain and a lock can be obtained.

2. Call `setHeadLink()`, `createLink()`, `addTransition()`, and `removeTransition()` to edit the chain, as described in the preceding Creating a New Processor Chain from Scratch (page 378) section.

3. Call `unlockChain()` to release the lock and allow execution of the chain to resume.

Replacing an Existing Processor Chain

Follow these steps to remove a chain and replace it with a new version of the same chain.

1. Create a copy of the chain you want to replace by calling `duplicateChain()` and passing the `chainId`, which will be replaced as the argument. This call will return a duplicate disabled copy of that chain.

2. Edit the new chain, using the `setHeadLink()`, `createLink()`, `addTransition()`, and `removeTransition()` methods, as described in the Editing an Existing Processor Chain (page 378) section.

3. Call `lockChain()` on the chain that is to be replaced.

4. Call `replaceChain()` with the new chain as the argument. This will replace the chain that has the same name as the new chain.

5. Call `enableChain()` and `unlockChain()` on the new chain, in that order.

Creating a New Processor Chain from an Existing Chain

Follow these steps to create a chain based on an existing chain in the `PipelineManager`.

1. Call `copyChain(oldChainName, newChainName)`. This returns a copy of the chain with a new name. The new chain is locked and disabled.

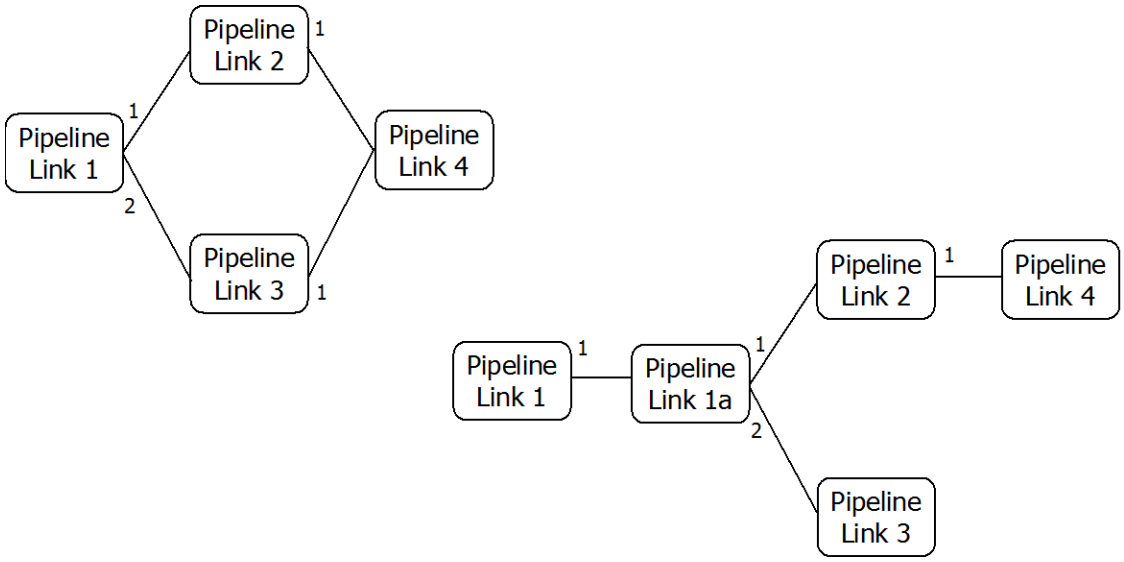
2. Edit the new chain if required, as described in the Editing an Existing Processor Chain (page 378) section.

3. Call `enableChain()` to allow execution on it and `unlockChain()` to allow other threads to edit the chain. These last two calls must be made in this order or an exception will be thrown, because `enableChain()` requires the caller to have a lock.

Adding and Removing Transitions to and from a Processor Chain

Adding and removing transitions to and from a processor chain requires the use of the `addTransition()` and `removeTransition()` methods.

The following before and after diagram demonstrates adding and removing transitions. Pipeline Link 1 is the head of the chain. Pipeline Link 1a is inserted after Pipeline Link 1 and before Pipeline Link 2 and Pipeline Link 3. The transition from Pipeline Link 3 to Pipeline Link 4 is removed.



1. Call `lockChain()` in the `PipelineManager` with the `chainId`. This insures that no thread executes in the chain and no other thread edits this chain.

2. Remove the two transitions from Pipeline Link 1 by doing the following:

   - Get references to Pipeline Link 2 and Pipeline Link 3.

   - Call `removeTransition()` twice to remove the two transitions that come from Pipeline Link 1.

3. Call `createLink()` in the `PipelineChain` to create Pipeline Link 1a. The arguments to `createLink()` will be as follows: Pipeline Link 1a, a `PipelineProcessor` object, a reference to the link whose ID is Pipeline Link 1, and 1. This creates the transition from Pipeline Link 1 to Pipeline Link 1a.

4. Call `addTransition()` twice to add the two transitions to Pipeline Link 2 and Pipeline Link 3.

5. Call `removeTransition()` with arguments Pipeline Link 3 and 1. This removes the transition from Pipeline Link 3 to Pipeline Link 4.

6. Call `unlockChain()` on the chain.

Extending the PipelineChain and PipelineResult Classes

The examples in the preceding section, Creating and Editing Processor Chains Programmatically (page 377), showed how to create processor chains using the default version of the `createChain()` method. This version of `createChain()` takes one argument and constructs and returns an object of type `atg.service.pipeline.PipelineChain` (the default). `atg.service.pipeline.PipelineResult` will be the object used for that chain's `PipelineResult` object.

If you subclass `PipelineChain` and `PipelineResult`, you can create a processor chain with your subclass in the same way you create a chain using the default `PipelineChain` and `PipelineResult` objects. The only difference is that you need to use another version of the `createChain()` method. This `createChain()` method does the same thing, except that it accepts two String arguments. These String arguments identify the classes to be used for the `PipelineChain` and `PipelineResult` objects.

Example

This example uses a specialized version of `PipelineChain` called `PipelineMonoChain`. This `PipelineChain` subclass represents a singly linked list, so each `PipelineLink` object transitions only to a single `PipelineLink`.

To create the `PipelineMonoChain` class, the first step is to subclass `PipelineChain`. The `PipelineMonoChain` class overrides the following methods in `PipelineChain`: `createLink()`, `addTransition()`, and the two `removeTransition()` methods.

The subclass's `createLink()` method returns `PipelineMonoLink`, which is a special type of `PipelineLink` object that only allows a single transition out of it. It extends `atg.service.pipeline.PipelineLink`. The declaration of `createLink()` looks like this:

```
public PipelineMonoLink createLink(String aLinkId, PipelineProcessor aProc,
       PipelineLink aFromLink, int aRetCode)
    throws CreateLinkException, TransitionException
```

This creates a `PipelineMonoLink` object instead of a `PipelineLink` object and returns it to the caller. The main difference is that if `aFromLink` already has a transition coming out of it, then a `TransitionException` would be thrown.

In `PipelineLink`, the `TransitionException` is thrown if the `aRetCode` was already mapped in `aFromLink`. The `aRetCode` value has a slightly different meaning in `PipelineMonoLink`. If its value is returned from the `PipelineProcessor` for that link, then the transition is followed. Otherwise execution on the chain ends and the `PipelineResult` is returned to the caller.

For `PipelineMonoChain` to instantiate a `PipelineMonoLink`, the `instantiatePipelineLink` method must be overridden to return a new instance of a `PipelineLink` subclass. The code would look like this:

```
protected PipelineLink instantiatePipelineLink(String pLinkId) {
    return new PipelineMonoLink(pLinkId);
```

```
}
```

The `addTransition()` method checks to see if there is a transition coming out of `aFromLink`. If there is, then a `TransitionException` is thrown. Otherwise, the link is mapped. Your subclass should include a new version of `addTransition()` with the following signature:

```
void addTransition(PipelineMonoLink aFromLink, int aRetCode, PipelineMonoLink
                     aToLink)
    throws TransitionException
```

This differs from the `addTransition()` in `PipelineLink` by the first and third arguments. The type in the method above is `PipelineMonoLink`, rather than `PipelineLink`, as in `PipelineChain`. The `addTransition()` method in `PipelineLink` however still exists and has not been overridden. This should be explicitly overridden and the code should throw a `TransitionException`. This method would only be called if a `PipelineLink` (or subclass) that is not a `PipelineMonoLink` was passed as one of the arguments.

The `removeTransition()` method checks to see if a transition is coming out of `aFromLink`. If none exists, then a `TransitionException` is thrown. If one does exist, then the transition would be removed. Again, new versions of `removeTransition()` should be added with the following signatures:

```
void removeTransition(PipelineMonoLink aFromLink, int aRetCode)
    throws TransitionException
void removeTransition(PipelineMonoLink aFromLink, PipelineMonoLink aToLink)
    throws TransitionException
```

These differ again by the arguments, which provide for `PipelineMonoLink` objects to be passed as parameters rather than `PipelineLink` objects. The `removeTransition()` methods that take `PipelineLink` objects should again be overridden explicitly and exceptions thrown.

The `PipelineMonoLink` object needs to extend `PipelineLink`. New methods that are specific to this implementation need to be defined and the following methods need to be overridden: `getNextLink()`, `getNextLinks()`, and `getRetCodes()`. The `PipelineMonoLink` would contain the following methods:

```
public PipelineMonoLink getNextLink()
public PipelineLink getNextLink(int aRetCode)
public PipelineMonoLink getNextLinks()
public PipelineLink[] getNextLinks(int[] aRetCodes)
public int getRetCode()
public int[] getRetCodes()
```

Although it takes a return code parameter, `getNextLink(int aRetCode)` should just return the `PipelineMonoLink` object that is mapped to the link that called the method. For cleanliness, a `getNextLink()` method should be defined that takes no arguments and returns a `PipelineMonoLink`. The inherited `getNextLink(int aRetCode)` method should just call the one with no arguments and return the `PipelineMonoLink` object that is cast to a `PipelineLink`. The `getNextLinks(int[] aRetCodes)` method should also just call `getNextLinks()` and ignore the `aRetCodes` and put the `PipelineMonoLink` into a `PipelineLink` array. The `int[] getRetCodes()` method should again just call `getRetCode()` and take the return value, insert it into an array and return it.

The example in this section implements a singly linked chain. It overrides `PipelineChain` and `PipelineLink` to force the objects to allow each link to have only one transition. The `PipelineMonoLink` enforces this. The

reason `PipelineChain` is overridden is to create `PipelineMonoLink` objects rather than `PipelineLink` objects.

Pipelines and Transactions

The Pipeline Manager is transactionally aware. This means that as it executes the processors in its processor chains, each processor has the ability to mark the chain execution to be rolled back. If none of the processors mark the transaction, then it is committed when the chain has finished executing.

The Pipeline Manager relies on the Java Transaction API (JTA) for its transactional needs. For more information about JTA transactions in the Oracle ATG Web Commerce platform, see the *Transaction Management* chapter in the *ATG Platform Programming Guide*. A processor may handle transactions in one of five modes:

```
TX_REQUIRED

TX_REQUIRES_NEW

TX_NOT_SUPPORTED

TX_SUPPORTS

TX_MANDATORY
```

Processor Transaction Management

Transaction management refers to the action of executing `PipelineProcessors` in the context of their defined transaction modes. This table describes each transaction mode and what the Pipeline Manager does for each of those modes.

`TX_REQUIRED`	The pipeline processor requires that a transaction be present for execution. If there is no transaction available when the pipeline manager is called, then the manager will create one and use it for execution of the pipeline. If the `PipelineManager` created the transaction then it will be committed after the processor completes, otherwise it will not be.
`TX_REQUIRES_NEW`	The pipeline processor requires its own transaction for execution. If a transaction is present, then it is suspended. A new transaction is created before the execution. The processor then executes its code. At completion, the transaction is either committed or rolled back. The original transaction is then resumed.
`TX_NOT_SUPPORTED`	The processor is not to be executed in a transaction. The current transaction is suspended, the processor executes, and then the transaction is resumed.
`TX_SUPPORTS`	The processor can be executed in a transaction. It will execute in the context of a transaction is one is available. If one is not available it will still execute, but without a transaction.
`TX_MANDATORY`	This means that a transaction must already be in place, otherwise an error occurs. The transaction is not automatically committed after the request.

Spanning Transactions over a Chain Subset

Transactions cannot span a subset of the chain. It must either span an entire chain or one element of a chain. If a transaction must only support a subset of a chain, you can code the logic for the subset of the chain to be in one processor and then set the transaction mode for that processor to TX_REQUIRES_NEW.

A more flexible way to span transactions over subsets of a chain is to break the subset into a new chain and then execute that chain within a processor of another chain. This new chain should have all its processors marked as either TX_REQUIRES or TX_MANDATORY. The processor of the calling chain should have its mode set to TX_REQUIRES_NEW.

Extending the Processor Pipeline Classes

You can extend the `atg.service.pipeline` API for customized behavior. An example of this was given in the Extending the PipelineChain and PipelineResult Classes (page 380) topic of the Creating a Processor Pipeline (page 371) section. The `atg.service.pipeline` API contains two interfaces that allow the objects in the system to be customized:

> `atg.service.pipeline.PipelineResult` provides access to the pipeline execution error data.
>
> `atg.service.pipeline.PipelineProcessor` is implemented by the processor components that the Pipeline Manager executes. Its main method is `runProcess()`.

The following table summarizes the classes in the `atg.service.pipeline` package:

`PipelineManager`	A global `GenericService` that controls the management of the pipelines and the execution of requests in the pipelines.
`PipelineChainConfiguration`	An object that contains data about a `PipelineChain` and a reference to the chain itself. This object is used internally in the `PipelineManager`.
`PipelineChain`	An object that contains a `PipelineLink` to the first `PipelineProcessor` for a given chain. It also manages the execution and editing of the `PipelineLinks`.
`PipelineLink`	An object that contains a reference to a `PipelineProcessor` and a mapping of return values to next processors. It is used by the `PipelineChain` to call the `runProcess()` method on the component and then get the reference to the next processor based on the return value.
`PipelineResult`	An object that implements the `PipelineResult` interface. This is the default `PipelineResult` object created by the `PipelineManager` when a chain is executed if no other `PipelineResult` object is specified.
`PipelineManagerException`	The base exception object extended by all exceptions thrown from methods in the `atg.service.pipeline` package.

Using Site-Based Forking in a Processor Chain

If you are using Oracle ATG Web Commerce as part of a multisite installation (see the *ATG Multisite Administration Guide* for general information on multisite features), you can include a processor that uses the current site ID to fork your processor chain. This allows you to specify different processing behaviors for different sites in your installation.

Site-based forking involves two components. The first is the `/atg/commerce/CommercePipelineManager` component. This component is based on the `atg.commerce.pipeline.CommercePipelineManager`, which extends the basic `PipelineManager` class. The `CommercePipelineManager` component adds the current site ID to the map of pipeline parameters (note, however, that if an ID is already in the parameter map, `CommercePipelineManager` does not override the existing value).

The `CommercePipelineManager` component includes the `useSiteFromOrder` property. If this property is set to `true` and there is no current site context, the `CommercePipelineManager` gets the site ID from the current `order` object instead of from the site context. This property is used for scheduled orders (see Scheduling Recurring Orders (page 318)), for which no site context is available.

The second component used in site-based forking is the `/atg/dynamo/service/pipeline/processor/SiteForkProcessor` component. It includes a `sitesMap` property, which you configure to map your sites to numeric values as shown:

```
sitesMap=\
  siteA=1;\
  siteB=2
```

The `SiteForkProcessor` retrieves the site ID from the pipeline parameters map and uses its own `sitesMap` to find the corresponding number. If there is no number for the site, the component finds the list of sites that occupy the same sharing group (if any), and iterates over that list; when it finds an entry that does appear in the sites map, it returns the number associated with that site. If no site is found, the `siteForkProcessor` returns a value of 0. You can use this returned value to select among options in your pipeline chain.

In addition to the `sitesMap`, the `SiteForkProcessor` component has a `shareableTypeId` property. By default this value is not set. It can contain the ID of a shareable type such as the shopping cart. To find the valid `shareableType` items, look at the `/atg/multisite/SiteGroupManager` component. An example of a `shareableType` is `atg.ShoppingCart`.

The `shareableTypeId` can be null, in which case there must be an exact match between the site ID in the pipeline parameters and the site ID in the map.

For example, to insert a fork that processes differently based on which of two sites the user is visiting, configure your `SiteForkProcessor` component's `sitesMap` property as shown:

```
sitesMap=\
  apparel=1;\
  home=2
```

Then create the following pipeline chain:

```
...
<pipelinelink name="SiteForkProcessor" transaction="TX_MANDATORY">
  <processor jndi="/myApp/pipeline/processor/SiteForkProcessor"/>
```

```
  <transition returnValue="1" link="apparelPipelineProcessor"/>
  <transition returnValue="2" link="homePipelineProcessor"/>
</pipelinelink>

<pipelinelink name="homePipelineProcessor" transaction="TX_MANDATORY">
  <processor jndi="/atg/commerce/HomeSiteProcessor"/>
  <transition returnValue="1" link="anotherProcessor"/>
</pipelinelink>

<pipelinelink name="apparelPipelineProcessor" transaction="TX_MANDATORY">
  <processor jndi="/atg/commerce/ApparelSiteProcessor"/>
  <transition returnValue="1" link="anotherProcessor"/>
</pipelinelink>
```

Adding a Commerce Processor Using XML Combination

There are two ways to extend a pipeline defined for a `PipelineManager`. One is to copy the entire XML file into your CONFIG layer and make the necessary changes. The other way is to use XML combination. Using XML combination is the preferred approach. The example below demonstrates of how to use XML combination to modify a pipeline chain.

The XML below is a section of the `processOrder` pipeline chain as it appears out of the box.

```
<pipelinechain name="processOrder" transaction="TX_REQUIRED"
           headlink="executeValidateForCheckoutChain">
  <pipelinelink name="executeValidateForCheckoutChain" transaction="TX_MANDATORY">
     <processor
            jndi="/atg/commerce/order/processor/ExecuteValidateForCheckoutChain"/>
     <transition returnvalue="1" link="checkForExpiredPromotions"/>
  </pipelinelink>
  <pipelinelink name="checkForExpiredPromotions" transaction="TX_MANDATORY">
     <processor jndi="/atg/commerce/order/processor/CheckForExpiredPromotions"/>
     <transition returnvalue="1" link="removeEmptyShippingGroups"/>
</pipelinelink>
```

The following example demonstrates how to add a new processor called `purgeExcessOrderData` between the `executeValidateForCheckoutChain` and `checkForExpiredPromotions` processors in the `processOrder` pipeline chain. The following XML code should be added to your `config` layer**.**

The important sections of this code are the additions of the `xml-combine` attributes in the `pipelinechain` and `pipelinelink` tags. The `pipelinechain` tag indicates what is being appended to its contents. The `pipelinelink` tag for `executeValidateForCheckoutChain` indicates what is replacing its contents.

```
<pipelinemanager>
 <pipelinechain name="processOrder" transaction="TX_REQUIRED"
            headlink="executeValidateForCheckoutChain" xml-combine="append">
   <pipelinelink name="executeValidateForCheckoutChain" transaction="TX_MANDATORY"
         xml-combine="replace">
     <processor
           jndi="/atg/commerce/order/processor/ExecuteValidateForCheckoutChain"/>
     <transition returnvalue="1" link="purgeExcessOrderData"/>
```

```
    </pipelinelink>
    <pipelinelink name="purgeExcessOrderData" transaction="TX_MANDATORY">
      <processor jndi="/atg/commerce/order/processor/PurgeExcessOrderData"/>
      <transition returnvalue="1" link="checkForExpiredPromotions"/>
    </pipelinelink>
  </pipelinechain>
</pipelinemanager>
```

The `purgeExcessOrderData` processor's transition is what the `executeValidateForCheckoutChain` transition is in the base file within the Oracle ATG Web Commerce platform.

Executing Processor Chains from Processors within Other Chains

A processor uses another processor called `ProcExecuteChain` to execute a subchain. The property file of the `ProcExecuteChain` is configured with a `PipelineManager` and a chain name to execute.

The key to the `ProcExecuteChain` execution is its return value. The value it returns is not that of the last processor in the chain that it executed, but rather whether or not the result object contains errors. If it did contain an error, `ProcExecuteChain` will return STOP_CHAIN_EXECUTION_AND_ROLLBACK, otherwise it will return SUCCESS (SUCCESS is mapped to the value 1 by default). These return values are configurable in the properties file.

18 Inventory Framework

The Inventory Framework facilitates inventory querying and inventory management for your sites. Interaction with an inventory system is vital to the various stages of an electronic purchase. Many of the Oracle ATG Web Commerce components interact with the Inventory System.

This chapter contains information on the following topics:

Overview of the Inventory System

The Inventory System provides a complete set of methods to support inventory handling. All users of the Inventory System need the same functionality to complete their varied tasks.

The Inventory System allows you to:

- Remove items from inventory.
- Notify the store of a customer's intent to purchase an item that is not currently in stock. (backorder)
- Notify the store of a customer's intent to purchase an item that has never been in stock. (preorder)

The administrator of the store uses the inventory system to:

- Place a specific number of items on a shelf for customers to purchase, backorder, or preorder.
- Decrease the number of items available for purchase, backorder, or preorder, perhaps because of an error in stocking the item.
- Determine the number of items available for purchase, backorder, or preorder.
- Determine when a specific item will be back in stock.

There are two types of methods: those that reflect the state of the store and those that change the state of the store. The Inventory API is not intended to be a complete inventory admin interface.

Oracle ATG Web Commerce uses the `InventoryManager` interface when performing any operation on inventory. By default, Commerce is configured with one full implementation of this interface, `RepositoryInventoryManager`. For more information, see the RepositoryInventoryManager (page 394) section.

`RepositoryInventoryManager` can be used as a complete inventory system. You can use a different inventory system by providing your own implementation of the `InventoryManager` interface. For more information on the `InventoryManager` interface, see the Inventory Classes (page 391) section.

Using the Inventory System

The Java code you write determines the order flow and customer experience by calling Inventory API methods in a particular order. Most sites use a 'shopping cart' system that allows a user to shop with the aid of an inventory manager in a way that's similar to a brick-and-mortar store's experience.

For example, after the customer checks out a shopping cart and enters the shipping and payment information, the rest of the order system (including fulfillment) uses the Inventory API.

The purchase call is made on each of the items in the shopping cart. If there is enough stock in the inventory to fulfill the order, the purchase call succeeds. If the purchase call does not succeed, the availability status of the item can be queried to determine if the item is preorderable, backorderable, or out of stock. Depending on the availability status, the item can be backordered or preordered. The fulfillment system makes this determination.

Note: Your sites can also be set up so that as the customer browses items, the inventory system can be queried for item availability.

If an item is purchased successfully, its `stocklevel` is decreased. If the item is backordered, the `backorderLevel` is decreased. If the item is preordered, the `preorderLevel` is decreased.

The store administrator uses `setStocklevel`, `increaseStocklevel`, and `decreaseStocklevel` to manage the inventory levels in the store. The store administrator uses `setBackorderLevel`, `increaseBackorderLevel`, and `decreaseBackorderLevel` to manage the backorderable amount. The store administrator uses `setPreorderLevel`, `increasePreorderLevel`, and `decreasePreorderLevel` for preorderable items. The administrator can get information on each of these levels with `queryStocklevel`, `queryBackorderLevel`, and `queryPreorderLevel`.

Each of these levels (`stocklevel`, `backorderLevel`, and `preorderLevel`) has a threshold associated with it. If the level falls below its associated threshold, an `InventoryThresholdReached` event is generated. There are methods in the API for the administrator to set each of these thresholds.

Two methods can be used to purchase items that were previously backordered or previously preordered. These are `purchaseOffBackorder` and `purchaseOffPreorder`. See the section on RepositoryInventoryManager (page 394) for further discussion of these methods.

Inventory System Methods

The Inventory System consists of implementations of an Inventory Manager interface. The interface consists of the following methods. Refer to the *ATG Platform API Reference* for more information on these methods.

Method	Action Performed
`purchase`	Decreases the inventory (`stocklevel`) of the item. Returns unsuccessfully if inventory is not available. This method is required by any implementation of the inventory system.
`purchaseOffBackorder`	Does the same thing as purchase and increases the `backorderLevel`.
`purchaseOffPreorder`	Does the same thing as purchase and increases the `preorderLevel`.
`preorder`	Decreases the `preorderLevel` of the item. Returns unsuccessfully if the `preorderLevel` is lower than the quantity being preordered.
`backorder`	Decreases the `backorderLevel` of the item. Returns unsuccessfully if the `backorderLevel` is lower than the quantity being backordered.
`setStocklevel`	Sets the `stocklevel` to a fixed number.
`setBackorderLevel`	Sets the `backorderLevel` to a fixed number.
`setPreorderLevel`	Sets the `preorderLevel` to a fixed number.
`increaseStocklevel`	Increases the `stocklevel` of an item by some quantity. This is used if an order is cancelled after the item has already been purchased or the administrator can call it.
`increaseBackorderLevel`	Increases the `backorderLevel` of an item by some quantity. This is used if an order is cancelled after the item has already been `backorded` or the administrator can call it.

Method	Action Performed
`increasePreorderLevel`	Increases the `preorderLevel` of an item. This is used if an order is cancelled after the item has already been preordered or the administrator can call it.
`decreaseStocklevel`	Decreases the `stocklevel` of an item. This should not be used in place of purchase. An administrator usually calls it.
`decreaseBackorderLevel`	Decreases the `backorderLevel` of an item. This should not be used in place of backorder. An administrator usually calls it.
`decreasePreorderLevel`	Decreases the `preorderLevel` of an item. This should not be used in place of preorder. An administrator usually calls it.
`setStockThreshold`	Sets the threshold associated with `stocklevel` (`stockThreshold`) to a fixed number.
`setBackorderThreshold`	Sets the threshold associated with `backorderLevel` (`backorderThreshold`) to a fixed number.
`setPreorderThreshold`	Sets the threshold associated with `preorderLevel` (`preorderThreshold`) to a fixed number.
`setAvailabilityStatus`	Sets the availability status.
`setAvailabilityDate`	Sets the date at which the item will become available.
`queryAvailabilityStatus`	Determines whether an item is in stock, out of stock, backorderable, or preorderable. Used when determining which method to call: purchase, backorder, or preorder.
`queryStocklevel`	Returns the number of items available for purchase (`stocklevel`).
`queryBackorderLevel`	Returns the number of items available for backorder (`backorderLevel`).
`queryPreorderLevel`	Returns the number of items available for preorder (`preorderLevel`).
`queryStockThreshold`	Returns the `stockThreshold`.
`queryBackorderThreshold`	Returns the `backorderThreshold`.
`queryPreorderThreshold`	Returns the `preorderThreshold`.
`queryAvailabilityDate`	Returns the date when this item will become available.
`inventoryWasUpdated`	This method is called when a set of items is added to inventory. It is a convenient way of notifying interested systems of a large update. Does not change the state of the inventory.

Inventory Classes

The inventory API includes the following classes:

- InventoryManager (page 391)
- InventoryException (page 393)

InventoryManager

The `InventoryManager` is a public interface that contains all of the Inventory system functionality. Each method described below returns an integer status code. All successful return codes should be greater than or equal to zero, all failure codes should be less than zero. By default, the codes are:

Status Code	Description
`int` INVENTORY_STATUS_SUCCEED=0	There was no problem performing the operation.
`int` INVENTORY_STATUS_FAIL=-1	There was an unknown/generic problem performing the operation.
`int` INVENTORY_STATUS_INSUFFICIENT_SUPPLY = -2	The operation couldn't be completed because there were not enough of the item in the inventory system.
`int` INVENTORY_STATUS_ITEM_NOT_FOUND = -3	The operation could not be completed because a specified item could not be found in the inventory system.

The methods that affect a change on the backend system return an integer status code. Methods that inspect the backend system return the answer to the query. The status return codes listed above and any implementer-defined status codes usually should be used to convey a definitive answer concerning a method's success.

When a method succeeds, an answer is prepared for return from the method. When a method fails, you can either return a failure code or throw an exception. In general, a method should return a failure code only if it is determined that the operation will not succeed. For example:

- If it is determined that five items cannot be purchased because there are only 4 items in the store, the implementer should return INVENTORY_STATUS_INSUFFICIENT_SUPPLY. This indicates that the program determined that the purchase method could not succeed.
- If a network error caused a connection to time out while the `InventoryManager` was querying the backend system, an exception should be thrown because the program was unable to determine whether the operation would succeed.

The following table lists all the methods provided by the `InventoryManager` interface. In methods below, the `pID` parameter would usually be a SKU ID.

Methods	Exception thrown
int purchase (String pId, long pHowMany)	InventoryException
int purchaseOffBackorder (String pId, long pHowMany)	InventoryException
int purchaseOffPreorder (String pId, long pHowMany)	InventoryException
int preorder (String pId, long pHowMany)	InventoryException
int backorder (String pId, long pHowMany)	InventoryException
int setStocklevel (String pId, long pNumber)	InventoryException
int setBackorderLevel (String pId, long pNumber)	InventoryException
int setPreorderLevel (String pId, long pNumber)	InventoryException
int increaseStocklevel (String pId, long pNumber)	InventoryException
int decreaseStocklevel (String pId, long pNumber)	InventoryException
int increaseBackorderLevel (String pId, long pNumber)	InventoryException
int decreaseBackorderLevel (String pId, long pNumber)	InventoryException
int increasePreorderLevel (String pId, long pNumber)	InventoryException
int decreasePreorderLevel (String pId, long pNumber)	InventoryException
int setStockThreshold (String pId, long pNumber)	InventoryException
int setBackorderThreshold (String pId, long pNumber)	InventoryException
int setPreorderThreshold (String pId, long pNumber)	InventoryException
int setAvailabilityStatus (String pId, int pStatus)	InventoryException
int setAvailabilityDate (String pId, Date pDate)	InventoryException
int queryAvailabilityStatus(String pId)	InventoryException, MissingInventoryItemException
long queryStocklevel(String pId)	InventoryException, MissingInventoryItemException
long queryBackorderLevel(String pId)	InventoryException, MissingInventoryItemException
long queryPreorderLevel(String pId)	InventoryException, MissingInventoryItemException
long queryStockThreshold(String pId)	InventoryException, MissingInventoryItemException

Methods	Exception thrown
`long queryBackorderThreshold(String pId)`	`InventoryException, MissingInventoryItemException`
`long queryPreorderThreshold(String pId)`	`InventoryException, MissingInventoryItemException`
`Date queryAvailabilityDate(String pId)`	`InventoryException, MissingInventoryItemException`
`int inventoryWasUpdated(List pItemIds)`	`InventoryException`

For more information, see the Examples of Using the Inventory Manager (page 399) section.

InventoryException

`InventoryException` is an exception designed for use with the Inventory Manager. `InventoryException` has the following methods:

```
Throwable getSourceException()
void setSourceException(Throwable t)
void printStackTrace()
void printStackTrace(PrintStream pStream)
void printStackTrace(PrintWriter pWriter)
String toString()
```

It has the following constructors:

```
InventoryException()
InventoryException(String pCause)
InventoryException(Throwable pRootException)
InventoryException(String pCause, Throwable pRootException)
```

MissingInventoryItemException

`MissingInventoryItemException` is a subclass of `InventoryException`. It is used if a method can't find the inventory item, but does not return a status code. If a method returns a status code and the item cannot be found, then INVENTORY_STATUS_ITEM_NOT_FOUND is returned.

InventoryManager Implementations

Oracle ATG Web Commerce includes the following implementations of the `InventoryManager` out of the box.

- AbstractInventoryManagerImpl (page 394)

AbstractInventoryManagerImpl

`AbstractInventoryManagerImpl` is an abstract class that removes all of the `InventoryManager` methods except `purchase`, which it defines as abstract. The `AbstractInventoryManagerImpl` provides a simple way for users to purchase items online, without all the extra features.

NoInventoryManager

`NoInventoryManager` is an implementation of the `InventoryManager` interface. It is intended to be a placeholder. It will be useful in cases where no `InventoryManager` functionality is required, but an inventory manager of some kind is needed for a property setting. It allows components that require an `InventoryManager` to function without actually having an inventory system on the back end. All methods in the `NoInventoryManager` return INVENTORY_STATUS_SUCCEED.

RepositoryInventoryManager

`RepositoryInventoryManager` implements all the methods of the `InventoryManager` interface. This `InventoryManager` broadcasts events when levels are at a configurable "critical" level and when it is notified of updated inventory.

`RepositoryInventoryManager` implements all the methods defined by the `InventoryManager` API. It is a thin wrapper around a repository that contains the inventory information. This allows a maximum amount of flexibility for potential third party integrators. Integrators can simply implement a repository containing the required properties for cooperation with the `RepositoryInventoryManager`. The Repository `InventoryManager` can then be configured to extract inventory manager information from the third party repository.

The initial implementation of the `RepositoryInventoryManager` uses the a SQL Repository to store inventory information. In the future, another repository can easily be swapped with the SQL Repository.

The `RepositoryInventoryManager` requires that the inventory items stored in the repository have certain attributes. All items must contain the following properties and types to represent information the `RepositoryInventoryManager` needs to store in the repository.

Note: The names of the properties are configurable in the `RepositoryInventoryManager`. This allows them to be internationalized, custom configured, etc.

- `java.lang.Long <stock level property>`

  Represents the number of items currently available for purchase. Every inventory item in the repository must have a Long property attached to it that represents the item's inventory level. The default value is `stockLevel`. The name of this property in the repository is configurable through the `stocklevelPropertyName` property in the `RepositoryInventoryManager`.

- `java.lang.Long <backorder level property>`

Represents the number of items that can be backordered currently. Every inventory item in the repository must have a Long property attached to it that represents the item's backorder level. The default value is `backorderLevel`. The name of this property in the repository is configurable through the `backorderLevelPropertyName` property in the `RepositoryInventoryManager`.

- `java.lang.Long <preorder level property>`

  Represents the number of items that can be preordered currently. Every inventory item in the repository must have a Long property attached to it that represents the item's preorder level. The default value is `preorderLevel`. The name of this property in the repository is configurable through the `preorderLevelPropertyName` property in the `RepositoryInventoryManager`.

- `java.lang.Long <stock threshold property>`

  Every inventory item in the repository must have a Long property attached to it that represents the item's stock level threshold. If the `stocklevel` falls below this value, an event is triggered. The default value is `stockThreshold`. The name of this property in the repository is configurable through the `stockThresholdPropertyName` property in the `RepositoryInventoryManager`.

- `java.lang.Long <backorder threshold property>`

  Every inventory item in the repository must have a Long property attached to it that represents the item's backorder level threshold. If the `backorderLevel` falls below this value, an event is triggered. The default value is `backorderThreshold`. The name of this property in the repository is configurable through the `backorderThresholdPropertyName` property in the `RepositoryInventoryManager`.

- `java.lang.Long <preorder threshold property>`

  Every inventory item in the repository must have a Long property attached to it that represents the item's preorder level threshold. If the `preorderLevel` falls below this value, an event is triggered. The default value is `preorderThreshold`. The name of this property in the repository is configurable through the `preorderThresholdPropertyName` property in the `RepositoryInventoryManager`.

- `java.lang.Integer < availability status property>`

  Every inventory item in the repository must have an Integer property attached to it that represents the item's availability status. The name of this property in the repository is configurable through the `availabilityStatusPropertyName` property. The default value of the property name is `availabilityStatus`.

  The possible values are:

  - AVAILABILITY_STATUS_IN_STOCK = 1000
  - AVAILABILITY_STATUS_OUT_OF_STOCK = 1001
  - AVAILABILITY_STATUS_PREORDERABLE = 1002
  - AVAILABILITY_STATUS_BACKORDERABLE = 1003
  - AVAILABILITY_STATUS_DERIVED = 1004
  - AVAILABILITY_STATUS_DISCONTINUED = 1005

  Note: If the status in the repository is AVAILABILITY_STATUS_DERIVED, then a call to `queryAvailabilityStatus` calculates the actual status based on the values of the item's `stocklevel`, `backorderLevel`, and `preorderLevel`.

If the status is "hardcoded" to something other than DERIVED, then the status might not reflect the actual state of the item. For example, if an item's availability status is AVAILABILITY_STATUS_IN_STOCK and the `stockLevel` is reduced to 0, then any call to `queryAvailabilityStatus` will return AVAILABILITY_STATUS_IN_STOCK even though there is no stock available. In most cases this would be an error, therefore AVAILABILITY_STATUS_DERIVED should be used for almost all inventory items.

- `java.util.Date <inventory availability date property>`

  The date and time at which more of the item will be available for purchase. If availability is AVAILABILITY_STATUS_IN_STOCK then this property is not used. If availability is AVAILABILITY_STATUS_OUT_OF_STOCK, AVAILABILITY_STATUS_PREORDERABLE, or AVAILABILITY_STATUS_BACKORDERABLE then this property is the date on which more of the product will be available for purchase. The default value is `availabilityDate`.

- `java.lang.String <catalog reference id property>`

  The ID of the item in the product catalog to which this inventory item refers. All the calls in the `InventoryManager` that take a SKU ID use this property to find the correct inventory item. The default value is `catalogRefId` and is configurable through the `catalogRefIdPropertyName` property.

The `RepositoryInventoryManager` implements the `InventoryManager` interface by using the configured properties listed above to extract data from a configured repository. For example, the `queryStocklevel` method is implemented by getting the item with the requested ID from the repository and reading the <stock level property> property.

Using the RepositoryInventoryManager to Implement the InventoryManager

The following section describes how the `RepositoryInventoryManager` implements the `InventoryManager` interface.

Every item in the inventory has an associated SKU (Stock Keeping Unit). Each SKU has three levels associated with it: `stocklevel`, `backorderLevel`, and `preorderLevel`. The behavior of each of these levels is similar. If someone makes a successful purchase call, `stocklevel` is decreased. If someone makes a successful backorder call, `backorderLevel` is decreased. If someone makes a successful preorder call, `preorderLevel` is decreased.

Every SKU also has an `availabilityStatus`. In most cases, the SKU will have an `availabilityStatus` of AVAILABILITY_STATUS_DERIVED. In some cases, it is strictly defined as AVAILABILITY_STATUS_IN_STOCK, AVAILABILITY_STATUS_OUT_OF_STOCK, AVAILABILIITY_STATUS_BACKORDERABLE, AVAILABILITY_STATUS_PREORDERABLE, or AVAILABILITY_STATUS_DISCONTINUED.

If it is derived, `queryAvailabilityStatus` calculates the value based on the three levels: `stocklevel`, `backorderLevel`, and `preorderLevel`.

- If `stocklevel` is not 0, then the SKU is IN_STOCK.
- If `stocklevel` is 0 but `backorderLevel` is not 0, then the SKU is BACKORDERABLE.
- If `stocklevel` and `backorderLevel` are both 0, but `preorderLevel` is not 0, then the SKU is PREORDERABLE.
- If all three levels are 0, then the SKU is OUT_OF_STOCK.

If a purchase call fails for a particular SKU and `queryAvailabilityStatus` says the item is backorderable, then backorder should be called. Calling backorder decreases the `backorderLevel`. To ensure the level remains consistent after the SKU is available again, `purchaseOffBackorder` should be called in place of purchase. This not only decreases `stocklevel`, but it also increases the `backorderLevel`.

If a purchase call fails for a particular SKU and queryAvailabilityStatus says the item is preorderable, then preorder should be called. Calling preorder decreases the preorderLevel. To ensure the level remains consistent after the SKU is available again, purchaseOffPreorder should be called in place of purchase. This not only decreases stocklevel, but it also increases the preorderLevel.

If your system does not need backorder levels and preorder levels, then you do not need to call backorder, preorder, purchaseOffBackorder, or purchaseOffPreorder. The purchase call is enough.

The default value for an item's stockLevel is –1. This value indicates that there is an infinite amount of stock. The default value for all other levels. (backorderLevel, preorderLevel, stockThreshold, backorderThreshold, and preorderThreshold) is 0.

If the fulfillment system attempts to purchase an item for a customer and the item is out of stock but BACKORDERABLE, then the fulfillment system can backorder the item. If the fulfillment system attempts to purchase an item for a customer and item is out of stock but PREORDERABLE, then the fulfillment system can preorder the item. Both these statuses mean that the whole order could be waiting for the item to be in stock. Therefore, it is important that the fulfillment system is notified when an item is in stock after being backordered, preordered, or even out of stock.

The UpdateInventory message indicates that new inventory is available for previously unavailable items. When the fulfillment system receives an UpdateInventory message, the fulfillment system knows that the items included in the message can be successfully purchased now. It is the responsibility of InventoryManager to send this message. The RepositoryInventoryManager sends the message when the inventoryWasUpdated method is called.

If a call is made to inventoryWasUpdated then an UpdateInventory message is constructed and sent out over the port specified in the updateInventoryPort property.

The UpdateInventory message has one property:

- String[] itemIds – A list of SKUs that were BACKORDERABLE, PREORDERABLE, or OUT_OF_STOCK, but are now IN_STOCK. This list is the same as the list of IDs passed into the inventoryWasUpdated method.

Refer to the Inventory JMS Messages (page 406) chapter for more information on the JMS Messages

CachingInventoryManager

The CachingInventoryManager is also included in the Oracle ATG Web Commerce out-of-the-box implementation. The CachingInventoryManager caches any read-only data for quick display to the site user. It is configured with a Cache and an UncachedInventoryManager.

The uncachedInventoryManager property of the CachingInventoryManager refers to any implementation of the InventoryManager API. It is recommended that the uncachedInventoryManager property refer to an instance of the RepositoryInventoryManager.

All methods are passed to the UncachedInventoryManager, except for

- queryStocklevel
- queryBackorderLevel
- queryPreorderLevel
- queryAvailabilityStatus

- queryAvailabilityDate
- queryStockThreshold
- queryBackorderThreshold
- queryPreorderThreshold.

These methods work by asking the cache for the item with the requested ID. The needed property value is then read from the cached item.

If a method is called that changes a property of an inventory item, then that item's cached value is invalidated and will be reloaded the next time it is needed. If the CachingInventoryManager is used, it should be used for all inventory operations or it could return invalid results. Invalid results could occur if some updates are made directly to the InventoryManager referred to by the uncachedInventoryManager property (as opposed to through the CachingInventoryManager). This is because the changes made outside of the CachingInventoryManager will not have invalidated the cache.

The flushCache method flushes the existing cache data. The flushCache method flushes the entire cache. flushCache(List) flushes the entry for each ID in list.

The CachingInventoryManager does not actually perform inventory management. It relies on another implementation of the InventoryManager interface. Therefore, if you provide your own implementation of the InventoryManager interface, instant caching can be configured using the CachingInventoryManager and setting its uncachedInventoryManager property to your implementation.

InventoryCache

InventoryCache is the cache used by the CachingInventoryManager. It can be found at /atg/commerce/inventory/InventoryCache in the component browser of the ACC. It is an instance of atg.service.cache.Cache. If the inventory data changes, resulting in stale data in the cache, it is possible to flush the cache. This class includes a public void method flush that can be accessed through the component browser. It flushes all the entries from the caching. For more information, see the Caching the Inventory (page 407) section.

The following table describes the properties of the InventoryCache component that you can use to define your cache policies:

maximumCacheEntries	The maximum number of elements in the cache. If set to 0, nothing will be cached. If set to –1, there is no limit on how many elements can be in the cache.
maximumCacheSize	The maximum memory size of the cache.
maximumEntrySize	The maximum memory size of a single entry in the cache.
maximumEntryLifetime	The maximum number of milliseconds that an entry will live in the cache. *

* By default, the maximumEntryLifetime property is set to 7,200,000 (2 hours) in the live configuration. The Motorprise reference application overrides this value and changes it to 60,000 (1 minute). This frequent update time is recommended during development, so that information will be updated rapidly.

LocalizingInventoryManager

LocalizingInventoryManager is an implementation of the InventoryManager interface that is used when you want to have more than one set of inventory data (and therefore more than one InventoryManager). LocalizingInventoryManager determines which data/manager set to use based on your customer's locale.

LocalizingInventoryManager contains the following properties:

- defaultLocaleKey – the locale to use to determine which InventoryManager to use when the locale cannot be determined from the request.
- defaultInventoryManager – the InventoryManager to use when the InventoryManager cannot be determined from the locale.
- inventoryManagers – a Map pairing localeKeys to their corresponding InventoryManager components.
- useDefaultInventoryManager – a Boolean property that determines whether to use the defaultInventoryManager if an InventoryManager cannot be found for the user's locale.

LocalizingInventoryManager implements all the InventoryManager methods twice:

- once with the signature provided in the InventoryManager interface
- once with an extra pLocaleKey String parameter

The methods with the InventoryManager signatures call the second implementations, passing the value of the defaultLocaleKey property as the extra parameter.

The second methods retrieve the proper InventoryManager by calling the getInventoryManager method with the localeKey as the parameter. Then, they call the corresponding method in the InventoryManager that was retrieved.

LocalizingInventoryManager is compatible with the InventoryManager interface because you can pass it the same parameters and it will use the default InventoryManager to perform operations. You also can make your calls with the extra pLocaleKey parameter, and use LocalizingInventoryManager to handle multiple InventoryManager instances.

The getInventoryManager method takes the localeKey parameter and checks the inventoryManagers map for a corresponding InventoryManager. If it finds one, it is returned. Otherwise, it checks the useDefaultInventoryManager property. If true, it returns the DefaultInventoryManager. If false, an error is thrown.

Examples of Using the Inventory Manager

This section provides a few simple examples of how the InventoryManager can be used during fulfillment. These examples include:

- Allocating Items for an Order (page 400)
- Canceling or Removing an Item from an Order (page 401)
- Displaying an Item's Availability to a Customer (page 401)

- Filling Partial Orders (page 402)
- Preventing Inventory Deadlocks (page 402)

Allocating Items for an Order

For the first example, if a customer orders five of item 'sku-0' then the inventory system can be used to allocate, or "purchase" that item:

```
String itemId = "sku-0";
long quantity = 5;
// Assume inventory manager is defined as a property in this class.
InventoryManager inventory = getInventoryManager();
int status = inventory.purchase(itemId, quantity);
```

The purchase call `status` at this time is either `INVENTORY_STATUS_SUCCEED`, or `INVENTORY_STATUS_FAIL`. If it is `INVENTORY_STATUS_SUCCEED`, then everything is ready and processing can continue. If the status is `INVENTORY_STATUS_FAIL`, then check the availability status of the item to determine why it failed:

```
int availability = inventory.queryAvailabilityStatus(itemId);
```

The next step depends on the availability status. The following sample sets the `newStatus` value based on the retrieved availability status.

```
 int newStatus;

 if(availability == inventory.AVAILABILITY_STATUS_BACKORDERABLE)
newStatus = inventory.backorder(itemId, quantity);

 else if(availability == inventory.AVAILABILITY_STATUS_IN_STOCK)
newStatus = inventory.backorder(itemId, quantity);

 else if(availability == inventory.AVAILABILITY_STATUS_PREORDERABLE_
newStatus = inventory.preorder(itemId, quantity);

 else  // the only other option is AVAILABILITY_STATUS_OUT_OF_STOCK
newStatus = inventory.INVENTORY_STATUS_FAIL;
```

If the availability status is IN_STOCK, then backorder is called. Backorder is called because the only way that the purchase call could fail if the item is in stock is if the requested quantity is higher than the current `stocklevel`. If the item is backordered, it can be reallocated later, after the stock level increases.

The `newStatus` value INVENTORY_STATUS_FAIL indicates one of the following situations:

- The item is out of stock.
- The request is attempting to allocate more items than available.
- The item is backorderable but the system could not successfully backorder the item. This could occur if the quantity in the order was higher than the number of items available to be backordered.
- The item is discontinued.

If the `newStatus` value is INVENTORY_STATUS_FAIL during fulfillment, then the system sets the state of the item to `FAILED`.

The above example does not include tracking features. Tracking is an important part of the inventory system. For example, you can track backordering or preordering the item and then change that item's state. If an item is backordered and then the `stockLevel` is later increased, simply calling `purchase` is insufficient to remove the item from backorder.

The following changes must be made to the first code sample. This ensures that the `backorderLevel` and `preorderLevel` for the given items are current. For more information on states, see Oracle ATG Web Commerce States (page 255).

```
String itemId = "sku-0";
long quantity = 5;
// assume we have some way of getting the state of the given item
// in DCS during fulfillment this is done using
// ShippingGroupCommerceItemRelationship.getState();
int itemState = getItemState();
InventoryManager inventory = getInventoryManager();
// now, use the appropriate method, depending on the state
int status;
if(itemState == INITIAL) // normal case
status = inventory.purchase(itemId, quantity);
else if(itemState == BACK_ORDERED)
status = inventory.purchaseOffBackorder(itemId, quantity);
else if(itemState == PRE_ORDERED)
status = inventory.purchaseOffPreorder(itemId, quantity)
```

Canceling or Removing an Item from an Order

The following example describes how to use the inventory manager to cancel or remove an item from the order.

```
String id = "sku-0";
long quantity = 5;
int itemState = getItemState();
InventoryManager inventory = getInventoryManager();
// now, use the appropriate method, depending on the state
int status;
if(itemState == PENDING_DELIVERY) // normal case
status = inventory.increaseStockLevel(itemId, quantity);
else if(itemState == BACK_ORDERED)
status = inventory.increaseBackorderLevel(itemId,quantity);
else if(itemState == PRE_ORDERED)
status = inventory.increasePreorderLevel(itemId, quantity);
```

Displaying an Item's Availability to a Customer

The inventory manager is also used when building site pages. Display an item's availability status to a customer browsing the site with the following code:

```
int availability = inventory.queryAvailabilityStatus(itemId);
```

Filling Partial Orders

You can configure the fulfillment system to fill a partial order and backorder the rest if there is not enough inventory to fulfill an entire order. For example, if a customer orders five towels, but there are only three towels available, you can configure the Fulfillment system to "purchase" as many items as possible and backorder any additional items. Follow these steps to make the configuration changes:

1. Determine how many items are available using `InventoryManager.queryStockLevel()`.

2. Purchase that amount remaining using `InventoryManager.purchase()`.

   Note: It is possible that another customer could purchase these items in the time between when you called `queryStockLevel` and called purchase. If this is the case, you can either loop until the purchase is successful or there is no inventory left or you can extend the `InventoryManager` to purchase all items available.

3. Create a new `ShippingGroupCommerceItemRelationship` in the same shipping group as the item currently being processed and for the same `CommerceItem`. Set the old relationship's quantity to whatever was successfully purchased. Set the new relationship's quantity to the remaining quantity. The state of the old relationship is PENDING_DELIVERY while the state of the new relationship will depend on `InventoryManager.queryAvailabilityStatus()`.

Preventing Inventory Deadlocks

`InventoryManager` includes the `acquireInventoryLocks` and `releaseInventoryLocks` methods. These methods can be used to prevent deadlocks in the database, especially if there are multiple Oracle ATG Web Commerce instances concurrently updating inventory.

`acquireInventoryLocks` acquires locks for the inventory items that apply to the given IDs. `releaseInventoryLocks` releases locks for the inventory items that apply to the given IDs.

`RepositoryInventoryManager` implements `acquireInventoryLocks` by calling `RepositoryInventoryManager.lock` for each id (plus each id within a bundle). It does not implement `releaseInventoryLocks` since those locks will be released automatically with the end of the transaction.

The following example demonstrates how to use these methods to prevent deadlocks:

```
void myPurchase(List pCatalogRefIds, long pQuantity)
{
   InventoryManager im = getInventoryManager();
   try {
      im.acquireInventoryLocks(pCatalogRefIds);
      Iterator idIterator = pCatalogRefIds.iterator();
      int success;
      while(idIterator.hasNext()) {
         String id = (String) idIterator.next();
         success = im.purchase(id, pQuantity);
         . . .
      }
   }
   finally {
      im.releaseInventoryLocks(pCatalogRefIds);
   }
}
```

Handling Bundled SKUs in the Inventory

When a SKU is bundled into a collection of SKUs, the `RepositoryInventoryManager` handles the bundle differently than an individual SKU.

The default Oracle ATG Web Commerce implementation defines bundles as lists of SKUs in the Product Catalog. For more information, see the *Using and Extending the Product Catalog* (page 23) chapter. SKUs that represent a bundle have a `bundleLinks` property that is a list of each `skuLink` in the bundle. Each `skuLink` contains a quantity and a SKU (from the Product Catalog).

SKUs that do not represent a bundle correspond to one item in the inventory repository and have an empty `bundleLinks` list. In this way, new bundles can be defined in the product catalog, without referring to or updating the inventory repository. Alternatively, bundles can be defined within the inventory as a separate inventory item by extending the `RepositoryInventoryManager`. Information on extending the `RepositoryInventoryManager` is described in this section.

The `RepositoryInventoryManager` calculates the `stocklevel` of a bundle based on the `stockLevel` of each item in bundle. These values are kept up to date in real time in each query method. The `stockLevel` is the largest number of bundles that could be purchased given the `stocklevel` of the SKUs in the bundle. For example:

The Bundle SKU D contains

- 1 of SKU A
- 2 of SKU B
- 10 of SKU C

The `stocklevel` values for the individual SKUs are:

- SKU A `stocklevel` = 20
- SKU B `stocklevel` = 20
- SKU C `stocklevel` = 20

Therefore, SKU D's (the bundle's) `stocklevel=2` because that is how many bundles could successfully be purchased given the current inventory.

A bundle's `backorderLevel` and `preorderLevel` are calculated in a similar way.

`RepositoryInventoryManager` sets the threshold of bundles as 0 to prevent events from being triggered when purchasing a bundle. If one of the bundle's SKUs falls below its threshold, an event is triggered. This is true for the `stockThreshold`, `backorderThreshold`, and `preorderThreshold`.

The `availabilityDate` of a bundle is the latest `availabilityDate` of the SKUs in the bundle. The `availabilityStatus` of a bundle is calculated as follows:

- OUT_OF_STOCK – Indicates that at least one of the bundled items is OUT_OF_STOCK or that the `stockLevel`, `backorderLevel`, and `preorderLevel` are all below the quantity needed for one bundle.
- PREORDERABLE – Indicates that none of the items is OUT_OF_STOCK, but at least one of the bundled items is PREORDERABLE.
- BACKORDERABLE – Indicates that none of the items are OUT_OF_STOCK or PREORDERABLE, but at least one of the items is BACKORDERABLE.

- IN_STOCK – Indicates that all of the bundled items are IN_STOCK with a `stockLevel` greater than the quantity of the item included in one bundle.

When a bundle is purchased, the call is successful if all of the bundled SKUs are IN_STOCK. If it is successful, the `stocklevel` of each SKU is decreased by the number of bundles purchased multiplied by the number of SKUs contained in the bundle. In the example above if someone successfully purchased the Bundle, the `stocklevel` of SKU A would be decreased to 19, the `stocklevel` of SKU B to 18, and the `stocklevel` of SKU C to 10.

Bundle processing makes it possible for a SKUs `backorderLevel` to decrease even if there are enough items in stock. Consider the example above replacing SKU A's `stockLevel` with 0. The bundle is not IN_STOCK now. Assume that all the items have a `backorderLevel` of 100. The fulfillment framework will try to backorder the bundle. The `RepositoryInventoryManager` will set SKU A's `backorderLevel` to 99, SKU B's `backorderLevel` to 98, and SKU C's `backorderLevel` to 90 even though there is stock available for SKU B and SKU C. When the fulfillment framework later calls `purchaseOffBackorder`, each `backorderLevel` will be increased back to 100 (assuming no one else backordered the items in the meantime) and each `stockLevel` will be decreased as described above.

When the `HardgoodFulfiller` receives an `UpdateInventory` message, it looks for orders that contain items with the `catalogRefId` specified in the `UpdateInventory` message. The items also must have a `ShippingGroupCommerceItemRelationship` in BACK_ORDERED, PRE_ORDERED, or OUT_OF_STOCK state. Therefore, if an item is a bundle, the `catalogRefId` of the bundle needs to be included in the message for orders waiting on that bundle to be updated. It is not sufficient to include only the component of the bundle that was updated.

Building a Store without Bundles

`RepositoryInventoryManager` can be used as the inventory system for a store even if it does not have bundles. If your system does not include bundles, then none of the items will be treated as bundled. If you don't want SKUs with a non-empty `bundleLinks` property to be treated as bundles, extend the `RepositoryInventoryManager` and override the `isBundle()` method to always return false. This will force the inventory system to treat all items the same way. Perform the same extension to the `RepositoryInventoryManager` if you want bundles to be processed in the same way as regular SKUs.

Changing the Bundle Handling

If you want to handle bundles in a different way than described above, but you want process SKUs that are not bundles in the same way, extend the `RepositoryInventoryManager` and override the bundle-specific methods.

The methods listed below are called with the methods of the `InventoryManager` API if the ID passed in is a bundle. For example, if `purchase(someId)` is called and `someId` refers to a bundle, purchase will call `purchaseBundle(someId)`. These methods are implemented in `RepositoryInventoryManager`, but not in any of the other classes.

- `isBundle`
- `purchaseBundle`
- `purchaseBundleOffBackorder`
- `purchaseBundleOffPreorder`
- `preorderBundle`
- `backorderBundle`
- `queryBundleStocklevel`
- `queryBundleBackorderLevel`

- queryBundlePreorderLevel
- queryBundleStockThreshold
- queryBundleBackorderThreshold
- queryBundlePreorderThreshold
- queryBundleAvailabilityDate
- deriveBundleAvailabilityStatus

Inventory Repository

Oracle ATG Web Commerce comes with two repositories that are relevant to the inventory system: the Product Catalog and the Inventory Repository. The Product Catalog stores prices, descriptions, images, and fulfiller information. For more information, see the *Using and Extending the Product Catalog* (page 23) chapter. The Inventory Repository stores the `stockLevel`, availability information, and other inventory data.

This section describes the inventory repository that is included with Commerce out of the box. The inventory information is stored in a separate repository from the product catalog. Each inventory method takes an ID from the product catalog and uses that information to get the inventory ID.

The inventory definition is stored in `atg/commerce/inventory/inventory.xml`. Each item in the inventory has the following properties.

Property	Definition
`creationDate`	The date this inventory item was created.
`startDate`	The inventory item will not be available until this date.
`endDate`	The inventory item will not be available after this date.
`displayName`	The name that is displayed to the user to represent this inventory item.
`description`	A description of this inventory item.
`catalogRefId`	The SKU ID in the Product Catalog to which this inventory item refers.
`availabilityStatus`	The status of this inventory item. It is an enumerated type. The integer code for each possible value is shown. The following codes represent availability status: INSTOCK (1000) - This item is in stock. OUTOFSTOCK (1001) – This item is out of stock. PREORDERABLE (1002)- This item may be preordered. BACKORDERABLE (1003) - This item may be backordered. DERIVED (1004) - This item's `availabilityStatus` should be derived from the values of its `stockLevel`, `backorderLevel`, and `preorderLevel`. DERIVED is the default value for `availabilityStatus`. DISCONTINUED (1005) – This item is discontinued.

Property	Definition
`availabilityDate`	The date on which this item will be available if not currently available.
`stockLevel`	The amount of stock available for purchase. The value -1 indicates that an infinite amount is available.
`backorderLevel`	The amount of this item that can be backordered. The value –1 indicates that the inventory system accepts an infinite number of backorders for this item.
`preorderLevel`	The amount of this item that can be preordered. The value –1 indicates the inventory system accepts an infinite number of preorders for this item.
`stockThreshold`	If the `stockLevel` falls below this amount, a warning event is generated.
`backorderThreshold`	If the `backorderLevel` falls below this amount, a warning event is generated.
`preorderThreshold`	If the `preorderLevel` falls below this amount, a warning event is generated.

Inventory JMS Messages

The `InventoryManager` creates Java Messaging Service (JMS) messages for the following events:

JMS Message Name	When message occurs	Information included in message
`InventoryThresholdReached`	When `stocklevel`, `backorderLevel`, or `preorderLevel` falls below its associated threshold.	The ID number of the item. Name of property (`stocklevel`, `backorderLevel`, or `preorderLevel`) Value of property Value of threshold
`UpdateInventory`	When the stock level increases. (In the `RepositoryInventoryManager`, it occurs when someone calls `inventoryWasUpdated`)	The IDs of all items that have stock available and were previously BACKORDERABLE, PREORDERABLE, or OUT_OF_STOCK.

Configuring the SQL Repository

The `InventoryManager` implementations that Oracle ATG Web Commerce provides out of the box require the `atg.service.cache` package and an instance of the Generic SQL Adapter. For more information on SQL Repositories, see the *ATG Repository Guide*.

The SQL Repository must be configured before the `RepositoryInventoryManager` can use it. You must set values for the following properties:

- `stocklevel` (optional)
- `backorderLevel` (optional)
- `preorderLevel` (optional)
- `stockThreshold` (optional)
- `backorderThreshold` (optional)
- `preorderThreshold` (optional)
- `availabilityStatus` (required)
- `availabilityDate` (optional)

If the properties above are not included, all items will be treated as having a `stockLevel` of –1 (infinite supply). The `backorderLevel`, `preorderLevel`, `stockThreshold`, `preorderThreshold`, and `backorderThreshold` are set to zero. These default values are configurable in the `RepositoryInventoryManager`.

The best way to insure that every item in the repository that represents an inventory item has the required properties is to have a view and/or `ItemDescriptor` in the repository dedicated to the "inventory item" type. However, this is not specifically required.

Caching the Inventory

By default, caching is turned off for the inventory repository. This is to insure that inventory data is always up to date across server instances. The `CachingInventoryManager` is provided as an effective inventory caching mechanism for displaying inventory information.

The `CachingInventoryManager` can be used to display information to customers as they browse the product catalog because, in most situations, inventory information displayed to customers during catalog browsing does not need to be updated in real time. Displaying inventory information using the `CachingInventoryManager` improves the performance of the site.

The `CachingInventoryManager` should not be used when real time inventory data is needed. Real time inventory information is usually needed during the purchase process and fulfillment process. In those cases, the (uncached) `InventoryManager` should be used during these processes. For more information on the `CachingInventoryManager`, see the InventoryManager Implementations (page 393) section.

The `InventoryDroplet` provides cached data to the user when appropriate and accesses real time inventory data from the repository when appropriate. The `useCache` property allows you to indicate when to use cached inventory data:

- If the `useCache` property is set to false, the inventory data in the repository is provided.
- If the `useCache` property is set to true, the cached data is provided.

The GSA can provide more complex types of caching. If appropriate, you can configure two instances of the `InventoryRepository` GSA, one with no caching and one with distributed caching (set cache-mode=distributed). You can configure one instance of the `RepositoryInventoryManager` to use the uncached GSA, and another instance of the `RepositoryInventoryManager` to use the cached GSA. For more information on distributed caching, see the *SQL Repository Caching* chapter in the *ATG Repository Guide*.

Inventory Repository Administration

You can use the repository editor to administer the inventory system. Because the `RepositoryInventoryManager` is implemented as a front-end to a repository, you can use the repository editor to edit the `RepositoryInventoryManager` backend. The disadvantage to using the repository editor is that the `InventoryManager` does not know when the Repository has changed. Therefore, it cannot perform actions like sending notifications about a pending order upon receipt of additional inventory.

Oracle ATG Web Commerce includes a very simple user interface for administration of the Inventory Manager. You can access the interface through the Dynamo Administration pages, as long as the Dynamo Administration UI is included in your application. Access the Inventory Manager UI using the URL appropriate for your application server. For example, the default URL on JBoss is:

```
http://hostname:8080/dyn/admin/atg/commerce/admin/inventory/index.jhtml
```

See the *ATG Installation and Configuration Guide* to find the default URL.

This page allows an administrator to view the results of the inventory query operations, to manipulate the various properties of each item, and to notify the system of inventory updates. Out-of-the-box, the interface allows the administrator to set, increase, or decrease the `stockLevel`, `backorderLevel`, and `preorderLevel` of any item in the inventory. This page also allows the administrator to set the `stockThreshold`, `backorderThreshold`, `preorderThreshold`, `availabilityStatus`, and `availabilityDate` for each item. is configured through The properties files of the servlet beans described below configure the updated inventory.

The following Dynamo Server Pages manage the inventory:

Dynamo Server Page	Description
`index.jhtml`	This page allows you to administer the `InventoryManager`. Provides access to the functionality mentioned above.
`DisplayInventory.jhtml`	Displays inventory information. You can modify this page to display more or different information for each item in the inventory. It displays the repository ID of the catalog item, the display name, the name of the fulfiller, the `stockLevel`, the `stockThreshold`, the `backorderLevel` the `backorderThreshold`, the `preorderLevel`, the `preorderThreshold`, the `availabilityStatus`, and any bundled items.

Both of these pages use the `InventoryFormHandler` to display and manipulate inventory information. This form handler has two handle methods:

Servlet Bean	Description
handleChangeInventory	Accepts four properties of the InventoryFormHandler: SKU - the ID of the SKU to change value - the value to change the property by changedProperty - the name of the property to change setType - the direction to change it in For example, if SKU="sku-0", value="5", changedProperty="backorderLevel", and setType="increase" then the servlet bean will call: InventoryManager.increaseBackorderLevel("sku-0", 5)
handleUpdateInventory	Uses the IDs stored in property InventoryFormhandler.updatedItemIdList to call InventoryManager.inventoryWasUpdated.

You can also use the InventoryFormHandler to display inventory information. It uses the properties lowerBound, upperBound, batchNumber, batchSize, and propertyName to determine which inventory items to display. The items that should be displayed are populated in the InventoryFormHandler.catalogRefIds. For example, if lowerBound is 'Q', upperBound is 'R', batchSize is 10, batchNumber is 0, and propertyName is "displayName" all the catalogRefIds for the first 10 items in the product catalog with a display name greater than Q and less than R (sorted by displayName) will be in the array InventoryFormHandler.catalogRefIds.

The SKULookup servlet bean is an instance of the ItemLookupRepository that returns information for a given SKU in the repository.

Using the InventoryLookup Servlet Bean

The InventoryLookup servlet bean returns inventory information based on the input parameters. The inventory information returned by this servlet bean includes:

- availabilityStatus: the numerical availability status
- availabilityStatusMsg: a string that maps to the numerical availabilityStatus with the following:

  1000: INSTOCK
  1001: OUTOFSTOCK
  1002: PREORDERABLE
  1003: BACKORDERABLE
  1005: DISCONTINUED

- availabilityDate: The date on which the item will become available.
- stockLevel: The total number of units currently in stock.
- preoderLevel: The total number of units that are available for preorder.
- backorderLevel: The total number of units that are available for backorder.
- stockThreshold: The threshold for the stock level.

- `preorderThreshold`: The threshold for the preorder level.
- `backorderThreshold`: The threshold for the backorder level.

If there is an error retrieving this information, then the `error oparam` will be rendered. All of this information is contained within a single `inventoryInfo` object, which will be rendered within the `output oparam`.

This servlet bean takes one required parameter and one optional parameter.

- The required parameter is `itemId`. The `itemId` is the `catalogRefId` of the product catalog SKU whose inventory information will be retrieved.
- The optional parameter is `useCache`. If set to true, cached data will be retrieved. This data may be out of date, depending on how the inventory is updated, so it should be used with caution. For general store browsing, where performance is critical, `useCache` should be true. If it is essential that the information matches the latest information in the repository, then `useCache` should be false.

The following code sample is an example of using the `InventoryLookup` servlet bean:

```
<dsp:droplet name="/atg/commerce/inventory/InventoryLookup">
 <dsp:param param="link.item.repositoryId" name="itemId"/>
  <dsp:param value="true" name="useCache"/>
  <dsp:oparam name="output">
      This item is
      <dsp:valueof param="inventoryInfo.availabilityStatusMsg"/><br>
      There are
      <dsp:valueof param="inventoryInfo.stockLevel"/>
      left in the inventory.<br>
   </dsp:oparam>
</dsp:droplet>
```

Note: The Inventory Framework can also be configured to use the `LocalizingInventoryDroplet`, which is similar to `InventoryLookup`, but can also include a `localeKey` parameter.

Building a New InventoryManager

This section describes how to replace the existing `InventoryManager` with one of your own. It describes the minimum requirements for implementation and the properties to set throughout inventory and fulfillment. For more information on the fulfillment system, see the *Configuring the Order Fulfillment Framework* (page 413) chapter.

If the Oracle ATG Web Commerce standard `RepositoryInventoryManager` does not meet your inventory management needs, you can create your own class that implements the `InventoryManager` interface. You could also create a new inventory manager if the `RepositoryInventoryManager` is more complex than you need or if you are building a bridge between the Commerce interface and some existing inventory system already in place. The only requirement Commerce has is that the new class implements `InventoryManager`.

You can build a new `InventoryManager` if your sites do not require all the functionality provided by the `InventoryManager`. For example, if you do not allow backorders or preorders, then you can extend the `AbstractInventoryManager`. This provides default implementations of all the methods other than purchase.

The first set of methods to consider when implementing the `InventoryManager` API is purchase, `purchaseOffBackorder`, `purchaseOffPreorder`, `backorder`, and `preorder`. If it is important to

maintain separate levels for backordered and preordered items, the `purchaseOffBackorder` and `purchaseOffPreorder` will need to maintain those, as well as provide the same functionality as purchase. If separate levels are not needed, they could just be implemented as follows: (this depends on your business rules):

```
public int purchaseOffBackorder (String pId, long pHowMany)
throws InventoryException
{
     return purchase(pId, pHowMany);
}
```

To allow an administrator to control the various inventory levels and to allow the system to automatically correct them, you need to implement the following methods:

- `setStockLevel`
- `setBackorderLevel`
- `setPreorderLevel`
- `increaseStockLevel`
- `increaseBackorderLevel`
- `increasePreorderLevel`
- `decreaseStockLevel`
- `decreaseBackorderLevel`
- `decreasePreorderLevel`

`setAvailabilityStatus` and `setAvailabilityDate` also control attributes for the items in inventory. These are important when informing customers on the availability of the items in which they are interested.

If you want to be notified when any of the inventory levels becomes dangerously low, a threshold for each level must be maintained. To allow for this, provide implementations for `setStockThreshold`, `setBackorderThreshold`, and `setPreorderThreshold`.

All the query methods are useful for providing information on each of the items in inventory. The only one of these methods that Commerce depends on out-of-the-box is `queryAvailabilityStatus`.

If your inventory manager will be providing notifications to other systems when inventory is increased, implement `inventoryWasUpdated`.

Configuring a New Inventory Manager

Once you have implemented your inventory manager, the next step is configuring the rest of Oracle ATG Web Commerce to work with it.

First, change the `InventoryManager` component in `atg/commerce/inventory/`.

Out-of-the-box, the class used by this component is `RepositoryInventoryManager`. Change the first line to read;

```
$class=mypackage.MyInventoryManager
```

If you have not changed any of the other references to `InventoryManager`, this should be the only change necessary. By default, `CachingInventoryManager` uses this Nucleus component because its uncached inventory manager and all other components refer to one of these two components. Because your new class implements the `InventoryManager` interface, it can be used throughout Commerce.

If you change any other links, make the appropriate adjustments. The following properties refer to the `InventoryManager`:

- `atg/commerce/fulfillment/HardgoodFulfiller.inventoryManager`
- `atg/commerce/inventory/CachingInventoryManager.uncachedInventoryManager`
- `atg/commerce/inventory/InventoryLookup.inventoryManager`

19 Configuring the Order Fulfillment Framework

Order fulfillment is the set of actions taken by the merchant to deliver the goods or services purchased by the customer. In general, merchandise can be broken up into two groups: *Hard Goods* and *Electronic Goods*. Hard Goods are physical goods that are shipped to the customer. Hard Goods include items such as books, CDs, or toys. Electronic Goods are purchases that do not result in a tangible product that requires shipping. Electronic Goods include software downloads, subscriptions, or gift certificates.

Oracle ATG Web Commerce is designed to handle both of these situations. This chapter describes the design of the order fulfillment framework and presents an overview of the basic classes used to build this infrastructure.

This chapter contains information on the following topics:

Overview of Fulfillment Process

Online shopping can be broken down into two major phases, the purchase process and the fulfillment process. The purchase process is everything that is done before checking out, while the fulfillment process begins after the checkout.

The transition from the purchase process to the fulfillment process occurs when the `SubmitOrder` message is sent out after a successful checkout. The successful delivery of this message signals the transfer of control and the beginning of the fulfillment process.

The `SubmitOrder` message is a JMS `ObjectMessage` that contains the serialized order object. The order is serialized so that fulfillment can be serviced by an entirely independent system.

Building the fulfillment system on top of JMS provides the flexibility of a distributed fulfillment system. For example, a site could contain products from various vendors that can be purchased through the same account. A large site might sell bikes from one vendor and books from another publisher. These orders would require different fulfillers because they would not be fulfilled from the same warehouse. The Purchase Process allows for multiple shipping groups and multiple payment methods. The Fulfillment Process then determines which shipping groups will be fulfilled by which fulfiller and forwards the requests to the relevant fulfillers.

The default implementation assumes that the Purchase Process and all the configured fulfillment systems share the same order repository (or database containing the order information). The architecture is designed to provide the flexibility to accommodate systems with various back-end requirements and fulfillment houses. JMS messages allow this flexibility during the communication between the different subsystems because it serves as an API between the different disparate components. JMS provides the flexibility of integrating with Oracle ATG Web Commerce's fulfillment framework regardless of how and where your existing fulfillment system resides. If the design of your fulfillment system follows the basic pattern defined by Commerce, then extending the basic functionality to support your existing fulfillment system should be straightforward.

Flow of control defines which components have privileges to edit different parts of the order. The basic premise is that once a component has control over a part of the order, only this component should edit this part of the order. Commerce does not verify that a component has the privileges to edit a specific part of an order. Commerce does not perform this verification because if the system is distributed, orders might be modified with a different system.

Commerce also assumes that all changes to the order will be tracked using Modification objects that capture the type of change that occurred. For example, if an item was added to a shipping group, then a `ModifyOrderNotification` message is sent with its array of Modifications including a `GenericAdd` modification. See the Modification Class (page 420) section for more details on modifications.

Commerce assumes that all the components in the system share the same repository. If not all components share the same repository, then `ModifyOrder` and `ModifyOrderNotification` messages can be sent into the vendor's system.

Vendors are responsible for listening for modifications in their subsystems. These modifications might be different from the modifications that Commerce is listening for by default. Vendors might have to implement the various `ModifyOrder` requests needed to synchronize the local order repository with the vendor's back-end systems. Vendors should follow the guidelines for modifying objects and sending out modifications that indicate the types of changes that have occurred. If these guidelines are followed, then it is possible to extend the various modification handlers to maintain accurate copies of the data in the databases.

The following list describes the control flow during the Order Fulfillment process. A diagram following this list illustrates the flow of JMS messages during this process.

1. `OrderFulfiller` receives a `SubmitOrder` message containing a serialized copy of the order. The owner of the order object is the component that receives this message. By default, the `OrderFulfiller` receives this message.

2. The `OrderFulfiller` passes control of the different components to the configured fulfillers using `FulfillOrderFragments`. In this example there is only one fulfiller, the `HardgoodFulfiller`.

   Note: The various fragments contain the shipping groups associated with the items in the fragment. All the shipping groups listed in the fragment are now controlled by the component receiving this message. In this example, the `HardgoodFulfiller` now controls the shipping groups.

3. While the `HardgoodFulfiller` controls the shipping groups, all modifications to the shipping groups take place through the `HardgoodFulfiller`. It is important that no other component modifies these shipping groups while the `HardgoodFulfiller` controls the shipping groups. The `HardgoodFulfiller` could be running on a back-end system in a different environment. If other components need to make changes to the shipping groups, the `ModifyOrder` requests are forwarded to the `HardgoodFulfiller`. The `HardgoodFulfiller` is responsible for making the requested changes to the shipping groups while they are under its control.

   Note: All modifications are performed by fulfillers by calling pipeline chains. For more information, see the *Processor Chains and the Pipeline Manager* (page 363) chapter.

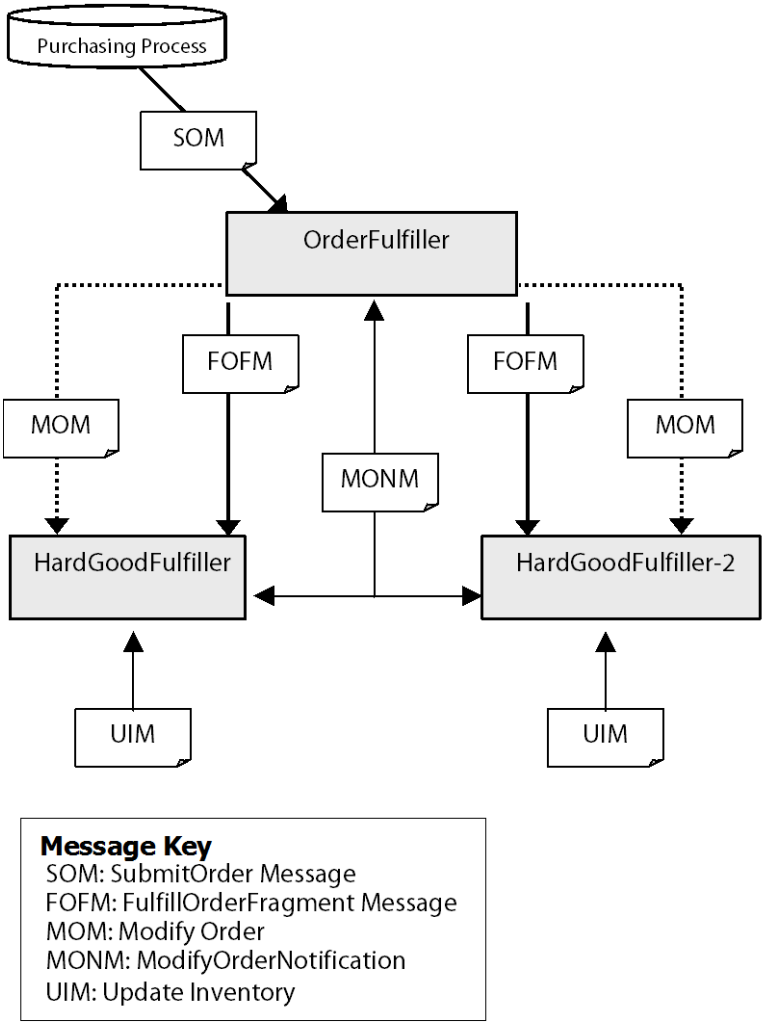
4. When the shipping groups are shipped, a `ModifyOrderNotification` message is sent. When this message is sent, the `HardgoodFulfiller` gives up control of the shipping groups within the order. Control is transferred back to the `OrderFulfiller` automatically if no one else has control until the complete fulfillment of the order. This follows the assumption in the pattern that the `OrderFulfiller` retains control until the order is complete.

5. The `OrderFulfiller` receives the `ModifyOrderNotification` message. If the business rules allow payment to settle on first shipment, then the payment groups are charged with the cost of the items, shipment and taxes. Business rules can also specify that payment be settled upon the shipment of the last shipping group.

6. After the order is settled, the `OrderFulfiller` changes its state to NO_PENDING_ACTION and no longer controls the order.

The following diagram provides an overview of the flow of the JMS messages during the Order Fulfillment process.

Note: By default, the system is set up with one `HardGoodFullfiller`. In this example, the system uses two `HardGoodFullfillers`.



Note: The `OrderFulfiller` is the only class that has control over the payment groups and the only class that can modify the highest-level `Order` object.

The fulfillment system is designed to be a flexible implementation that is easily extensible. This flexibility allows for the different ways businesses handle their fulfillment. If the Commerce order fulfillment system is not appropriate for a site, it is easy to remove the Commerce order fulfillment framework. Remove the Commerce order fulfillment framework by not running a fulfillment server and having another component listen for the `SubmitOrder` message. The `SubmitOrder` should contain all the information necessary for the vendor to start the fulfillment process. See SubmitOrder Class (page 418) for more information.

Note: Commerce supports scenarios where the fulfillment of certain shipping groups has no access to the database.

Running the Fulfillment Server

When you assemble your application, be sure to specify the `Fulfillment` module. For more information on Oracle ATG Web Commerce modules, see the *ATG Platform Programming GuideATG Platform Programming Guide*.

Note: If your Commerce site runs on multiple servers, only one of the instances of the site application should include the `Fulfillment` module.

Order Fulfillment Classes

The fulfillment architecture consists of the following base classes:

Commerce Messages

Modification classes

Other classes

CommerceMessage Class

This class is the base class used by all the object messages sent in the fulfillment process. The properties in this class can be used for tracking messages. The class includes the following properties:

- `source` – identifies the source from which this message was sent. This property can be used to help screen messages.
- `id` – a unique identifying string, the combination of the id and the source should be unique.
- `userId` – the id of the last user to act on this message.
- `originalSource` – the original source of this message. If a message is forwarded on from one component to the next, the `originalSource` never changes but the source does.
- `originalId` – the original message ID as it came from the original source.
- `originalUserId` – the ID of the end user whose action initiated the message.

Although all these fields exist, only `source`, `id`, `originalSource` and `originalId` are used in the fulfillment subsystem. The other fields exist to accommodate extensions to the base class.

SubmitOrder Class

The `SubmitOrder` message is sent to the `OrderFulfiller` when the order is submitted for fulfillment. The `SubmitOrder` message, like all of the other messages in fulfillment, is a serializable object contained within a JMS object message. The message includes a serialized order object containing all the information needed to fulfill the order.

The `SubmitOrder` message is sent at the end of the checkout process. The message is constructed and sent in a processor as part of the `processOrder` chain called `sendFulfillmentMessage`. The properties are in `/atg/commerce/order/processor/SendFulfillmentMessage`.

In the default implementation, the `SubmitOrder` message is sent over `localdms` to the `ScenarioManager` and to the `MessageForwardFilter`. The `MessageForwardFilter` forwards the message over `sqldms` to the `/Fulfillment/SubmitOrder` durable topic. The `OrderFulfiller` is the only listener on that durable topic. See the *Dynamo Message System* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide* for more information. See the OrderFulfiller Class (page 421) section for more details about what happens after the message is received.

FulfillOrderFragment Class

The `OrderFulfiller` sends the `FulfillerOrderFragment` message to the various fulfillment systems responsible for sending out the products in the shipping groups. All shipping groups in an order with the same fulfiller are sent in `FulfillerOrderFragment`. The `FulfillerOrderFragment` message contains the shipping group IDs.

The `FulfillOrderFragment` message, like all of the other messages in fulfillment, is a serializable object contained within a JMS object message. The message includes a serialized order and the list of shipping groups IDs included in this fragment. This message is sent by the `OrderFulfiller` to the various fulfillment systems responsible for fulfilling the shipping groups. When this message is sent, control of the object is transferred to the system that receives the message. By default, this system is the `HardgoodFulfiller`.

ModifyOrder Class

The `ModifyOrder` message extends the `CommerceMessage` class and adds the following properties:

- `orderId` – the ID of the order being modified
- `type` – the JMS message type
- `modifications` – an array of modifications to be performed.

The `ModifyOrder` message allows external sources to request changes to the order object. The list of modifications is included in the message's modifications array. The recipient of the `ModifyOrder` message is responsible for determining whether a modification is possible given the flow of control and the ownership of the objects for which the modification is requested.

After attempting to perform the modification, Oracle ATG Web Commerce sends a `ModifyOrderNotification` referencing the modifications that were requested and indicating whether the modification was successful or not. If a component receives a `ModifyOrder` message for an object to which the component does not have access, the request is forwarded on to other configured systems with access rights to the objects to be modified.

In Commerce, systems only listen for, and work on, one type of Modification in the `ModifyOrder` message. By default, the modification listened for is the one requesting a cancellation of an order. This type is implemented in the `OrderFulfillerModificationHandler` and the `HardgoodFulfillerModificationHandler`, both of which extend the `ModificationHandler` class. These classes are designed to deal with `ModifyOrder` and `ModifyOrderNotification` messages.

The `ModifyOrder` message is received by the `OrderFulfiller`. The `OrderFulfiller` checks that none of the shipping groups have been shipped by examining their states. If any of the shipping groups have been shipped then a `ModifyOrderNotification` message is sent with the requested modifications marked as failed. The sender of the original `ModifyOrder` message is responsible for listening for the `ModifyOrderNotification`.

The `ModifyOrder` message and its modification array are flexible enough to accommodate any changes to the order structure and its subcomponents. However, Commerce implements only the most basic cancel order features because of the variety of business rules that can apply for requested changes and the legality of certain changes.

The `ModificationHandler` class provides the flexibility to change the behavior in handling `ModifyOrder` and `ModifyOrderNotification` messages.

ModifyOrderNotification Class

The `ModifyOrderNotification` class extends `CommerceMessage` and adds the following properties:

- `orderId` – the ID of the order being modified
- `modifyOrderSource` – the originator of the `ModifyOrder` message if this `ModifyOrderNotification` is in response to a `ModifyOrder`
- `modifyOrderId` – the ID of the `ModifyOrder` message if this `ModifyOrderNotification` is in response to a `ModifyOrder`
- `modifications` – an array of the modifications that were made to this order

The `ModifyOrderNotification` message provides a running record of all changes to the order or any of its sub-components. All changes made by components in the system are recorded and distributed in a `ModifyOrderNotification` message. This allows distributed systems a way to keep their various databases synchronized when it pertains to certain aspects of the order. For example, it is possible for a business that has several fulfillers to have each fulfiller use a different backend system.

The control flow described earlier in this section clearly defines which components are responsible for order objects during different points in the fulfillment process. If one of the fulfillers makes a change to a shipping group for which it is responsible, the change is captured in a Modification, which is sent inside a `ModifyOrderNotification`. For example, the fulfiller could change the state of a given item relationship to backordered.

The `ModifyOrderNotifcation` is received by all the systems that are listening for it and it is the responsibility of those systems to update back ends to keep all the systems synchronized. In Oracle ATG Web Commerce,

all the repositories are accessible by all of the components. This eliminates the need to synchronize various disparate databases. However, if a customer requires that a disparate system make modifications to the order objects, the `OrderFulfillerModificationHandler` would need to be augmented to reflect the changes reported by the `ModifyOrderNotification` messages being sent by the disparate systems.

UpdateInventory Class

The `UpdateInventory` message extends the `CommerceMessage` class. It adds one property:

- `itemIds` – a list of IDs of items that were previously unavailable (BACKORDERABLE, PREORDERABLE, or OUT_OF_STOCK) but now have stock available.

`UpdateInventory` is sent by a third party system, such as an inventory subsystem, to indicate that items are available. The `HardgoodFulfiller` in Oracle ATG Web Commerce listens for these messages. The `HardgoodFulfiller` queries the order repository for all shipping groups that contain items from the list that are in a preordered or backordered state.

Modification Class

Each `ModifyOrder` and `ModifyOrderNotification` message contains an array of `Modification` objects. The `Modification` class is the base class for each of these modification objects. All modifications represent some change to a specified Order. In the default implementation of Oracle ATG Web Commerce, there are four types of Modification objects: ADD, REMOVE, UPDATE, or SHIPPING_GROUP_UPDATE. Refer to `GenericAdd`, `GenericRemove`, `GenericUpdate`, and `ShippingGroupUpdate` for more information.

Each Modification also targets a specific kind of object within an Order. For example, the Modification can remove a shipping group. The different possible targets are TARGET_ITEM, TARGET_SHIPPING_GROUP, TARGET_PAYMENT_GROUP, TARGET_ORDER, or TARGET_RELATIONSHIP. A status for each modification indicates success or failure.

The `IdTargetModification` and `IdContainerModification` classes are two abstract subclasses of Modification. For more information, see the *ATG Platform API Reference*.

GenericAdd Class

The `GenericAdd` class is used to add a target specified by ID or value to a target specified by ID or value. For example:

- Add item by ID or value to shipping group by ID or value.
- Add item by ID or value to payment group by ID or value.
- Add item by value to order by ID.
- Add shipping group by value to order by ID.
- Add payment group by value to order by ID.

These are the only valid combinations. Both the ID and the value should not be set for either the target or the container. Everything should be added to the order before it is used as either a target or container for another `GenericAdd`.

For example if you are adding a new item, shipping group, and payment group, and want to add the item to both of the groups you would do the following:

1. Add the item to the order.

2. Add the shipping group to the order.
3. Add the payment group to the order.
4. Add the item to the shipping group.
5. Add the item to the payment group.

GenericRemove Class

The `GenericRemove` class is used to remove an object specified by ID from a container specified by ID. If an item, shipping group, or payment group is removed from an order, it is removed from any relationships.

GenericUpdate Class

This class contains the information that describes a property change for an object. It contains the original value of the property (as a serializable Object) and the new value for the property.

For example, to change the state of a `ShippingGroup` from PENDING_SHIPMENT to NO_PENDING_ACTION (for example ship the shipping group):

- Set the `targetId` of the `GenericUpdate` to the ID of the shipping group to be shipped.
- Set the `containerId` of the `GenericUpdate` to the ID of the order containing that shipping group.
- Set the `propertyName` of the `GenericUpdate` to "state" to update the state property of the shipping group.
- Set the `originalValue` to PENDING_SHIPMENT and set the new value to NO_PENDING_ACTION.

If you included the resulting `GenericUpdate` in a `ModifyOrder` message and sent it to the `OrderFulfiller`, the status of the given shipping group would change to reflect that it has shipped.

ShippingGroupUpdate Class

This special Modification notifies the fulfillment system of any changes within a Shipping Group that happen externally. If the `OrderFulfiller` receives a `ModifyOrderNotification` with a `ShippingGroupUpdate` Modification in it, the shipping groups listed in the Modification are reprocessed.

This is a convenient way of notifying fulfillment of complex changes to an order. It contains an order ID and an array of shipping group IDs that have been updated. After receiving this message, the `OrderFulfiller` forwards it to each of the appropriate fulfillers, who then reprocess the entire shipping group.

PaymentGroupUpdate Class

This special Modification notifies the fulfillment system of any changes within a `PaymentGroup` that happen externally.

OrderFulfiller Class

The `OrderFulfiller` receives the `SubmitOrder` message, which marks the start of the fulfillment process. The fulfillment process relies on a persistent, durable JMS subsystem to deliver messages as they become available. There should be only one instance of an `OrderFulfiller` in place to receive the `SubmitOrder` message. The method invoked on the reception of a message is the `receiveMessage` method. The method will determine what type of message was sent and call the appropriate handle method for that message type.

The following methods handle the different messages:

- `getModificationHandler().handleModifyOrder` – This method handles `ModifyOrder` messages. The handling of `ModifyOrder` messages is delegated to the `ModificationHandler` class, which is configured as a property of the `OrderFulfiller`.
- `getModificationHandler().handleModifyOrderNotification` – This method deals with `ModifyOrder` messages. The handling of `ModifyOrderNotification` messages is delegated to the `ModificationHandler` class, which is configured as a property of the `OrderFulfiller`.
- `handleNewMessageType` – This method is called if the types of the messages don't match up to any of the above three types. By default, this is left as an empty method for future extensibility.
- `handleSubmitOrder` - This method is called to handle all `SubmitOrder` messages. It runs the `handleSubmitOrder` pipeline chain. For more information, see Appendix F, *Pipeline Chains* (page 699).

For information on the handling of `ModifyOrder` and `ModifyOrderNotification` messages, refer to the OrderFulfillerModificationHandler Class (page 423) section.

HardgoodFulfiller Class

The `HardgoodFulfiller` class receives the `FulfillOrderFragment` message and begins the Fulfillment Process for the shipping groups listed within the message. This class is responsible for calling the appropriate pipeline chains.

When a shipping group is shipped, the warehouse notifies the `HardgoodFulfiller`. The `HardgoodFulfiller` then calls the appropriate pipeline chain to change the state of the shipping group and items within it and sends a `ModifyOrderNotification` detailing the changes. For more information on fulfillment pipelines, see the Fulfillment Pipelines (page 722) section of this chapter.

ElectronicFulfiller Class

The `ElectronicFulfiller` is used to fulfill any type of good that is delivered electronically. Electronic goods should be associated with an `ElectronicShippingGroup`. The `ElectronicFulfiller` then fulfills the order by calling the appropriate pipeline chain.

Currently, the only items that use the `ElectronicFulfiller` are gift certificates. The `ElectronicFulfiller` could be used to fulfill any item using the following two actions

- Create a claimable item in the claimable repository
- E-mail user notification (and a claim code) that they have something waiting for them.

For more information on claimable items, see the *Configuring Commerce Services* (page 71) chapter.

Electronic goods are fulfilled differently than hard goods. Electronic goods can take on a variety of forms. `ElectronicFulfiller` creates items in a repository. These items represent an electronic good that the user can then obtain.

The `ElectronicFulfiller` is responsible for fulfilling the order of various electronic goods. `ElectronicFulfiller` fulfills electronic goods by performing two actions:

- Creating an entry in a repository that represents the electronic good that the user can purchase.
- Notifying a user that an electronic good is waiting for them to claim.

By default, Oracle ATG Web Commerce includes a component called `SoftgoodFulfiller`, which is located at `/atg/commerce/fulfillment/SoftgoodFulfiller`. This component is an instance of the `atg.commerce.fulfillment.ElectronicFulfiller`.

OrderFulfillmentTools Class

The `OrderFulfillmentTools` class contains methods that help create messages, modify objects, and manipulate the states in the Order, Shipping Groups, Payment Groups and relationships.

This class is used by fulfillment pipelines. The `OrderFulfillmentTools` class contains various convenience methods for commonly performed tasks in fulfillment. For more information, please refer to the *ATG Platform API Reference*.

The `OrderFulfillmentTools` also maintains the mapping of fulfillers to port names. The `OrderFulfiller` uses these ports to send `FulfillOrderFragment` messages to the correct fulfiller. The `OrderFulfiller` has a different output port for each fulfiller. Messages sent through these ports are written to a different topic for each fulfiller. It is important that each possible value of the fulfiller property of each item in the product catalog is included in this mapping.

OrderFulfillerModificationHandler Class

The `OrderFulfillerModificationHandler` class extends the `ModificationHandler` interface. It is configured to handle the `ModifyOrder` and `ModifyOrderNotification` messages for the `OrderFulfiller` class. The `OrderFulfiller` class contains the `ModificationHandler` property, which deals with both `ModifyOrder` and `ModifyOrderNotification` messages. To change the handling behavior of `ModifyOrder` and `ModifyOrderNotification` messages, extend the `OrderFulfillerModificationHandler` class and change the `ModificationHandler` property of `OrderFulfiller` to point to the new class.

The default implementation deals with the following `ModifyOrder` modifications:

- Remove an order by sending a `ModifyOrder` message containing a Modification of type REMOVE. The `OrderFulfiller` receives this message. If the order and its shipping group are not in a NO_PENDING_ACTION state, then `ModifyOrder` messages are sent to the various fulfillers handling the shipping groups. Every fulfiller who can cancel the shipping group responds by setting the state of the shipping group to PENDING_REMOVE. If all of the shipping group states are changed to PENDING_REMOVE, then the order state changes to REMOVED and all of the shipping group states can be changed to REMOVED. An order cannot be cancelled if any of its shipping groups have been shipped. If you attempt a `GenericRemove` modification on an order that cannot be removed (for example, if one of the shipping groups in the order has shipped) then the order is set to PENDING_MERCHANT_ACTION.
- Notify the fulfillment system that a shipping group has shipped by sending a `ModifyOrder` message with a `GenericUpdate` that changes the state of the shipping group from PENDING_SHIPMENT to NO_PENDING_ACTION. The `OrderFulfiller` will receive this message and forward it to the responsible fulfiller. For more information, see the GenericUpdate Class (page 421) section.

The default implementation deals with the following `ModifyOrderNotification` modifications:

- Shipping group's state changes to NO_PENDING_ACTION, PENDING_MERCHANT_ACTION, PENDING_REMOVE, or failure to change to PENDING_REMOVE.
- When a customer updates a shipping group, the `OrderFulfiller` sends a `ModifyOrderNotification` message to the fulfiller responsible for this shipping group. This forces a reprocessing of the shipping group.

The Oracle ATG Web Commerce default implementation settles payment on first or last shipment of the shipping groups. You can configure when to charge payment in the `SettleOnFirstShipment` property of the `OrderFulfiller`. By default, charging takes place after the shipment of the last shipping group. The settlement is for the total value of the order. If settlement occurs on first shipment and a shipping group that hasn't been shipped is cancelled, a credit must be issued for the items that were paid for but never shipped.

The extendible infrastructure allows all types of `ModifyOrder` messages and `ModifyOrderNotifications` depending on your business requirements.

HardgoodFulfillerModificationHandler Class

The `HardgoodFulfillerModificationHandler` deals with the `ModifyOrder` and `ModifyOrderNotifications` messages received by the `HardgoodFulfiller`. The `HardgoodFulfiller` contains a `ModificationHandler` property, which is set by default to the `HardgoodFulfillerModificationHandler`. This class is similar to the `OrderFulfillerModificationHandler`.

To change the handling behavior of `ModifyOrder` and `ModifyOrderNotification` messages, extend the `HardgoodFulfillerModificationHandler` class and change the `ModificationHandler` property `HardgoodFulfiller` to point to the new class.

The default implementation deals with the following `ModifyOrder` modification:

- Remove the shipping group from the order:

  The fulfillers can remove shipping groups if they have not been shipped. Determining whether a shipping group has been shipped can be difficult because of the asynchronous nature of shipping items. Consulting the states may not be enough to determine if the group has been shipped. Oracle ATG Web Commerce consults the state to make sure that it isn't in a NO_PENDING_ACTION or REMOVED state. This is sufficient because in the default Commerce configuration, there is no integration with a real warehouse so shipment is indicated by changing a set of states in the order repository. Some vendors might decide to create business rules that limit the time in which cancellations can occur because it is difficult to determine the exact shipping time for a shipping group.

- Ship the shipping group:

  The `HardgoodFulfiller` can be notified that a shipping group has shipped through a `ModifyOrder` message (which is originally sent to the `OrderFulfiller`, then forwarded to the `HardgoodFulfiller`). The `HardgoodFulfiller` gets a `GenericUpdate` modification through the `ModifyOrder` message, checks the current state of the shipping group to ensure that it is PENDING_SHIPMENT. If everything is fine, it sets the state to NO_PENDING_ACTION and notifies the rest of the system of the change with a `ModifyOrderNotification` message.

It also handles the following `ModifyOrderNotification` modification:

- Shipping group update:

  A shipping group is re-processed when the method `processMultipleShippingGroups` in `HardgoodFulfiller` is called. This method is called when a modification of type SHIPPING_GROUP_UPDATE is received. The `HardgoodFulfiller` does not listen on the topic over which `ModifyOrderNotification` messages are sent. Instead, the `OrderFulfufiller` listens on that topic and forwards the appropriate messages directly to the `HardgoodFulfiller` using the port in defined in `OrderFulfillmentTools.fulfillerPortNameMap`.

Using Locking in Fulfillment

An important concept in the message processing in the fulfillment process is that no component should handle more than one message per order at any given time.

For example, a component is currently handling a `ModifyOrderNotification` message for `orderId` '1234' and a `ModifyOrder` message for `orderId` '1234' is received during processing. The `ModifyOrder` message blocks and waits until the first message finishes running. This does not prohibit any messages that come for

another `orderId` from being processed. This is accomplished with locking and the `ClientLockManager`. All fulfillment components use the lock manager located at `/atg/dynamo/service/ClientLockManager`.

The lock acquired is for the key that is returned by the method `getKeyForMessage` in `OrderFulfiller` and `HardgoodFulfiller`. The default implementation returns the `orderId` specified in the message. This method can be overridden if the key to determine the locking needs to be changed but you want to preserve the principle of having one message per key/message at a time.

The design of the `OrderFulfiller` and the `HardgoodFulfiller` uses the `ClientLockManager` to prevent one component from processing messages for two different orders at the same time. Extending the `ModificationHander` for either class does not require any locking changes. The only time you should be concerned with locking is if the `HardgoodFulfiller` is not extended when a fulfiller class is created.

Note: Every `ClientLockManager` (one per Oracle ATG Web Commerce instance) should be configured to point to the Commerce instance running the `ServerLockManager`. Every Commerce component should use the same `ClientLockManager`. For more information on the `ServerLockManager`, see the *SQL Repository Caching* chapter in the *ATG Repository Guide*.

The following example demonstrates how the lock manager is used:

```
TransactionDemarcation td = new TransactionDemarcation();
try {
       td.begin(getTransactionManager(), td.REQUIRED);
       getClientLockManager().acquireWriteLock(pOrderId);
       LockReleaser lr = new LockReleaser(getClientLockManager(),
                getTransactionManager().getTransaction());
       lr.addWriteLock(pOrderId);

           <insert your code here>

catch (DeadlockException de) {
       if(isLoggingError())
             logError(de);
       return false;
}
catch (LockManagerException lme) {
       if(isLoggingError())
             logError(lme);
       return false;
}
catch(TransactionDemarcationException t) {
      if(isLoggingError())
             logError(t);
      return false;
}
finally {
     try {
        td.end();
     }
     catch(TransactionDemarcationException tde) {
        if(isLoggingError())
              logError(tde);
     }
}
```

The `LockReleaser` registers each lock with the transaction. The lock is released when the transaction ends. Because of this, it is imperative that a transaction be in place when the `LockReleaser` is created. This is the

reason for all the code using `TransactionDemarcation`. For more information on transactions, see the *Transaction Management* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

Using the OrderFulfiller Interface

The `OrderFulfiller` is the interface to the fulfillment system through which all other systems communicate. Messages intended for any object within fulfillment are sent to the `OrderFulfiller` first. The `OrderFulfiller` calls the appropriate fulfillment pipeline, which forwards the message to the appropriate place. This section describes the functionality of the `OrderFulfiller`.

There is only one instance of the `OrderFulfiller`. Order fulfillment begins when the `OrderFulfiller` receives the `SubmitOrder` message containing an order. Three things happen when the `OrderFulfiller` receives the `SubmitOrder` message:

1. The states of the order and each shipping group are set to `PROCESSING`.
2. `OrderFulfiller` determines the fulfiller for each shipping group. If the items in a shipping group have more than one fulfiller then that shipping group is split. After the split, each shipping group can be fulfilled by exactly one fulfiller.
3. The `OrderFulfiller` creates `FulfillOrderFragment` messages and sends the messages to each fulfiller. These messages include the shipping group IDs that the fulfiller is responsible for fulfilling. For details on what the fulfiller does with that message, see the Using the Fulfiller (page 427) section.

After sending the `FulfillOrderFragment` messages, `OrderFulfiller` relinquishes control of the order until the fulfiller has performed all necessary functions. The `OrderFulfiller` listens for all `ModifyOrderNotification` messages for events notifying it that the fulfiller is finished.

The following situations describe how the `OrderFulfiller` regains control of an order:

- If the shipping group's state is set to `PENDING_MERCHANT_ACTION`, the `OrderFulfiller` will set the order's state to `PENDING_MERCHANT_ACTION`. The customer service representative or someone representing the merchant must change this state back to `PROCESSING`. The system should then be notified (through a `ShippingGroupUpdate`) message to reprocess each shipping group.
- The shipping group's state is set to `NO_PENDING_ACTION` after that shipping group is shipped to the customer. The `OrderFulfiller` checks if the order can be settled and cost of the order can be charged to the customer. This is done with a method in `OrderFulfillmentTools` called `isOrderSetttleable`. Oracle ATG Web Commerce allows orders to be settled at one of two points: after the first shipping group has shipped, or after all the shipping groups have shipped. This behavior can be configured through a property in the `OrderFulfiller`:

  ```
  OrderFulfiller.settleOnFirstShipment
  ```

  The default value of this property is false. If the order can be settled, then the following method is called:

  ```
  protected void settleOrder(Order pOrder, List
  pModificationList) throws PaymentException
  ```

  This method uses the `PaymentManager` to debit each `PaymentGroup` in the order.

  After determining settlement, the `OrderFulfiller` uses the `isOrderFinished` method in `OrderFulfillmentTools` to determine if the order is complete.

OrderFulfillmentTools.isOrderFinished returns true if all the shipping groups are in a NO_PENDING_ACTION state and all the PaymentGroups are in a SETTLED state. If the order is finished, the OrderFulfiller calls the following method:

```
protected void finishOrder(Order pOrder, List
pModificationList)
```

This sets the order into a NO_PENDING_ACTION state, indicating that all processing on this order is complete.

Using the Fulfiller

Oracle ATG Web Commerce includes two fulfiller objects: the HardgoodFulfiller and the ElectronicFulfiller. This section will describe how fulfillers are used in the fulfillment process, how to replace a fulfiller, or add a new fulfiller of your own. Most of the examples in this section refer to the HardGoodFulfiller. For information on the ElectronicFulfiller, see the ElectronicFulfiller Class (page 422) section.

The fulfiller's responsibility for an order begins with the receipt of the FulfillOrderFragment message. Prior to this, the OrderFulfiller owns the order. The first method called is:

```
public void receiveMessage (String pPortName, Message pMessage)
      throws JMSException
```

This method is inherited from the MessageSink interface. HardgoodFulfiller is actually a subclass of SourceSinkTemplate, which implements the MessageSink. For more information, see the *ATG Platform API Reference*. All that this method does is get the JMS object message that was sent, check the contained objects type and call the appropriate method. In the case of a FulfillOrderFragment message, the method called is:

```
public void handleFulfillOrderFragment(String pPortName,
      ObjectMessage pMessage) throws JMSException
```

This method begins the processing of each shipping group included in the FulfillOrderFragment (pMessage in the signature above).

Notifying the HardgoodFulfiller of a Shipment

The HardgoodFulfiller can be notified that a shipping group has been shipped in three different ways. All three methods call shippingGroupHasShipped of the HardgoodFulfiller. This takes the order ID and the shipping group ID.

- /atg/commerce/admin/en/fulfillment/ShippableGroups.jhtml is a Dynamo Server Page where the order and shipping group IDs can be specified. This page only lists shipping groups whose state is PENDING_SHIPMENT and whose shipOnDate is the current date or earlier (or null). This page also allows you to print an order and notify fulfillment of an order's shipment. For more information of the *ATG Commerce Fulfillment Administration* section of the *ATG Commerce Guide to Setting Up a Store*.
- The atg.commerce.fulfillment.HardgoodShipper method is a scheduled service that can be scheduled to run when it is convenient for the store. For instructions on setting up a scheduled service, see the *Scheduler*

Services section of the *Core Dynamo Services* chapter of the *ATG Platform Programming GuideATG Platform Programming Guide*. This service will process shipping groups whose state is `PENDING_SHIPMENT` and whose `shipOnDate` is the current date or earlier (or null).

Note: Start the `HardgoodShipper` by setting the `hardgoodShipper` property in the `HardgoodFulfiller` component. The `HardgoodFulfiller` component is located in the Nucleus path: `atg/commerce/fulfillment`. If you change the `HardgoodShipper` schedule, you must redeploy the application that includes the fulfillment server. See the Running the Fulfillment Server (page 417) section for more information.

- Send a `ModifyOrder` message with a `GenericUpdate` modification to the `OrderFulfiller` setting the state of a shipping group to `NO_PENDING_ACTION`. The `OrderFulfiller` will forward this message to the appropriate fulfiller.

The scheduled service queries the repository for all shipping groups with the state PENDING_SHIPMENT. It then calls the `shippingGroupHasShipped` of the `HardgoodFulfiller`.

The scheduled service cannot actually communicate with the warehouse. This scheduled service is useful if you do not have an extensive backend system and need a way to automatically mark shipping groups as shipped.

HardGoodFulfiller Examples

The following examples demonstrate the behavior of the `HardgoodFulfiller` in different situations.

Example 1: An order is received with one item that is IN_STOCK:

- `FulfillOrderFragment` received with one shipping group.
- The one item in the shipping group successfully allocates. The `ShippingGroupCommerceItemRelationship's` state is set to PENDING_DELIVERY.
- The state of the `ShippingGroup` is set to PENDING_SHIPMENT.
- The group ships. When the `HardgoodFulfiller` is notified, the shipping group is set to NO_PENDING_ACTION and the item is set to DELIVERED.

Example 2: `HardgoodFulfiller.outOfStockIsError=false` and an order is received with one item that is OUT_OF_STOCK:

- `FulfillOrderFragment` received with one shipping group.
- The 1 item in the shipping group fails to allocate but is successfully backordered. The state of `ShippingGroupCommerceItemRelationship` is set to OUT_OF_STOCK. (This example also applies to BACK_ORDERED and PRE_ORDERED items.)
- Some time later, an `UpdateInventory` message is received notifying the `HardgoodFulfiller` that the item has new inventory available. The item is reallocated.
- The state of the `ShippingGroup` is set to PENDING_SHIPMENT.
- The group ships. When the `HardgoodFulfiller` is notified, the shipping group is set to NO_PENDING_ACTION and the item is set to DELIVERED.

Example 3: `HardgoodFulfiller.outOfStockIsError=true` and an order is received with one item that is OUT_OF_STOCK:

- `FulfillOrderFragment` received with one shipping group.

- The one item in the shipping group fails to allocate but is successfully backordered. The state of `ShippingGroupCommerceItemRelationship` is set to OUT_OF_STOCK.
- Since `outOfStockIsError` is true, the shipping group's state gets set to PENDING_MERCHANT_ACTION.
- At this point, it is the responsibility of the Customer Service Representative to correct the problem.

Example 4: `HardgoodFulfiller.allowPartialShipmentDefault=true` and `HardgoodFulfiller.outOfStockIsError=false` and an order is received with one item that is out of stock and one item that is available.

- `FulfillOrderFragment` received with one shipping group containing two items.
- The first item in the shipping group fails is out of stock. The state of `ShippingGroupCommerceItemRelationship` is set to OUT_OF_STOCK.
- The second item in the shipping group successfully allocates. Its state is set to PENDING_DELIVERY.
- Since `allowPartialShipment` is true, the shipping group is split into two shipping groups. The first contains the PENDING_DELIVERY item and its state is set to PENDING_SHIPMENT. The second shipping group contains the out of stock item and its state remains PROCESSING.
- The first shipping group continues similarly to the fourth step in example 1. The second shipping group continues similarly to the third step in example 2.

Example 5: An order is received with one item that has no information in the inventory system.

- `FulfillOrderFragment` received with one shipping group.
- The one item in the shipping group is not found. The state of `ShippingGroupCommerceItemRelationship` is set to ITEM_NOT_FOUND.
- The state of `ShippingGroup` is set to PENDING_MERCHANT_ACTION.

Example 6: An order contains a shipping group in a state of PENDING_MERCHANT_ACTION. `HardgoodFulfiller.shippingGroupHasShipped` is called with this shipping group.

- An error is logged stating that a shipping group that was not in a PENDING_SHIPMENT state cannot be shipped.
- `shippingGroupHasShipped` returns false.

Example 7: An order contains a shipping group in a state of PENDING_SHIPMENT. A `ModifyOrder` message with a `GenericRemove` Modification on the order is received by the `OrderFulfiller`.

- The `OrderFulfiller` sets the order to PENDING_REMOVE.
- The `OrderFulfiller` sends a `ModifyOrder` message with a `GenericRemove` modification for each shipping group in the order to the fulfiller.
- The fulfiller receives the `ModifyOrder`.
- The state of the `ShippingGroup` is set to PENDING_REMOVE.
- Each of the items in the shipping group has its state set to PENDING_REMOVE.
- If the item was PENDING_DELIVERY, the stock level in the inventory is increased.
- If the item was BACK_ORDERED, the backorder level in the inventory is increased.
- If the item was PRE_ORDERED, the preorder level in the inventory is increased.

- A `ModifyOrderNotification` is sent with each of the updates.
- This message is received by the `HardgoodFulfiller`. The `HardgoodFulfiller` sets the state of each object to REMOVED.
- `HardgoodFulfiller` sends a `ModifyOrderNotification` with these updates.
- `OrderFulfiller` receives the `ModifyOrderNotification`, when all shipping groups are REMOVED, `OrderFulfiller` sets the order and each `CommerceItem` to REMOVED.

Example 8: An order contains a shipping group in a state of NO_PENDING_ACTION. A `ModifyOrder` message with a `GenericRemove` Modification on the order is received by the `OrderFulfiller`.

- The `OrderFulfiller` sets the order to PENDING_REMOVE.
- The `OrderFulfiller` sends a `ModifyOrder` message with a `GenericRemove` modification for each shipping group in the order to the fulfiller.
- The fulfiller receives the `ModifyOrder`.
- The shipping group cannot be removed since it has already shipped.
- A `ModifyOrderNotification` is sent with the original `GenericRemove` message set to a status of STATUS_FAILED.
- This message is received by the `OrderFulfiller`. The `OrderFulfiller` sets the state of the order to PENDING_MERCHANT_ACTION.

Creating a New Fulfiller

You can create a new fulfiller if your sites require fulfillment functionality different from that of the `HardgoodFulfiller` or `ElectronicFulfiller` that ship with Oracle ATG Web Commerce. For example, you could create a new fulfiller if your customers can purchase a research report. Instead of allocating this report from inventory and shipping it to the customer via UPS or Federal Express, you send them the report as an attachment via e-mail. There is no need to check inventory since the report is automatically in stock. Shipment can be done immediately since your newly created fulfiller can interface directly with an e-mail system. This section will describe how to implement and configure this new simpler fulfiller. In this example, the new fulfiller is called `MyOwnFulfiller`.

JMS messages connect the new fulfiller and the rest of Commerce. Specifically, fulfillment of a shipping group begins with a `FulfillOrderFragment` message. To receive these messages, the new fulfiller must implement `atg.dms.patchbay.MessageSink`.

In addition to receiving the `FulfillOrderFragment` messages, `MyOwnFulfiller` must send out messages to inform the rest of Commerce about changes made to each order. To send these messages, `MyOwnFulfiller` must implement `atg.dms.patchbay.MessageSource`. There is a class in `atg.commerce.messaging` that provides most of the functionality `MyOwnFulfiller` will need to send and receive messages. This class is called `SourceSinkTemplate`. It implements both `MessageSource` and `MessageSink`. `MyOwnFulfiller` will be defined as follow:

```
package myPackage;
import atg.commerce.fulfillment.*;
import atg.commerce.messaging.*;
```

```
public class MyOwnFulfiller extends SourceSinkTemplate
```

The only method in `SourceSinkTemplate` that `MyOwnFulfiller` needs to overwrite is `receiveMessage`.

```
public void receiveMessage (String pPortName, Message pMessage)
      throws JMSException
{
```

The Patch Bay system calls this method when a message is sent to `MyOwnFulfiller`. For more information, see the *Dynamo Message System* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*. At this time, the only message `MyOwnFulfiller` listens for is `FulfillOrderFragment`. The `receiveMessage` method can check the type of the object in the message. Only the interested code will be listed here, error checking will be assumed.

```
if(pMessage.getJMSType().equals(FulfillOrderFragment.TYPE))
     handlFulfillOrderFragment(pMessage);
} // end of receiveMessage
```

The `handleFulfillOrderFragment` method retrieves the order from the message and the list of shipping groups that need to be fulfilled. For this example, the necessary error checking will be listed here.

1. Retrieve the `FulfillOrderFragment` `CommerceMessage` from the incoming message:

   ```
   public void handleFulfillOrderFragment(ObjectMessage pMessage)
   {
   FulfillOrderFragment fragment =
   (FulfillOrderFragment) pMessage.getObject();
   ```

2. Retrieve the order and the shipping groups IDs:

   ```
   Order order = fragment.getOrder();
   String[] shippingGroupIds = fragment.getShippingGroupIds();
   ```

3. Call a new method called `processShippingGroup` for each shipping group in the message.

   ```
   for(int i=0; i<shippingGroupIds.length; i++) {
   ShippingGroup sg =
   order.getShippingGroup(shippingGroupIds[i]);
   processShippingGroup(order, shippingGroup);
   }
   ```

4. One of the responsibilities of `MyOwnFulfiller` is to notify the rest of Oracle ATG Web Commerce what changes were made to the order included in this `FulfillOrderFragment`. To do this, `MyOwnFulfiller` needs to remember each modification made. Therefore, inside `handleFulfillOrderFragment`, declare a new List that will contain `Modification` objects. These objects are added to the list as changes are made. This list will need to be included in any method calls. The above code now looks like this:

   ```
   List modifications = new ArrayList();
   for(int i=0; i<shippingGroupIds.length; i++) {
   ShippingGroup sg =
   order.getShippingGroup(shippingGroupIds[i]);
   processShippingGroup(order, shippingGroup,
   modifications);
   }
   ```

After processing is complete, `MyOwnFulfiller` saves the order and include the modifications in a message. This is handled by a method in the `OrderFulfillmentTools`:

```
OrderManager orderManager = getOrderManager();
OrderFulfillmentTools tools = getOrderFulfillmentTools();

orderManager.updateOrder(order);
tools.sendModifyOrderNotification(order.getId(), // order
   modificationList, // modifications
   null,             // ModifyOrder
   this,             // MessageSource
   getModifyOrderNotificationPort(),// port
   null);            // original message
}  // end of handleFulfillOrderFragment
```

See the *ATG Platform API Reference* for a detailed description of the `OrderFulfillmentTools` `sendModifyOrderNotification` method.

`MyOwnFulfiller` depends on a few properties being set to properly function. It will need an `OrderManager`, `OrderFulfillmentTools`, and a `ModifyNotificationPort`. This port is explained in the Configuring a New Fulfiller (page 433) section.

The only method that has not been discussed is `processShippingGroup`. The following example is a simple example of the functions of `processShippingGroup` in order processing. The most important function is the manipulation of the object states. There are two more properties necessary for the fulfiller: `ShippingGroupStates` and `ShipItemrRelationshipStates`.

```
public void processShippingGroup(Order pOrder,
                ShippingGroup pShippingGroup,
                List pModificationList)
{
   ShippingGroupStates sgStates = getShippingGroupStates();
   ShipItemRelationshipStates sirStates =
     getShipItemRelationshipStates();
   OrderFulfillmentTools tools =
     getOrderFulfillmentTools();

. . . get each item relationship in the shipping group

. . . for each item

  . . . send the report via e-mail using the orders
       profile
     // set the state to DELIVERED
     tools.setItemRelationshipState(shipItemRel,
        sirStates.getStateValue(sirStates.DELIVERED),
        "Report has been e-mailed to customer",
        pModificationList);
. . . end for loop

// the shipping groups is finished
  tools.setShippingGroupState(pShippingGroup,
     sgStates.getStateValue(sgStates.NO_PENDING_ACTION),
     "This shipping group has finished shipping",
     pModificationList);
}
```

The methods in `OrderFulfillmentTools` create the appropriate modifications, which are sent in a `ModifyOrderNotification` message by `handleFulfillOrderFragment`. The new fulfiller is now ready to be configured into Oracle ATG Web Commerce.

It is possible that the e-mail cannot be sent because of some error. For example, if a message is sent out because the shipping group cannot be shipped and the message contains an invalid e-mail address. One possibility for dealing with this error is to set the `ShippingGroup` to `PENDING_MERCHANT_ACTION`. If you implement your fulfiller to do this, then the Customer Service Representative must correct the order and tell the fulfillment system to reprocess that shipping group with a `ShippingGroupUpdate` Modification sent within a `ModifyOrderNotification` message.

To facilitate this, `MyOwnFulfiller.receiveMessage` should be configured to listen for `ModifyOrderNotification` messages and call `handleModifyOrderNotification` if one of these messages is received. That method can then call `processShippingGroup` for each shipping group and send a new `ModifyOrderNotification` with all modifications that were made.

A new fulfiller must be configured within Nucleus before it can be used by the fulfillment system. See the Configuring a New Fulfiller (page 433) section for more information.

Configuring a New Fulfiller

A new fulfiller must be configured within Nucleus before it can be used by the fulfillment system. This section uses the example of configuring `MyOwnFulfiller`, the fulfiller created in the Creating a New Fulfiller (page 430) section.

Use the ATG Control Center to edit the `Configuration` component located in `atg/commerce/fulfillment/`.

1. Change the property `fulfillerPortNameMap` to include the name of this new fulfiller.

2. Configure the port on which messages will be sent to this fulfiller. This is the port on which the `OrderFulfiller` component will send JMS messages for this fulfiller.

   For example, add the following to `Configuration.properties`:

   ```
   fulfillerPortNameMap+=\
   MyOwnFulfiller=MyOwnFulfillerPort
   ```

   You will also need to define which types of shipping groups can be handled by your fulfiller. `OrderFulfiller` uses this information to verify that a shipping group can be fulfilled by its fulfiller. For example, add the following to `Configuration.properties`:

   ```
   fulfillerShippingGroupMap+=\
   MyOwnFulfiller=mypackage.MyShippingGroup
   ```

   In this example, the fulfiller being added is called `MyOwnFulfiller`. The component using an instance of `HardgoodFulfiller` should make the name property of the `HardgoodFulfiller` "`MyOwnFulfiller`"

   For example, add the following to `MyOwnFulfiller.properties`:

   ```
   fulfillerName=MyOwnFulfiller
   ```

   In addition, add the following properties to `MyOwnFulfiller.properties`:

   ```
   orderManager^=Configuration.orderManager
   orderFulfillmentTools^=Configuration.orderFulfillmentTools
   messageSourceName=MyOwnFulfiller
   ```

```
modifyNotificationPort=ModifyNotificationPort
shippingGroupStates=/atg/commerce/states/ShippingGroupStates
shipItemRelationshipStates=
/atg/commerce/states/ShipItemRelationshipStates
```

3. Configure the `MyOwnFulfillerPort` in the `dynamoMessagingSystem.xml` file so that the `OrderFulfiller` component can send out the `FulfillOrderFragment` messages on this port.

   For example, add the following `dynamoMessagingSystem.xml` for `OrderFulfiller`:

```
<message-filter>
<nucleus-name>
/atg/commerce/fulfillment/OrderFulfiller
</nucleus-name>
. . .
<output-port>
<port-name>
MyOwnFulfillerPort
</port-name>
<output-destination>
<provider-name>
sqldms
</provider-name>
<destination-name>
sqldms:/Fulfillment/MyOwnGoods
</destination-name>
<destination-type>
Topic
</destination-type>
</output-destination>
</output-port>
. . .
</message-filter>
```

4. Configure the `MyOwnFulfiller` component to send messages on the `modifyNotificationPort` and listen for messages on the `sqldms:/Fulfillment/MyOwnGoods` topic. These topics are described above.

```
<message-filter>
<nucleus-name>
/myPackage/MyOwnFulfiller
</nucleus-name>
<output-port>
<port-name>
ModifyNotificationPort
</port-name>
<output-destination>
<provider-name>
sqldms
</provider-name>
<destination-name>
sqldms:/Fulfillment/ModifyOrderNotification
</destination-name>
<destination-type>
Topic
</destination-type>
```

```
</output-destination>
</output-port>
</message-filter>
<message-sink>
<nucleus-name>
/myPackage/MyOwnFulfiller
</nucleus-name>
<input-port>
<port-name>
DEFAULT
</port-name>
<input-destination>
<provider-name>
sqldms
</provider-name>
<destination-name>
sqldms:/Fulfillment/MyOwnGoods
</destination-name>
<destination-type>
Topic
</destination-type>
</input-destination>
</input-port>
</message-sink>
```

For more information, see the *Dynamo Message System* chapter of the *ATG Platform Programming GuideATG Platform Programming Guide*.

5. Set the `MessageSourceName` property of the `MyOwnFulfiller` to "`MyOrderFulfiller`" or another value that indicates who sent a message. This allows the component to ignore messages that it sent itself.

6. Add another value to the fulfiller property of the SKU in the product catalog. (Defined in `/atg/commerce/catalog/productCatalog.xml`) This should match the name of the fulfiller used to map to a port in `OrderFulfillmentTools.fulfillerPortNameMap`.

```
<item-descriptor name="sku" display-name="SKU"
sub-type-property="type"
display-property="displayName"
. . .
<property name="fulfiller" data-type="enumerated"
column-name="fulfiller" queryable="false">
<attribute name="useCodeForValue" value="false"/>
<option value="HardgoodFulfiller" code="0"/>
<option value="SoftgoodFulfiller" code="1"/>
<option value="MyOwnFulfiller" code="2"/>
</property>
. . .
</item-descriptor>
```

The `modificationHandler` property can be modified to point to another component that extends `atg.commerce.fulfillment.ModificationHandler` to handle with different forms of modifications received by the fulfiller. The `ModificationHandler` class provides a simple framework for changing the handling of `ModifyOrder` and `ModifyOrderNotifications`. It is not necessary to use a separate `ModificationHandler`. In the example above, `handleModifyOrderNotification` was implemented directly within the fulfiller class `MyOwnFulfiller`.

Order Fulfillment Events

Order Fulfillment events are created and sent by the `OrderChangeHandler` in the Order Fulfillment system. The `OrderChangeHandler` listens to the various `ModifyOrderNotifications` being delivered in the system and constructs one of three basic events:

Events	SubType
`OrderModified`	FINISHED HASUNAVAILABLEDITEMS PENDING_MERCHANT_ACTION REMOVED
`ShippingGroupModified`	SHIPPED SPLIT SPLITFORSHIPPING SPLITFORFULFILLER PENDING_MERCHANT_ACTION REMOVED (if subtype is removed, then) the `shippingGroup` property will be null.
`PaymentGroupModified`	CREDITED DEBITED DEBIT_FAILED

Note: The `OrderChangeHandler` component includes a property called `sendEventsWithNoProfile`. If this property is set to false, the `sendScenarioEvent()` method does not send events if the profile is null.

These three basic messages are sent with their subtypes set according to the reason it is being sent.

Actions in Oracle ATG Web Commerce include the delivery of promotions when certain conditions are met. For example, give a promotion to the user if the `OrderFinished` event contains an order whose total is greater than $50.

Other actions include sending e-mails to a customer about progress of their order. For example, an e-mail sent indicating that a shipping group has been shipped. This e-mail might include information about when the shipping group will be received and the tracking information.

The information necessary for the actions should be in the profile, order, payment group, or shipping group objects that are included in the event. See the list below for a list of the objects included in each of the messages.

The following extend Scenario Event and include the profile and the JMS type.

Message type	Objects included
`OrderModified`	Includes profile, order and an array of `ShippingGroupCommerceItemRelationships`.
`ShippingGroupModified`	Includes the profile, order, shipping group, and new shipping group.

Message type	Objects included
`PaymentGroupModified`	Includes the profile, order and an array of the payment groups.

Fulfillment Server Fault Tolerance

Because the Oracle ATG Web Commerce fulfillment framework uses SQL JMS messages, you do not need a complex system of redundant fulfillment servers with automatic failover. SQL JMS messages are persistent and saved until successfully delivered. If the fulfillment server goes down, re-assemble and redeploy the application that includes it.

Fulfillment work occurs within the context of a transaction (started by the SQL JMS system). If the fulfillment server goes down, all current transactions roll back. The message is resent after a transaction rolls back because message delivery and processing occur within the same transaction. Any messages that are sent to the fulfillment server while it is down, including those that are resent, are persistent in the database and will be delivered once the fulfillment server is back online.

Fulfillment Message Redelivery

The Fulfillment module of Oracle ATG Web Commerce uses Patch Bay to receive order messages. Patch Bay delivers messages from the fulfillment-related JMS destination to a fulfillment `MessageSink`. Fulfillment uses the Pipeline Manager to allow the execution of transactional, multi-stage, fulfillment-related processes.

Commerce uses the message redelivery features in Patch Bay. The Commerce Fulfillment Patch Bay configuration document contains configuration settings to enable the message redelivery. These configurations are set in `/atg/dynamo/messaging/dynamoMessagingSystem.xml` in the Fulfillment Module.

For more information on Patch Bay, see the *Dynamo Message System* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

The following messaging components and destinations are configured for redelivery ports.

- Component: `/atg/commerce/fulfillment/OrderChangeHandler`
  Destination: `patchbay:/Fulfillment/ModifyOrderNotification`
- Component: `/atg/commerce/fulfillment/OrderFulfiller`
  Destination: `patchbay:/Fulfillment/ModifyOrderNotification`
- Component: `/atg/commerce/fulfillment/OrderFulfiller`
  Destination: `patchbay:/Fulfillment/SubmitOrder`
- Component: `/atg/commerce/fulfillment/OrderFulfiller`
  Destination: `patchbay:/Fulfillment/ModifyOrder`
- Component: `/atg/commerce/fulfillment/HardgoodFulfiller`
  Destination: `patchbay:/Fulfillment/HardGoods`
- Component: `/atg/commerce/fulfillment/SoftgoodFulfiller`
  Destination: `patchbay:/Fulfillment/SoftGoods`

Each of these was given the following redelivery port configuration:

- max-attempts: 3
- delay: 60000
- failure-output-port: `FulfillmentError`
- redelivery-port: `FulfillmentError`
- output destination: `patchbay:/Fulfillment/ErrorNotification`
- output destination: `patchbay:/Fulfillment/DeadMessageQueue`

In addition, `/atg/commerce/fulfillment/HardgoodFulfiller` component with the `patchbay:/Fulfillment/UpdateInventory` destination is configured to use redelivery.

- max-attempts: 3
- delay: 60000
- failure-output-port: `UpdateInventoryError`
- redelivery-port: `FulfillmentError`
- output destination: `patchbay:/Fulfillment/ UpdateInventoryErrorNotification`
- output destination: `patchbay:/Fulfillment/DeadMessageQueue`

The general fulfillment and inventory-related error notification destinations are configured with respective sinks that display an error message in the Oracle ATG Web Commerce log file.

Fulfillment-related error notification

- message sink: `/atg/commerce/fulfillment/FulfillmentErrorSink`
- input-destination: `patchbay:/Fulfillment/ErrorNotification`

Inventory-related error notification

- message sink: `/atg/commerce/fulfillment/UpdateInventoryErrorSink`
- input destination: `/Fulfillment/UpdateInventoryErrorNotification`

Replacing the Default Fulfillment System

You can replace the fulfillment system that ships with Oracle ATG Web Commerce with another fulfillment system. For example, if you wanted to use a test fulfillment system with or in place of the existing fulfillment system.

Follow these steps to replace the default Commerce fulfillment system:

1. Configure your new fulfillment system within Patchbay (`/atg/dynamo/messaging/dynamoMessagingSystem.xml`) to subscribe to the `sqldms://Fulfillment/SubmitOrder` topic. This is where the Purchase Process sends messages. For more information on Patch Bay, see the *Dynamo Message System* chapter of the *ATG Platform Programming GuideATG Platform Programming Guide*.
2. If your test fulfillment system is using a separate repository for orders, configure a new `OrderManager` with a new `OrderRepository`. Use this from your test fulfiller. For more information on `OrderManager`, see the *Configuring Purchase Process Services* (page 263) chapter.

3. If you want the Scenario Server to perform actions based on fulfillment events, configure Patchbay so that your fulfillment systems sends the events and the `ScenarioManager` listens for them. For more information on scenarios, see the Using Scenarios in the Fulfillment Process (page 440) section.

Integrating the Order Fulfillment Framework with an External Shipping System

The Order Fulfillment Framework can be integrated with an external shipping system that actually ships the order to the customer.

In the Oracle ATG Web Commerce default configuration, the `HardGoodShipper` simulates the shipping process. There is also a mechanism for notifying fulfillment of shipment by hand, using the Fulfillment Administration pages. See the *ATG Commerce Fulfillment Administration* chapter of the *ATG Commerce Guide to Setting Up a Store* for more information on the Fulfillment Administration pages.

An external system can be integrated with a warehouse or with a shipment company such as Federal Express. These systems are responsible actually tracking the packing and shipping of the items. There are a two ways that an external system can be integrated with the existing Order Fulfillment Framework.

- Create a JMS Message Sink and Message Source that communicates with Oracle ATG Web Commerce Order Fulfillment Framework through JMS messages. This approach provides a simple integration point. The JMS Message Sink can be registered to receive `ModifyOrderNotification` messages. When the shipping group state changes to `PENDING_SHIPMENT`, a `ModifyOrderNotification` message is sent. The new class can then communicate with the external shipping system through some other mechanism. The Order Fulfillment Framework can indicate that a shipping group has been shipped in either of the following three ways:
  - Change the state of the shipping group to NO_PENDING_ACTION and send a `ModifyOrderNotification` to notify the rest of Order Fulfillment Framework about the shipment.
  - Call `HardgoodFulfiller.shippingGroupHasShipped`
  - Call the `shippingGroupHasShipped` pipeline. For more information, see Appendix F, *Pipeline Chains* (page 699).
- Extend the `HardgoodShipper` scheduled service. Extend `HardgoodShipper.shipShippingGroup` to communicate with the external system. When the external shipping system says that a shipping group has shipped, then call `HardgoodFulfiller.shipShippingGroup`.

  By default, there is no integration with an external shipping system. The `HardgoodShipper` is designed to query the database for all "shippable" shipping groups (groups with a state of PENDING_SHIPMENT and with a `shipOnDate` that is not in the future). It then calls `HardgoodFulfiller.shippingGroupHasShipped`.

Changing Payment Behavior in Fulfillment Server

Payment in the fulfillment framework can be handled in two ways. The value of `OrderFulfiller.settleOnFirstShipment` determines how payment is handled:

- If `settleOnFirstShipment` is true, all of the payment groups are settled after the first shipping group ships.

- If `settleOnFirstShipment` is false, all of the payment groups are settled after the last shipping group ships.

You can also configure the fulfillment framework to charge each shipping group to a different credit card at the time of shipment. There are two methods in the system that can be overwritten to facilitate this behavior:

- `OrderFulfillmentTools.isOrderSettleable`

  This method is called each time the `OrderFulfiller` is notified that a shipping group has been shipped. (This notification occurs through a `ModifyOrderNotification` message that contains a `GenericUpdate` on the state of the shipping group. The new value is `PENDING_SHIPMENT`.)

  After the `OrderFulfiller` receives the notification, it calls `OrderFulfillmentTools.isOrderSettleable`. If this method returns true, then `OrderFulfiller.settleOrder` is called. To configure the fulfillment framework to charge each shipping group to a different credit card at the time of shipment, have this method check if there is a `PaymentGroup` specific to this shipping group. If there is, return true. It should also return true if all `ShippingGroups` have shipped.

- `OrderFulfiller.settleOrder`

  By default, this method settles all `PaymentGroups` in the order. To pay for each shipping group as it ships, this method should settle each `PaymentGroup` that is not `SETTLED` and refers to a `ShippingGroup` that has shipped.

  This method should also check if all shipping groups have shipped. If they have, than any other `PaymentGroups` that have not settled, should be settled. Settling a `PaymentGroup` most likely will involve using the `PaymentManager`.

To facilitate payment by shipping group as each shipping group is shipped, your system must create the appropriate `PaymentGroupRelationship` objects. To pay for each shipping group separately (and the items that appear in each shipping group), you will probably create each of these `PaymentGroupRelationship` objects yourself. When a user creates an order, the default behavior is for the order to include one `PaymentGroup` with one `PaymentGroupOrderRelationship` that accounts for the cost of the entire order. You might want to create a `PaymentGroupCommerceItemRelationship` for each item in the order, and a `PaymentGroupShippingGroupRelationship` for each shipping group in the order. It depends on your business rules.

For more information on payment groups, see the Creating Payment Groups (page 288) section. For more information on the payment manager, see the Processing Payment of Orders (page 300) section.

Using Scenarios in the Fulfillment Process

Scenarios allow business users to define the set of actions to be performed when certain events or conditions occur. A detailed discussion of how scenarios are created and used can be found in the *ATG Personalization Guide for Business Users*.

The fulfillment process uses of the scenario engine to provide features including e-mail notifications to customers. These e-mails can indicate that a shipping group has shipped, an item was backordered/preordered or whether the order is complete.

The scenario engine listens for the three different classes of messages described in the Order Fulfillment Events (page 436) section. The default Oracle ATG Web Commerce implementation uses several of these messages as the events that trigger e-mail.

The following list describes the `Fulfillment` and `ReceiveOrder` scenarios included with Commerce. These scenarios are located in the Scenarios section of the ACC in the DCS folder. For a description of the events, see the previous section. By default, this scenario is disabled. Enable this scenario using the ATG Control Center.

`Fulfillment` scenario:

- `OrderFinished` – When an `OrderModified` event with a `subType` of `OrderFinished` is received, send an e-mail to the owner of the order (the Profile that is included in the message). The e-mail uses the e-mail template found at `dynamo/commerce/en/email_templates/jsp/OrderFinished.jsp`. It notifies the customer that their order has shipped and payment has been made.
- `ShippingGroupShipped` – When an `ShippingGroupModified` event with a `subType` of `ShippingGroupShipped` is received, send an e-mail to the owner of the order. The e-mail uses the e-mail template found at `dynamo/commerce/en/email_templates/jsp/ShippingGroupShipped.jsp`. It notifies the customer that a shipping group in their order has been shipped.
- `UnavailableItems`: When an `OrderModified` event with a `subType` of `OrderHasUnavailableItems` is received, send an e-mail to the owner of the order. The e-mail uses the e-mail template found at `dynamo/commerce/en/email_templates/jsp/OrderHasUnavailableItems.jsp`. It notifies the customer that their order cannot be completed yet.
- `OrderCancelled`: When an `OrderModified` event with a subtype of `OrderWasRemoved` is received, then send an e-mail to the owner of the order. The e-mail uses the template found at `dynamo/commerce/en/email_templates/jsp/OrderCancelled.jsp`. It notifies the customer that their order has been cancelled.
- `ItemRemoved`: When an `ItemRemovedFromOrder` message is received and the order's state is 0, send and e-mail to the send an e-mail to the owner of the order. The e-mail uses the e-mail template found at `dynamo/commerce/en/email_templates/jsp/ItemRemovedFomOrder.jsp`. It notifies the customer that the item has been removed from the order.
- `PaymentGroupChanged`: When a `PaymentGroupModified` message is received, send and e-mail to the send an e-mail to the owner of the order. The e-mail uses the e-mail template found at `dynamo/commerce/en/email_templates/jsp/PaymentGroupChanged.jsp`. It notifies the customer that their payment information has been updated.

`ReceiveOrder` scenario:

- `SubmitOrder` – When a `SubmitOrder` message is received, send an e-mail to the owner of the order. The e-mail uses the e-mail template found at `dynamo/commerce/en/email_templates/jsp/SubmitOrderReceived.jsp`. It notifies the customer that their order has been received.

Questions & Answers

The following section answers some commonly asked questions about the fulfillment framework.

- Question: Why does Oracle ATG Web Commerce only use durable topics? Aren't some messages sent to only one component that might be listening?

  Answer: Topics are used because various subsystems might be interested in the message being sent. Examples of this include the `ModifyOrderNotification` message. It can be sent by any of the components in the system. The `OrderFulfiller` and the `OrderChangeHandler` components both listen for this message but each does something different with it. `OrderFulfiller` might determine that it now has control of the shipping group whose modifications are included in the `ModifyOrderNotification`

message. The `OrderChangeHandler` might choose to send some other message as an event into another subsystem.

- Question: Where do we deal with Payment groups? When do we charge?

  Answer: The `OrderFulfiller` module handles payment groups. The default implementation charges the whole order either at the time of the order's first shipment or at the time of the order's last shipment. This is configurable by changing the state of the `SettleOnFirstShipment` property in the `OrderFulfiller` to true or false as is needed by the business rules.

- Question: Why does Commerce use Java Messaging Service (JMS)?

  Answer: JMS allows you to build a distributed system that enables disparate subsystems to handle fulfillment for various parts of the fulfillment process. JMS and messaging allows you to abstract out all the connections and gives you the flexibility to adapt your existing systems to the Commerce system. For example, the `OrderFulfiller` system might be located on the same set of machines in the site hosting facility. The `HardgoodFulfiller` might be based in some other set of headquarters. The actual warehouse that does the shipping might be in another location. If the warehouse receives an e-mail when a shipping group is submitted, then a service can listen on the JMS message indicating that the shipping group is to be shipped. An e-mail can be constructed from the contents of the message.

- Question: What are modification objects? What purpose do they serve?

  Answer: The Modification object is an abstraction that tries to capture the various ways in which changes can occur to the order. There are several types of modifications: add, remove, or update. Modifications can be targeted and therefore can modify shipping groups, payment groups, orders or relationships. Modifications contain a status field indicating whether the modification was successful or a failure.

  This abstraction allows the system the flexibility to interface with existing legacy or distributed systems. A disparate system can construct an array of Modifications that will capture the types of changes that it is requesting or the modifications it already performed.

  Refer to the sections on ModifyOrder Class (page 418) and ModifyOrderNotification Class (page 419) for more information.

- Question: How do scenarios find out about what is going on in the fulfillment system?

  The scenario engine receives messages that are sent during the fulfillment process. Those messages are documented in the Inventory JMS Messages (page 406) section.

  The events contain the profile and the information needed for performing an action on those events. For example, the `ShippingGroupShipped` event contains the profile, the order and the shipping group that was shipped. This allows the scenario writer to create an action that sends an e-mail to the user (the profile) with the order information (from the order) and the details of the shipping group that was shipped (the shipping group). For more information, refer to the Order Fulfillment Events (page 436) section.

- Question: How do I change the behavior of `ModifyOrder` messages?

  The `ModificationHandler` class deals with all the `ModifyOrder` messages. Both the `OrderFulfiller` and the `HardgoodFulfiller` have their own versions of those handler classes called `OrderFulfillerModificationHandler` and the `HardgoodFulfillerModificationHandler`. The class contains two methods `handleModifyOrder` and `handleModifyOrderNotification`.

  To change the behavior of one of those two handling methods, override the method if you extended the existing `OrderFulfillerModificationHandler` or `HardgoodFulfillerModificationHandler` classes. Otherwise, a new class implementing the `ModificationHandler` interface should be written and configured for the `OrderFulfiller` or `HardgoodFulfiller`.

- Question: How do I change the behavior of `ModifyOrderNotification` messages?

  Answer: See the answer for changing the handling of the `ModifyOrder` message in the previous question.

- Question: How do we deal with the Modification IDs? Who is generating them? How do we guarantee the uniqueness?

  Answer: Modification IDs are generated using the ID generator. The combination of the message source and message ID need to be unique to allow external systems to track the various messages in the system.

20 Managing the Order Approval Process

Oracle ATG Web Commerce enables you to implement an order approval process for your application. The approval process can identify the customers for whom order approvals are required and for specific conditions that trigger an approval requirement, such as when an order limit is exceeded.

This chapter describes the default implementation of the order approval process provided with Commerce and highlights the areas you're likely to customize to meet your application's needs. It includes the following sections:

Understanding the Order Approval Process

At its most basic level, the order approval process consists of the following phases:

1. During checkout, the application determines if an order requires approval.
2. If the order requires approval, then an approver approves or rejects the order.
3. The application determines if the approval process for the order is complete.
4. If the approval process for the order is complete, then the order proceeds through checkout.

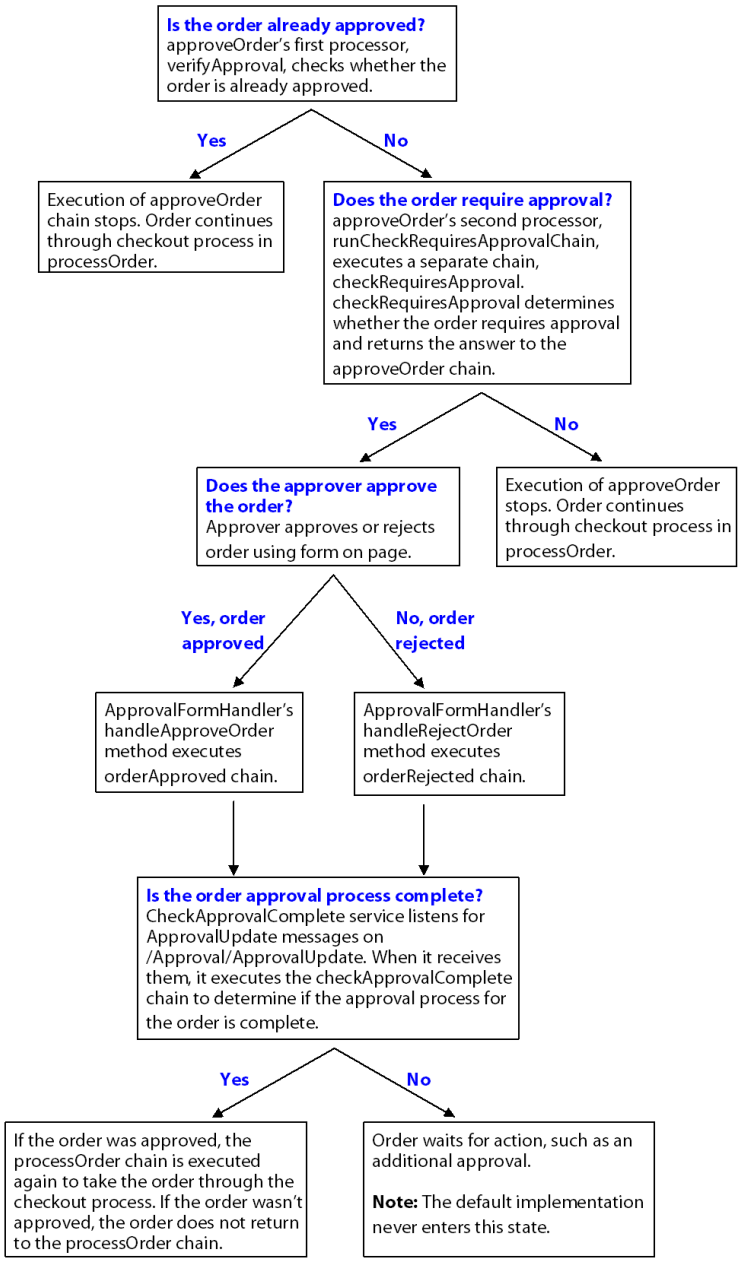
The following properties of an Order object (class `atg..order.Order`) support the order approval process and maintain historical approval data for an order as it moves through these phases.

Property Name	Property Type	Description
`authorizedApproverIds`	List	The list of profile IDs of the users who are authorized to approve the given order.

Property Name	Property Type	Description
`approvalSystemMessages`	List	The list of system messages that the application attaches to the order. These messages indicate what conditions triggered an approval being required, such as "order limit exceeded." They are defined by the processors in the `checkRequiresApproval` pipeline chain.
`approverIds`	List	The list of one or more profile IDs of the users who have approved or rejected the order.
`approverMessages`	List	The list of messages the approvers attach to the order. The `ApprovalFormHandler` form handler enables an approver to add comments when he or she approves or rejects an order. These comments are added to this property.

The following diagram illustrates the business logic that drives the order approval process and briefly indicates which components move an order from one step in the process to the next. The complete order approval process is subsequently explained in detail.

In this diagram, a customer has just submitted an order for checkout, and, consequently, the `processOrder()` method has executed the `processOrder` pipeline chain. `processOrder`'s first processor, `executeValidateForCheckoutChain`, has validated the order. `processOrder`'s second processor, `executeApproveOrderChain`, begins the order approval process by executing the `approveOrder` chain. The first processor in the `approveOrder` chain is `verifyApproval`.



As is briefly illustrated in the preceding diagram, the order approval process is as follows:

1. `approveOrder`'s first processor, `verifyApproval`, checks whether the order already is approved.
   - If the order is approved, then execution of `approveOrder` stops, and the order proceeds through the checkout process in `processOrder`.
   - If the order isn't approved, then the order proceeds to the second processor in `approveOrder`.
2. `approveOrder`'s second processor, `runCheckRequiresApprovalChain`, executes a separate pipeline chain named `checkRequiresApproval`. `checkRequiresApproval` determines whether approval is needed for the order and reports back to `approveOrder`.

- If approval is needed, then the order continues through the `approveOrder` chain. `approveOrder` changes the order's state to PENDING_APPROVAL. The chain adds to the order's `authorizedApproverIds` property the list of profile IDs of the users who are allowed to approve the order, and it adds to the order's `approvalSystemMessages` property the conditions that triggered an approval to be required. Finally, `approveOrder` sends out an `ApprovalRequired` message to the `/Approval/Scenarios` JMS message topic. This message can then be used to execute scenarios.

- If approval isn't needed, then execution of `approveOrder` stops, and the order proceeds through the checkout process in `processOrder`. (For detailed information on the `processOrder` pipeline, see Checking Out an Order (page 294) in the *Checking Out Orders* section of the *Configuring Purchase Process Services* chapter.)

Note: The default implementation of the `checkRequiresApproval` chain checks the `approvalRequired` property in the customer's profile. If the `approvalRequired` property is true, then approval is required for the customer. An error is then added to the `PipelineResult` object, which tells the system that an approval is required, and the reason that approval is required is stored in the `errorMessages` property of the Order. This reason for approval is later added to the order's `approvalSystemMessages` property by the `approveOrder` chain's `addApprovalSystemMessagesToOrder` processor. If the `approvalRequired` property is false, then approval isn't required for the customer.

You can edit the `checkRequiresApproval` chain to create specific requirements for whether an approval is required for a given customer. For example, you might want to include a processor that checks the total amount of the customer's order against an order limit specified in the customer's profile. If the order amount exceeds the specified limit, then approval for the customer's order would be required. Similarly, you might want to include a processor that checks the manufacturers of the items in the customer's order against a list of preferred suppliers specified in the customer's profile. If a manufacturer isn't in the list of preferred suppliers, then approval for the customer's order would be required.

3. If approval for the order is needed, an approver then approves or rejects the order and submits this decision in a form using the `ApprovalFormHandler` form handler. The form's "Approve" and "Reject" submit buttons call the `handleApproveOrder` method or the `handleRejectOrder` method, respectively.

   - If the `handleApproveOrder` method is called, the method executes the `orderApproved` pipeline chain. `orderApproved` adds the profile ID of the user who approved the order to the order's `approverIds` property, and adds the messages associated with the approver's decision to the order's `approverMessages` property. Finally, `orderApproved` sends out an `ApprovalUpdate` message whose `approvalStatus` property is set to "approved." This message is sent to both the `/Approval/ApprovalUpdate` JMS message queue and the `/Approval/Scenarios` JMS message topic. This message can then be used to execute scenarios.

   - If the `handleRejectOrder` method is called, the method executes the `orderRejected` pipeline chain. `orderRejected` adds the profile ID of the user who rejected the order to the order's `approverIds` property, and adds the messages associated with the approver's decision to the order's `approverMessages` property. Finally, `orderRejected` sends out an `ApprovalUpdate` message whose `approvalStatus` property is set to "rejected." This message is sent to both the `/Approval/ApprovalUpdate` JMS message queue and the `/Approval/Scenarios` JMS message topic. This message can then be used to execute scenarios.

   Note: The `orderApproved` and `orderRejected` pipeline chains are the same with the exception of the value of `approvalStatus` property of the `ApprovalUpdate` message that is sent. You can edit these chains to add or remove functionality to meet your application's needs.

4. The `ApprovalCompleteService`, located in Nucleus at `/atg/commerce/approval/ApprovalCompleteService`, is configured to listen for the `ApprovalUpdate` messages sent by the `orderApproved` and `orderRejected` chains to the `/Approval/ApprovalUpdate` JMS message queue.

When ApprovalCompleteService receives a message, it executes the checkApprovalComplete pipeline chain. checkApprovalComplete determines whether the approval process for the order is complete. By default, checkApprovalComplete's second processor, approvalCompleteAnalyzer, checks whether at least one person has approved or rejected the order. If so, then the approval process for the order is considered to be complete.

- If at least one person has *approved* the order, checkApprovalComplete changes the order's state from PENDING_APPROVAL to APPROVED and executes the processOrder chain so the order can go through the checkout process. (For detailed information on the processOrder pipeline, see Checking Out an Order (page 294) in the *Checking Out Orders* section of the *Configuring Purchase Process Services* chapter.) Additionally, the checkApprovalComplete chain sends an ApprovalComplete message whose approvalStatus property is set to "approval_passed" to the /Approval/Scenarios JMS message topic. This message can then be used to execute scenarios.
- If at least one person has *rejected* the order, checkApprovalComplete changes the order's state from PENDING_APPROVAL to FAILED_APPROVAL, and sends an ApprovalComplete message whose approvalStatus property is set to "approval_failed" to the /Approval/Scenarios JMS message topic. This message can then be used to execute scenarios.

Note: As previously mentioned, the default implementation of the approvalCompleteAnalyzer processor merely checks whether at least one person has approved or rejected the order. If so, then the approval process for the order is considered completed. You can change the implementation of approvalCompleteAnalyzer in order to change the requirements for completion of the approval process.

For detailed information about the pipeline chains and processors mentioned in this section, refer to Appendix F, *Pipeline Chains* (page 699).

Modifying the Order Approval Process

The ApprovalPipelineManager uses an approval pipeline configuration file to manage the pipelines in the order approval process. The configuration file is located at <ATG10dir>/DCS/atg/commerce/approval/approvalpipeline.xml.

If you modify the pipelines in the order approval process, you'll need to extend the approvalpipeline.xml file to override the default configuration. To do so, create a new approvalpipeline.xml file at /atg/commerce/approval/ in your config directory. During deployment, the Oracle ATG Web Commerce platform uses XML file combination to combine the approvalpipeline.xml files in the CONFIGPATH into a single composite XML file.

For general information about how to modify existing pipelines, see the *Processor Chains and the Pipeline Manager* (page 363) chapter. For more information on XML file combination, see the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Platform Programming Guide*.

Servlet Beans and Form Handlers for Approving Orders

When an order has been determined to require approval, and its order state has been set to PENDING_APPROVAL, an approver must then review the order and approve or reject it. ApprovalRequiredDroplet and ApprovalFormHandler are provided for this purpose.

Additionally, an approver might want to view a historical list of the orders he or she has approved and/or rejected. ApprovedDroplet is provided for this purpose.

ApprovalRequiredDroplet Servlet Bean

Use the `ApprovalRequiredDroplet` servlet bean (class `atg..approval.ApprovalRequiredDroplet`) to retrieve all orders requiring approval by a given approver. `ApprovalRequiredDroplet` queries the order repository and returns all orders that meet the following two criteria:

- The order's `authorizedApproverIds` property contains the approver's ID.
- The state of the order requires approval, meaning that the state is defined in the `ApprovalRequiredDroplet` `orderStatesRequiringApproval` property. The order's state is held by the property of the order that is specified in the `ApprovalRequireDroplet` `orderStatePropertyName` property. The default value is `PENDING_APPROVAL`.

Refer to *Appendix: ATG Commerce Servlet Beans* of the *ATG Commerce Guide to Setting Up a Store* for detailed information about the input, output, and open parameters of `ApprovalRequiredDroplet`, as well as a JSP code example using this servlet bean.

ApprovedDroplet Servlet Bean

Use the `ApprovedDroplet` servlet bean (class `atg..approval.ApprovedDroplet`) to retrieve all orders that have been approved and/or rejected by a given approver. `ApprovedDroplet` queries the order repository and returns all orders that have the approver's profile ID in the `approverIds` property.

Refer to *Appendix: ATG Commerce Servlet Beans* of the *ATG Commerce Guide to Setting Up a Store* for detailed information about the input, output, and open parameters of `ApprovedDroplet`, as well as a JSP code example using this servlet bean.

ApprovalFormHandler

The `ApprovalFormHandler` form handler (class `atg..approval.ApprovalFormHandler`) processes an approver's approval or rejection of an order. The `ApprovalFormHandler` class contains two handle methods, `handleApproveOrder` and `handleRejectOrder`. You can associate these handle methods with Submit properties in the following manner:

```
<input type=submit bean="ApprovalFormHandler.approveOrder" value=" Approve Order">
<input type=submit bean="ApprovalFormHandler.rejectOrder" value=" Reject Order">
```

If the `handleApproveOrder` method is called for `ApprovalFormHandler.approveOrder`, the `handleApproveOrder` method executes the `orderApproved` pipeline chain. Similarly, if the `handleRejectOrder` method is called for `ApprovalFormHandler.rejectOrder`, the `handleRejectOrder` method executes the `orderRejected` pipeline chain.

Refer to *Implementing an Order Approval Process* chapter of the *ATG Commerce Guide to Setting Up a Store* for a JSP code example that uses `ApprovalFormHandler`. Refer to Appendix F, *Pipeline Chains* (page 699) for more information about the `orderApproved` and `orderRejected` chains.

JMS Messages in the Order Approval Process

The following JMS messagessupport the order approval process:

- `ApprovalRequiredMessage`
  JMS Type: `atg.commerce.approval.ApprovalRequired`

  Extends the class `atg.commerce.messaging.CommerceMessageImpl`.

  This message includes the order requiring approval and the profile repository item for the customer associated with the order as `order` and `profile` message properties, respectively. If you are using multisite, the `siteId` property is populated as well.

  This message is sent to the `/Approval/Scenarios` JMS message topic, which allows the scenario server to interact with the message using SQL JMS and obtain the profile outside the current thread.

- `ApprovalMessage`
  JMS Type: `atg.commerce.approval.ApprovalUpdate`
  `atg.commerce.approval.ApprovalComplete`

  Extends the class `atg.commerce.messaging.CommerceMessageImpl`.

  This message includes the order requiring approval and the profile repository item for the customer associated with the order as `order` and `profile` message properties, respectively. It also includes an `approvalStatus` message property.

  If you are using multisite, the `siteId` property is populated as well.

  A message with a `JMSType` of either `ApprovalUpdate` or `ApprovalComplete` is sent to the `/Approval/Scenarios` JMS message topic, which allows the scenario server to interact with the message using SQL JMS and obtain the profile outside the current thread.

  A message with a `JMSType` of `ApprovalUpdate` is *also* sent to the `/Approval/ApprovalUpdate` JMS message queue. `ApprovalCompleteService` listens for messages on this queue.

21 Using Abandoned Order Services

An abandoned order or shopping cart is one that a customer creates and adds items to, but never checks out. Instead, the customer simply exits the Web site, thus "abandoning" the incomplete order.

The Abandoned Order Services module that is provided with Oracle ATG Web Commerce includes a collection of services and tools that enable you to detect, respond to, and report on abandoned orders and related activity. As such, it enables you to better understand what kinds of orders your customers are abandoning, as well as what campaigns effectively entice them to reclaim and complete them. The result is an increase in order conversion and revenue.

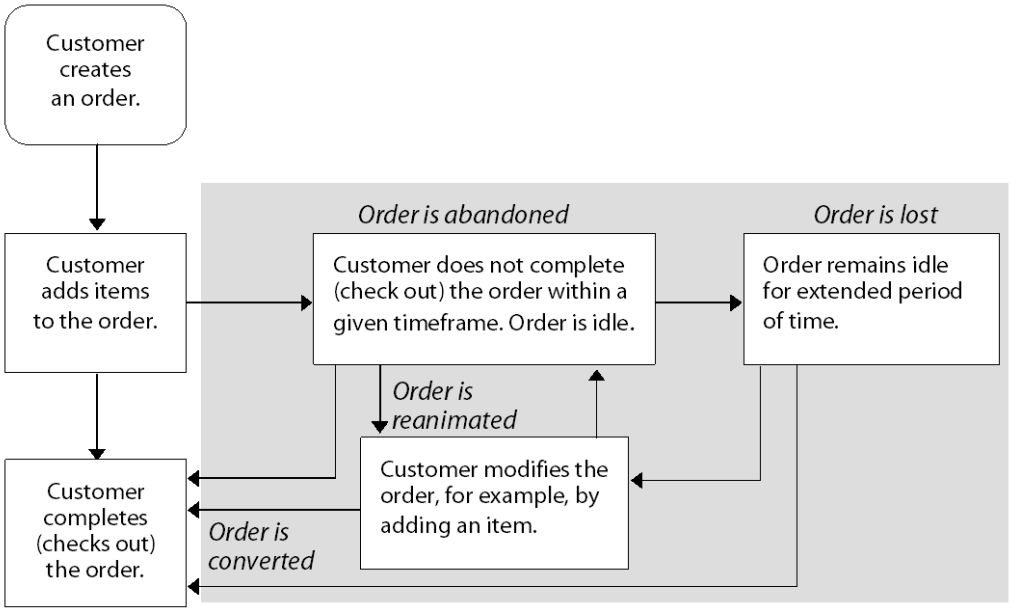
This chapter is intended for developers who must configure the module according to specific Web site requirements. It includes the following sections:

Important: When you want to work with the Abandoned Order Services module, you need to include the `DCS.AbandonedOrderServices` module when you assemble your application. See the *ATG Platform Programming GuideATG Platform Programming Guide* for information on modules and application assembly.

For information on related tasks that are typically performed by merchants and other business users, such as creating scenarios that respond to abandonment activity, see the *Managing Abandoned Orders* chapter of the *ATG Commerce Guide to Setting Up a Store*.

An Overview of Abandoned Orders

Examine the following process flow diagram, which illustrates the various paths an order can take once created by a customer.



As mentioned in the introduction to this chapter, the Abandoned Order Services module contains a collection of services and tools that enable you to detect, respond to, and report on abandoned orders and related activity, that is, activity that falls within the shaded area of the diagram above. As the diagram implies, there are several general types of orders that fall within this area:

- **Abandoned orders** – Incomplete orders that have not been checked out by customers and instead have remained idle for a duration of time.
- **Reanimated orders** – Previously abandoned orders that have since been modified by the customer in some way, such as adding items or changing item quantities.
- **Converted orders** – Previously abandoned orders that have been successfully checked out by the customer.
- **Lost orders** – Abandoned orders that have been abandoned for so long that reanimation of the order is no longer considered realistic.

Note in the diagram that the process flow is not always linear. For example, an order can be abandoned, then reanimated, then abandoned again.

The subsections that follow describe the various abandonment states, repository extensions, and repositories that are required to support these orders and the tracking of related order abandonment activity:

- Abandonment States (page 454)
- Order Repository Extensions (page 455)
- Profile Repository Extensions (page 63)
- The AbandonedOrderLogRepository (page 456)

Abandonment States

The table below describes the default abandonment states in the Abandoned Order Services module:

State	Description
ABANDONED	The order is incomplete, that is, not checked out, and meets the criteria for identification as abandoned. Typically, an abandoned order is one that has been idle for a specific number of days. You define the criteria for abandoned orders in the `AbandonedOrderService` component; see Defining and Detecting Abandoned Orders (page 457).
REANIMATED	The order was previously abandoned or lost but has since been modified by the customer in some way, for example, by changing item quantities.
CONVERTED	The order was previously abandoned or lost but has since been checked out by the customer.
LOST	An abandoned order that meets the criteria for identification as lost. Typically, a lost order is one that has been idle for a specific number of days. You define the criteria for lost orders in the `AbandonedOrderService` component; see Defining and Detecting Abandoned Orders (page 457).

Order Repository Extensions

The Abandoned Order Services module extends the repository definition for the order repository by adding:

- an additional property named `abandonmentInfo` to the `order` item descriptor. This property stores an item of type `abandonmentInfo`.
- an additional item descriptor named `abandonmentInfo`. This item stores the abandonment information for an order.

  The following table describes each `abandonmentInfo` property:

Property	Description
`version`	An integer that indicates the number of times the item has been modified.
`order`	The `order` item associated with this `abandonmentInfo` item.
`orderId`	The ID of the order.
`orderLastUpdated`	The date the order was most recently modified. This property is used to detect activity on abandoned orders. It is important to distinguish this property from the `lastModifiedDate` order property, which is updated whenever a session is created for a user who has an incomplete order associated with his or her profile. The `lastModifiedDate` order property cannot be used to accurately detect abandoned order activity (or, more specifically, lack thereof) because it is updated even when a user has not accessed an incomplete order.

Property	Description
`state`	The order's current abandonment state. For a list of possible states, see Abandonment States (page 454) above.
`abandonmentCount`	The number of times the order has been identified as ABANDONED. Because an order can be abandoned multiple times, this count can be greater than one.
`abandonmentDate`	The date and time that the order was most recently abandoned.
`reanimationDate`	The date and time that the order was most recently reanimated.
`conversionDate`	The date and time that the order was converted, that is, checked out successfully.
`lostDate`	The date and time that the order was most recently lost.

For more information, see the definition file at `<ATG10dir>/DCS/AbandonedOrderServices/config/atg/commerce/order/orderrepository.xml`.

Profile Repository Extensions

The Abandoned Order Services module extends the repository definition for the profile repository by adding:

- an additional item descriptor named `abandoned-order`. Items of this type have two properties:
  - `orderId`, which stores the ID of the abandoned order.
  - `profileId`, which stores the ID of the user profile associated with the abandoned order.
- two additional properties to the `user` item descriptor:
  - `abandonedOrders`, which stores the list of `abandoned-order` items currently associated with the user.
  - `abandonedOrderCount`, which is a derived property that stores the number of items in the `abandonedOrders` user property.

For more information, see the definition file at `<ATG10dir>/DCS/AbandonedOrderServices/config/atg/userprofiling/userProfile.xml`.

The AbandonedOrderLogRepository

The Abandoned Order Services module defines an `AbandonedOrderLogRepository` that stores information about converted orders. Converted orders are previously abandoned orders that subsequently have been checked out.

The `AbandonedOrderLogRepository` is located in Nucleus at `/atg/commerce/order/abandoned/AbandonedOrderLogRepository`. It defines a single item descriptor named `convertedOrder` with the following properties:

Property	Description
`orderId`	The ID of the converted order.
`convertedDate`	The date and time that the order was converted.
`amount`	The total price of the converted order.
`promotionCount`	The number of promotions that were applied to the converted order.
`promotionValue`	The total value of the promotions that were applied to the converted order.

When an abandoned order is checked out by a user and, therefore, is identified as converted, the Log Promotion Info scenario action in the Abandoned Orders scenario calculates the number and total value of the promotions applied to the converted order and stores the information in a `convertedOrder` item in the `AbandonedOrderLogRepository`.

The data in the `AbandonedOrderLogRepository` is particularly important for reporting on abandonment activity.

Defining and Detecting Abandoned Orders

The Abandoned Order Services module provides mechanisms for defining and detecting abandoned and lost orders. See the subsections that follow for details:

- Defining Abandoned and Lost Orders (page 457)
- Detecting Abandoned and Lost Orders (page 458)

For information on using scenarios to detect order abandonment activity, see the *Managing Abandoned Orders* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Defining Abandoned and Lost Orders

By default, you can define what constitutes an abandoned and lost order using the following criteria:

- number of idle days
- minimum amount (optional)

You set these criteria for abandoned and lost orders in the following properties of the `/atg/commerce/order/abandoned/AbandonedOrderService` component:

- `idleDaysUntilAbandoned`
- `idleDaysUntilLost`
- `minimumAmount`

For the default values of these properties, see Configuring AbandonedOrderService (page 459) later in this chapter. Note that an amount specified in the `AbandonedOrderService.minimumAmount` property is used as a criterion when detecting both abandoned and lost orders.

You may want to define different types of abandoned or lost orders. For example, you may want to differentiate between high-priced and low-priced abandoned orders in order to respond differently to each type. For information on this type of customization, see Customizations and Extensions (page 473).

Detecting Abandoned and Lost Orders

The `/atg/commerce/order/abandoned/AbandonedOrderService` not only defines what constitutes an abandoned or lost order, but also queries the order repository for these types of orders according to the schedule that you specify in its `schedule` property. The default schedule is "every day at 3:00 AM."

When an `AbandonedOrderService` job is run, the service queries the order repository for both abandoned and lost orders. The following table lists the criteria orders must meet to be identified as abandoned or lost:

Criteria for Identification as "Abandoned"	Criteria for Identification as "Lost"
The order's state matches one in the `AbandonedOrderTools.abandonableOrderStates` property.	Same
The abandonment state is REANIMATED.	The abandonment state is not LOST.
The order has been idle for the number of days specified in the `AbandonedOrderService.idleDaysUntilAbandoned` property.	The order has been idle for the number of days specified in the `AbandonedOrderService.idleDaysUntilLost` property.
The order's subtotal is greater than or equal to the amount specified in the `AbandonedOrderService.minimumAmount` property, if set.	Same

See Configuring AbandonedOrderTools (page 461) and Configuring AbandonedOrderService (page 459) for information on setting the properties referenced in the table above.

For each *abandoned* order found, the `AbandonedOrderService` does the following:

1. Adds the order to the list of abandoned orders in the user's `abandonedOrders` profile property.
2. If necessary, creates an `abandonmentInfo` item for the order; then updates the item with the relevant information:
   - The `state` property is set to ABANDONED.
   - The `abandonmentDate` property is set to the current date and time.
   - If the `abandonmentInfo` item is new, the `abandonmentCount` property is set to 1. Otherwise, it is incremented.

3. Fires an `OrderAbandoned` message if the `AbandonedOrderTools.sendOrderAbandonedMessage` property is set to `true`.

For each *lost* order found, the `AbandonedOrderService` does the following:

1. Removes the order from the list of abandoned orders in the user's `abandonedOrders` profile property.

2. If the `AbandonedOrderTools.deleteLostOrders` property is set to `true`, the lost order is deleted from the order repository.

3. If the `AbandonedOrderTools.leaveAbandonmentInfoForDeletedOrders` property is set to `true`, the `abandonmentInfo` item for the order is updated with the relevant information:

   - The `state` property is set to LOST.

   - The `lostDate` property is set to the current date and time.

4. Fires an `OrderLost` message if the `AbandonedOrderTools.sendOrderLostMessage` property is set to `true`.

As previously mentioned, the `AbandonedOrderService` is a configured instance of class `atg.commerce.order.abandoned.AbandonedOrderService`. This class extends `atg.service.scheduler.SingletonSchedulableService`, which uses locking to enable multiple servers to run the same scheduled service while ensuring that only one instance performs the scheduled task at a given time.

Configuring AbandonedOrderService

The `/atg/commerce/order/abandoned/AbandonedOrderService` is a schedulable service that has two important functions:

- Storing the criteria that orders must meet to be identified as abandoned and lost.

- Querying the order repository on a specific schedule for orders to identify as abandoned or lost.

The following table describes each `AbandonedOrderService` property you may want to configure:

Property	Description	Default value
`orderStatePropertyName`	The `order` property that stores the state of the order.	`state`
`dateQueryPropertyName`	The `order` property to use when determining how long an order has been idle.	`abandonmentInfo.orderLastUpdated`
`idleDaysUntilAbandoned`	The number of days that an order must be idle for it to be considered abandoned.	7

Property	Description	Default value
`idleDaysUntilLost`	The number of days that an order must be idle for it to be considered lost. If `AbandonedOrderTools.deleteLostOrders` is set to `true`, you can use the `processLostOrders` method of `AbandonedOrderService` to delete such orders.	30
`minimumAmount`	The minimum amount that an order must cost for it to be considered abandoned **or** lost (for example, "10.00"). Set this property to zero or leave it unset if you do not want to use order price as a criterion. **Warning:** If you do use `minimumAmount` as a criterion for abandoned orders, you **must** set the `orderStateSaveModes` for INCOMPLETE to a value other than NONE (the default setting is NONE). If you do not, the query for incomplete orders will search for pricing information that is not saved, and will never return any orders. See the `updatePriceInfoObjects` pipeline processor information in Appendix F, *Pipeline Chains* (page 699) for additional information.	0
`subtotalPropertyName`	The `OrderPriceInfo` property to use when determining if an order satisfies the "minimum amount" criterion for identification as abandoned or lost. Set this property to `rawSubtotal` (the default) to use the order's price before promotions, taxes, and shipping costs are applied. Alternatively, use "amount" to use the order's price after these items are applied.	`rawSubtotal`
`priceInfoPropertyName`	The `order` property that stores the order's `OrderPriceInfo` object.	`priceInfo`

Property	Description	Default value
abandonedOrderTools	The AbandonedOrderTools helper component. (See Configuring AbandonedOrderTools (page 461) below.)	/atg/commerce/order/abandoned/AbandonedOrderTools
jobName	The name of the scheduled job to run.	AbandonedOrderService
jobDescription	A description of the scheduled job.	Identify abandoned and lost orders
schedule	The schedule by which to run AbandonedOrderService jobs.	every day at 3:00 AM
scheduler	The Scheduler service that should keep track of AbandonedOrderService jobs and call on this service to execute them.	/atg/dynamo/service/Scheduler
clientLockManager	The client lock manager that should ensure that only one instance of this service is running at a given time.	/atg/dynamo/service/ClientLockManager
lockName	The name of the global write lock that identifies this service.	AbandonedOrderService
lockTimeOut	The maximum time in milliseconds to wait for a lock. To wait indefinitely, set this property to zero.	2000
transactionManager	The TransactionManager used by this service.	/atg/dynamo/transaction/TransactionManager
maxItemsPerTransaction	The maximum number of items to include in a single transaction.	1000
numberOfBatchesPerTransaction	The number of batches of orders to fetch in a single transaction.	5
numberOfConcurrentUpdateThreads	The number of batches or orders to process at once (1 batch per thread).	5
numberOfHoursToTimeOutThreads	The maximum length of time to allow for the processing of a single batch.	1

Configuring AbandonedOrderTools

The /atg/commerce/order/abandoned/AbandonedOrderTools component stores the central configuration for the entire Abandoned Order Services module, including the definition of names for required properties,

repository items, and abandonment states. The following table describes important properties you may want to configure.

Properties that store state names	Description	Default value
`abandonableOrderStates`	The list of possible order states an order can be in to be considered for identification as abandoned or lost.	INCOMPLETE
`reanimateableAbandonmentStates`	The list of possible abandonment states that an order can be in to be considered for identification as reanimated.	ABANDONED,LOST
`defaultAbandonedState`	The abandonment state to assign to orders identified as abandoned.	ABANDONED
`defaultReanimatedState`	The abandonment state to assign to orders identified as reanimated.	REANIMATED
`defaultConvertedState`	The abandonment state to assign to orders identified as converted.	CONVERTED
`defaultLostState`	The abandonment state to assign to orders identified as lost.	LOST
`reanimatedAbandonmentStates`	The list of possible abandonment states that an order can be in to be considered reanimated. Used to identify reanimated orders that should be considered re-abandoned.	REANIMATED
`lostAbandonmentStates`	The list of possible abandonment states that an order can be in to be considered lost. Used to determine if an order has already been identified as lost.	LOST
Properties that store item names	**Description**	**Default value**

Properties that store state names	Description	Default value
orderItemName	The name of the order item descriptor in the order repository.	Reference to /atg/commerce/order/OrderTools.orderItemDescriptorName
profileItemName	The name of the profile item descriptor in the profile repository.	Reference to /atg/userprofiling/ProfileTools.defaultProfileType
abandonmentInfoItemName	The name of the item descriptor in the order repository that holds abandonment information for an order.	abandonmentInfo
abandonedOrderItemName	The name of the abandoned order item descriptor in the profile repository.	abandoned-order
Properties that store property names	**Description**	**Default value**
abandonmentInfoPropertyName	The name of the property in the order item that holds its abandonmentInfo item.	abandonmentInfo
profileIdPropertyName	The name of the property in the order item that holds the profile ID of the user that owns the order.	profileId
abandonedOrderOrderIdPropertyName	The name of the property in the abandoned-order item that holds the ID of the order.	orderId
abandonedOrderProfileIdPropertyName	The name of the property in the abandoned-order item that holds the ID of the profile that owns the order.	profileId
abandonmentStatePropertyName	The name of the property in the abandonmentInfo item that holds the abandonment state of the order.	state
orderPropertyName	The name of the property in the abandonmentInfo item that holds the order with which it is associated.	order

Properties that store state names	Description	Default value
`orderIdPropertyName`	The name of the property in the `abandonmentInfo` item that holds the ID of the associated order.	`orderId`
`abandonmentCountPropertyName`	The name of the property in the `abandonmentInfo` item that indicates the number of times the associated order has been abandoned.	`abandonmentCount`
`abandonmentDatePropertyName`	The name of the property in the `abandonmentInfo` item that holds the date and time when the order was most recently identified as abandoned.	`abandonmentDate`
`reanimationDatePropertyName`	The name of the property in the `abandonmentInfo` item that holds the date and time when the order was most recently reanimated.	`reanimationDate`
`conversionDatePropertyName`	The name of the property in the `abandonmentInfo` item that holds the date and time when the order was identified as converted.	`conversionDate`
`lostDatePropertyName`	The name of the property in the `abandonmentInfo` item that holds the date and time when the order was most recently identified as lost.	`lostDate`
`lastUpdatedPropertyName`	The name of the property in the `abandonmentInfo` item to be updated by the `SetLastUpdated` scenario action and servlet bean.	`orderLastUpdated`
Messaging-related properties	**Description**	**Default value**
`messageSender`	The component that sends abandonment-related JMS messages.	`/atg/commerce/order/abandoned/OrderAbandonedSender`

Properties that store state names	Description	Default value
`messageFactory`	The component that builds abandonment-related JMS messages. See Scenario Events and Actions (page 466) in this chapter.	`/atg/commerce/order/abandoned/AbandonedOrderMessageFactory`
`sendOrderAbandonedMessage`	boolean. True if an `OrderAbandoned` message should be sent when an order is identified as abandoned.	`true`
`sendOrderLostMessage`	boolean. True if an `AbandonedOrderLost` messages should be sent when an order is identified as lost.	`true`
`sendOrderReanimatedMessage`	boolean. True if an `AbandonedOrderReanimated` message should be sent when an abandoned order is reanimated.	`true`
`sendOrderConvertedMessage`	boolean. True if an `AbandonedOrderConverted` message should be sent when an abandoned order is converted.	`true`
Other important properties	**Description**	**Default value**
`deleteLostOrders`	boolean. True if orders identified as lost should be removed from the order repository. Orders must be in an `INCOMPLETE` state and idle for more days than configured in the `AbandonedOrderService.idleDaysUntilLost` property.	`false`
`leaveAbandonmentInfoForDeletedOrders`	boolean. True if the `abandonmentInfo` items should be retained for lost orders that are deleted. Lost orders are deleted if the `AbandonedOrderTools.deleteLostOrders` property is set to true.	`false`
`orderRepository`	The order repository in which to look for abandoned and lost orders.	`/atg/commerce/order/OrderRepository`

Properties that store state names	Description	Default value
orderManager	The OrderManager to use to delete lost orders. Note that orders identified as lost are deleted only if the AbandonedOrderTools.deleteLostOrders property is set to true.	/atg/commerce/order/OrderManager
profileRepository	The profile repository in which to edit users' lists of abandoned orders.	/atg/userprofiling/ProfileAdapterRepository

Scenario Events and Actions

This section provides technical descriptions of the abandonment-related scenario events and actions that are provided with the Abandoned Order Services module.

- Scenario Events (page 466)
- Scenario Actions (page 469)

For information on creating scenarios that include these elements, as well as information on testing them via the *Abandoned Order Messages* page of the Commerce Administration UI, see the *Managing Abandoned Orders* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Scenario Events

The Abandoned Order Services module includes the following scenario events to watch for abandonment-related events:

- Order Abandoned (page 467)
- Abandoned Order is Modified (page 467)
- Abandoned Order is Converted (page 468)
- Abandoned Order is Lost (page 469)

The message that triggers the events listed above contains the following properties:

Property	Type	Scenario editor label
abandonmentState	java.lang.String	abandonmentState
	The abandonment state of the order.	
orderId	java.lang.String	orderId
	The ID of the order.	

Property	Type	Scenario editor label
`profileId`	`java.lang.String`	`profileId`
	The profile ID of the user associated with the order.	
`type`	`java.lang.String`	`type`
	The JMS type of `OrderAbandoned` message.	

For an example scenario that utilizes some of these events, see *Creating Scenarios that Respond to Abandonment Activity* in the *Managing Abandoned Orders* chapter of the *ATG Commerce Guide to Setting Up a Store*.

Order Abandoned

This event is triggered when an order is identified as abandoned.

Class name	`atg.commerce.order.abandoned.OrderAbandoned`
JMS name	`atg.commerce.order.abandoned.OrderAbandoned`
Display name	Order Abandoned
Message context	request
Message scope	individual

Message source	Component:
	`/atg/commerce/order/abandoned/OrderAbandonedSender`
	
	Class:
	`atg.commerce.messaging.MessageSender`

Component that calls the message source	`/atg/commerce/order/abandoned/AbandonedOrderService`
How this event is triggered	Triggered when an order is identified as abandoned by the `AbandonedOrderService`.
How to turn off this event	Set the `/atg/commerce/order/abandoned/AbandonedOrderTools.sendOrderAbandonedMessage` property to `false`.

Abandoned Order is Modified

This event is triggered when an abandoned order is modified by its owner. Modifications to an order can include adding or removing items, changing item quantities, and merging orders.

Class name	`atg.commerce.order.abandoned.OrderAbandoned`

JMS name	`atg.commerce.order.abandoned.OrderReanimated`
Display name	Abandoned Order is Modified
Message context	request
Message scope	individual

Message source	Component:
	`/atg/commerce/order/abandoned/OrderAbandonedSender`
	
	Class:
	`atg.commerce.messaging.MessageSender`

Component that calls the message source	`/atg/commerce/order/abandoned/AbandonedOrderTools`
How this event is triggered	Triggered by the Reanimate Abandoned Order (page 470) scenario action or `ReanimateAbandonedOrderDroplet` servlet bean.
How to turn off this event	Set the `/atg/commerce/order/abandoned/AbandonedOrderTools.sendOrderReanimatedMessage` property to `false`.

Abandoned Order is Converted

This event is triggered when an abandoned or lost order is checked out.

Class name	`atg.commerce.order.abandoned.OrderAbandoned`
JMS name	`atg.commerce.order.abandoned.OrderConverted`
Display name	Abandoned Order Converted
Message context	request
Message scope	individual

Message source	Component:
	`/atg/commerce/order/abandoned/OrderAbandonedSender`
	
	Class:
	`atg.commerce.messaging.MessageSender`

Component that calls the message source	`/atg/commerce/order/abandoned/AbandonedOrderTools`
How this event is triggered	Triggered by the Convert Abandoned Order (page 471) scenario action or `ConvertAbandonedOrderDroplet` servlet bean.
How to turn off this event	Set the `/atg/commerce/order/abandoned/AbandonedOrderTools.sendOrderConvertedMessage` property to `false`.

Abandoned Order is Lost

This event is triggered when an order is identified as lost.

Class name	`atg.commerce.order.abandoned.OrderAbandoned`
JMS name	`atg.commerce.order.abandoned.OrderLost`
Display name	Abandoned Order Lost
Message context	request
Message scope	individual

Message source	Component:
	`/atg/commerce/order/abandoned/OrderAbandonedSender`
	
	Class:
	`atg.commerce.messaging.MessageSender`

Component that calls the message source	`/atg/commerce/order/abandoned/AbandonedOrderService`
How this event is triggered	Triggered when an order is identified as lost by the `AbandonedOrderService`.
How to turn off this event	Set the `/atg/commerce/order/abandoned/AbandonedOrderTools.sendOrderLostMessage` property to `false`.

Scenario Actions

The Abandoned Order Services module includes the following scenario actions to respond to user activity on abandoned orders:

- Set Order's Last Updated Date (page 469)
- Reanimate Abandoned Order (page 470)
- Convert Abandoned Order (page 471)
- Log Promotion Information (page 472)

All of the scenario actions listed above are utilized in the Abandoned Orders scenario that is provide out-of-the-box. For information on this scenario, see the *Managing Abandoned Orders* chapter in the *ATG Commerce Guide to Setting Up a Store*.

Set Order's Last Updated Date

This action checks whether the given order has an `abandonmentInfo` item and, if it does not, creates one and associates it with the order. It then updates the `orderLastUpdated` property of the order's `abandonmentInfo` item with the current date and time.

Action Registry Tag	Value
action name	Set Last Updated
configuration component	`/atg/scenario/configuration/SetLastUpdatedConfiguration`
action execution policy	individual
action error response	continue

The Set Order's Last Updated Date action has the following parameters:

Parameter	Is required?	Description
`orderId`	yes	The ID of the order that has been modified by the user.

See also *SetLastUpdatedDroplet* in *Appendix: ATG Commerce Servlet Beans* of the *ATG Commerce Guide to Setting Up a Store*.

Reanimate Abandoned Order

This action reanimates an abandoned or lost order. More specifically, it does the following:

1. Removes the order from the list of abandoned orders in the user's `abandonedOrders` profile property *if the order is abandoned and not lost*.
2. Modifies the order's `abandonmentInfo` item as follows:
   - Sets the `state` property to REANIMATED.
   - Sets the `reanimationDate` property to the current date and time.
3. Fires an `AbandonedOrderReanimated` message if the `AbandonedOrderTools.sendOrderReanimatedMessage` property is set to `true`.

Note that if the given order is not abandoned or lost, the action does nothing.

Action Registry Tag	Value
action name	Reanimate Abandoned Order
configuration component	`/atg/scenario/configuration/ReanimateAbandonedOrderConfiguration`
action execution policy	individual
action error response	continue

The Reanimate Abandoned Order action has the following parameters:

Parameter	Is required?	Description
`orderId`	yes	The ID of the reanimated order.

See also *ReanimateAbandonedOrderDroplet* in *Appendix: ATG Commerce Servlet Beans* of the *ATG Commerce Guide to Setting Up a Store*.

Convert Abandoned Order

This action converts an abandoned, reanimated, or lost order. More specifically, it does the following:

1. Removes the order from the list of abandoned orders in the user's `abandonedOrders` profile property *if the order was abandoned and not lost or reanimated*.
2. Modifies the order's `abandonmentInfo` item as follows:
   - Sets the `state` property to CONVERTED.
   - Sets the `conversionDate` property to the current date and time.
3. Fires an `AbandonedOrderConverted` message if the `AbandonedOrderTools.sendOrderConvertedMessage` property is set to `true`.

Note that if the `state` property in the order's `abandonmentInfo` item is null, then the order has never been abandoned, and the action does nothing.

Action Registry Tag	Value
action name	Convert Abandoned Order
configuration component	`/atg/scenario/configuration/ConvertAbandonedOrderConfiguration`
action execution policy	individual
action error response	continue

The Convert Abandoned Order action has the following parameters:

Parameter	Is required?	Description
`orderId`	yes	The ID of the converted order.

See also *ConvertAbandonedOrderDroplet* in *Appendix: ATG Commerce Servlet Beans* of the *ATG Commerce Guide to Setting Up a Store*.

Log Promotion Information

This action logs promotion-related information for a converted order. It calculates the number and total value of the promotions applied to the converted order and stores the information in a `convertedOrder` item in the `AbandonedOrderRepository`.

Action Registry Tag	Value
action name	Log Promotion Information
configuration component	`/atg/scenario/configuration/LogPromotionInfoConfiguration`
action execution policy	individual
action error response	continue

The Log Promotion Information action has the following parameters:

Parameter	Is required?	Description
`orderId`	yes	The ID of the order whose promotion information is to be logged.

Tracking Abandoned Orders of Transient Users

The Abandoned Order Services module can also track orders abandoned by transient users, that is, anonymous or guest users who are not associated with a profile maintained in the profile repository database. Orders abandoned or submitted by transient users are recorded by a scenario in a dataset, making information about these orders available for analysis.

AbandonedOrderEventListener

Transient order tracking is handled by the component `/atg/commerce/order/abandoned/AbandonedOrderEventListener`. This component has two purposes.

When a session is destroyed (either because the user logs out or the session times out), the `AbandonedOrderEventListener` checks whether a profile was associated with the session, and whether that profile is transient. If so, it then checks whether a non-empty shopping cart is also associated with the session, and whether that shopping cart's total value is at least as great as the `AbandonedOrderEventListener.minimumAmount` property. By default, the `minimumAmount` property of the `AbandonedOrderEventListener` points to the `minimumAmount` property in `AbandonedOrderService`. If the order meets the criteria, then a new `TransientOrderEvent` is created, and populated with the `orderId`, `amount`, and the `siteId` if applicable. The value `0` is placed in the event's `submitted` property. Then, the event is sent out on the messaging system.

The AbandonedOrderEventListener also registers as a listener for SubmitOrder events. When a SubmitOrder event is received, the AbandonedOrderEventListener checks whether the profile associated with the order is transient. If so, then it also creates a new TransientOrderEvent, this time populating the submitted property in the event with the value 100. This allows Commece to calculate the percentage of transient orders by averaging the values of the submitted property.

TransientOrderRecorder

The TransientOrderEvents generated by the AbandonedOrderEventListener are caught by the TransientOrderRecorder scenario. This scenario listens for the TransientOrderEvents and records the events in the Transient Order Reporting Dataset, storing all the values that the event was populated with. The Transient Order Reporting Dataset lets you generate information about orders abandoned by transient users.

Turning Off Transient Order Tracking

If you don't want to track transient orders, you can turn off this feature by:

- setting the enabled property of the AbandonedOrderEventListener to false, so that TransientOrderEvents will not be generated; and
- disabling the AbandonedOrders > TransientOrderRecorder scenario.

Customizations and Extensions

This section provides information on how to customize or extend the Abandoned Order Services module in the following ways:

Defining Additional Types of Abandoned and Lost Orders (page 473)

Modifying the Criteria Used to Identify Abandoned and Lost Orders (page 475)

Defining Additional Types of Abandoned and Lost Orders

While the default implementation of the Abandoned Order Services module enables you to identify both abandoned and lost orders, some sites may require more granularity. For example, you may want to differentiate between high-priced and low-priced abandoned orders in order to respond differently to each type via scenarios and other campaigns. This section describes how to customize the module in this way, using this very example of high-priced and low-priced abandoned orders. You could use the same process to define and manage additional types of lost orders.

First, configure an instance of atg.commerce.order.abandoned.AbandonedOrderTools for each type of abandoned order you want to manage. The AbandonedOrderTools component stores the definitions of abandonment states, including the default states. In this example, you'll need to configure two instances, one for high-priced abandoned orders and a second for low-priced abandoned orders. The following table describes how to configure each component:

Component	Description of Configuration
`AbandonedOrderTools` #1 for high-priced abandoned orders	Set the `defaultAbandonedState` property to HIGH_ABANDONED. Set the `reanimatableAbandonmentStates` property to the list of all possible states that an order can be in to be considered for identification as reanimated. In this example, set the property to: HIGH_ABANDONED, LOW_ABANDONED, LOST Configure the remaining properties as necessary.
`AbandonedOrderTools` #2 for low-priced abandoned orders	Set the `defaultAbandonedState` property to LOW_ABANDONED. Set the `reanimatableAbandonmentStates` property to the list of all possible states that an order can be in to be considered for identification as reanimated. In this example, set the property to: HIGH_ABANDONED, LOW_ABANDONED, LOST Configure the remaining properties as necessary.

Then, configure an instance of `atg.commerce.order.abandoned.AbandonedOrderService` for each type of abandoned order you want to identify and manage. Again, in this example you'll need to configure two instances, one for high-priced abandoned orders and a second for low-priced abandoned orders. The following table describes how to configure each component:

Component	Description of Configuration
`AbandonedOrderService` #1 for high-priced abandoned orders	Set the `minimumAmount` property as desired, for example, to $100 dollars. Set the `abandonedOrderTools` property to point to the `AbandonedOrderTools` component you created for high-priced orders. Set the `schedule` property so that this service runs **before** the `AbandonedOrderService` that you created for low-priced abandoned orders. Otherwise, the latter service will return and identify both the high-priced and low-priced orders as abandoned. Configuring the remaining properties as necessary.

Component	Description of Configuration
`AbandonedOrderService` #2 for low-priced abandoned orders	Set the `minimumAmount` property as desired, for example, to $10 dollars. Set the `abandonedOrderTools` property to point to the `AbandonedOrderTools` component you created for low-priced orders. Set the `schedule` property so that this service runs **after** the `AbandonedOrderService` that you created for high-priced abandoned orders. Otherwise, this service will return and identify both the high-priced and low-priced orders as abandoned. Configuring the remaining properties as necessary.

Modifying the Criteria Used to Identify Abandoned and Lost Orders

As described in Detecting Abandoned and Lost Orders (page 458), the `/atg/commerce/order/abandoned/AbandonedOrderService` uses a specific set of criteria when it queries the order repository for orders to identify as abandoned and lost. (See that section for the list of criteria.) The queries used are returned by the following methods of class `atg.commerce.order.abandoned.AbandonedOrderService`:

- `generateAbandonedQuery()`, which returns a query created from the following sub-queries:
  - `getDateQueryForAbandonedOrders()`
  - `getOrderStatesQuery()`
  - `getAbandonmentInfoQueryForAbandonedOrders()`
  - `getMinimumAmountQuery()`
- `generateLostQuery()`, which returns a query created from the following sub-queries:
  - `getDateQueryForLostOrders()`
  - `getOrderStatesQuery()`
  - `getAbandonmentInfoQueryForLostOrders()`
  - `getMinimumAmountQuery()`

For details on these methods, see `atg.commerce.order.abandoned.AbandonedOrderService` in the *ATG Platform API Reference*.

If desired, you can modify the criteria used to identify abandoned or lost orders. To do so, create a class that extends `atg.commerce.order.abandoned.AbandonedOrderService` and overrides the appropriate method listed above. Then configure an instance of your custom class in Nucleus by layering on a configuration file for the `/atg/commerce/order/abandoned/AbandonedOrderService`.

22 Generating Invoices

Commerce sites must be able to accept payment via invoice. Oracle ATG Web Commerce allows end users to specify a purchase order number and billing address and generate an invoice. It supports purchase order numbers as a payment type, generates invoices that refer to purchase order numbers as well as to other payment types, and allows a user to split the bill for an order among several invoices, or among a combination of invoices and other payment types (corporate purchasing cards, for example).

The validation process can be customized and ensures that necessary information is present. Messaging allows communication with your other financial systems, and features are designed to make integration with back-end systems simple.

This chapter includes the following sections:

Invoice Overview

The steps involved in invoice creation and processing are:

1. A user asks for an invoice to be generated upon checkout and provides a purchase order number. An `invoiceRequest` payment group is created and attached to the order. See the Invoices in Checkout (page 478) section. At this time, processors in the commerce pipeline validate the `invoiceRequest`, allowing checkout to proceed. For more information on these processors, see the *Configuring Purchase Process Services* (page 263) chapter in this manual.

2. When the order is fulfilled, the `PaymentManager` debits its payment groups. For invoices, the `PaymentManager` performs a debit by calling on the `InvoiceManager` to generate a new `Invoice` object, just as with credit cards the `PaymentManager` performs a debit by contacting a billing service to bill the credit card.

3. The `Invoice` is created and stored in the Invoice repository.

4. JMS messages are generated informing the Dynamo Scenario Manager and any other modules you designate that an invoice has been created.

The rest of this chapter explains the pieces involved in invoice processing, in the order they are mentioned above.

Invoices in Checkout

The `InvoiceRequest` object represents the customer's request to be billed for a purchase. It implements the `PaymentGroup` interface and extends `atg.commerce.order.PaymentGroupImpl`. In addition to the usual payment group fields, `InvoiceRequest` holds a purchase order number and a billing address, which are mandatory, and can also include the user's preferred invoice format and delivery mode, the payment due date, and the payment terms.

Use the `atg..order.purchase.CreateInvoiceRequestFormHandler` class to create an `InvoiceRequest` payment group. This form handler has only one implemented method, `handleNewInvoiceRequest()`. It also includes empty `preCreateInvoiceRequest()` and `postCreateInvoiceRequest()` methods for you to extend if your sites require.

Property	Description
`invoiceRequestType`	Indicates the type of `PaymentGroup` to create.
`AddToContainer`	Boolean property that determines whether to add the `InvoiceRequest` to the `PaymentGroupMapContainer` and make it the default payment group for the invoice.
`container`	The `PaymentGroupMapContainer` to which the `InvoiceRequest` is added.
`invoiceRequest`	A reference to a new `InvoiceRequest`. JSP forms can edit its properties directly.
`billingAddressPropertyName`	The name of the Profile that holds the user's billing address.
`invoiceRequestPropertiesMap`	Maps `InvoiceRequest` property names to profile property paths. The form handler determines the values for each profile property and sets them on the given `InvoiceRequest` property.

`HandleNewInvoiceRequest()` uses `GetBillingAddressPropertyName()` to check the user's profile for a business address. If one is found, it becomes the `InvoiceRequest`'s `billingAddress` as well. Then `getInvoiceRequestPropertiesMap()` copies over the specified information from the user's contract, if one exists, to the `InvoiceRequest`. See the *Using Requisitions and Contracts* (page 485) chapter for more information on these properties.

The `InvoiceRequestInfo` object includes the above information and adds references to the `Order` and `PaymentGroup` to be used when the invoice is paid.

Invoice validation is done as part of the `ValidateForCheckout` pipeline chain; see Appendix F, *Pipeline Chains* (page 699) for more information on this chain. It ensures that the `PaymentGroup` contains a valid billing address. Optionally, it verifies that the `poNumber` property is not empty. This is optional so you can configure your sites to offer payment by invoice without requiring a purchase at the time the invoice is created.

Invoice Payment

The InvoiceRequestProcessor does the work of creating invoices, based on the InvoiceRequestInfo object it receives. InvoiceRequestProcessor is then used by the PaymentManager, just like the credit card and gift certificate processors.

The InvoiceRequestProcessor holds authorize(), debit(), and credit() methods. Authorize() and credit() are empty; you can add any business logic your sites need for these procedures. The debit() method invokes the InvoiceManager's createInvoice() method, which creates a new invoice from the order and other information, then sends a JMS message indicating that the invoice was created.

To enable invoice payment, add a line to the PaymentNameToChainNameMap configuration file, located at /atg/commerce/payment/PaymentManager:

```
paymentGroupToChainNameMap=\
  atg.commerce.order.CreditCard=creditCardProcessorChain,\
  atg.commerce.order.GiftCertificate=giftCertificateProcessorChain,\
  atg.commerce.order.StoreCredit=storeCreditProcessorChain,\
  atg.commerce.order.Invoice=invoiceRequestProcessorChain
```

If you want to add further validation logic to your invoice processing, you should extend the InvoiceRequestProcessor.authorize() method. An example of how to extend validation can be found in the Extending Order Validation to Support New Payment Methods (page 314) section of the *Configuring Purchase Process Services* chapter.

Using the Invoice Manager

The InvoiceManager class provides high-level access for creating, manipulating, saving and deleting Invoice objects in the Invoice Repository. Each action leads to execution of a pipeline chain, described later in this section. The InvoiceManager methods are:

Method	Description
addInvoice	Adds a new Invoice repository item to the repository.
createInvoice	Creates a new Invoice repository item.
loadInvoiceForUpdate	Gets the invoice with the specified repository id.
getInvoicesForInvoiceNumber	Loads all invoices with a given invoice number. This method returns a list because an invoice number may not be unique. It is possible to have several invoices with the same invoice number but different repository id's, for example if a new invoice is generated to reflect partial payment of an order and a reduction in the balance due. In most cases, however, this method returns a list containing only a single invoice.
getNextInvoiceNumber	Generates the next unique invoice number to use.

Method	Description
`loadInvoice`	Loads the invoice identified by a given repository id and executes the `loadInvoice` pipeline chain. See Invoice Pipelines for more information.
`removeInvoice`	Removes the specified invoice from the repository.
`updateInvoice`	Updates invoice properties in the invoice repository and sets the repository item's `lastModified` property to the current date and time.

Invoice Pipelines

The `InvoiceManager` class executes pipelines whenever an invoice is created, loaded, updated, or removed. Each pipeline chain receives an object of type `InvoicePipelineArgs` as a parameter. `InvoicePipelineArgs` provides `getInvoice()` and `getInvoiceManager()` methods. You can also subclass `InvoicePipelineArgs` and alter the pipeline's expected type if you need additional methods.

The chains are defined in the `PipelineManager` at `/atg/commerce/invoice/pipeline/InvoicePipelineManager`. Out of the box, the only thing they do is send the messages described in the next section, but you can add links if you want to associate other actions with invoice creation and modification.

Pipeline Chain	Description
`addInvoice`	Sends a JMS message indicating that the invoice has been added to the repository.
`updateInvoice`	Sends a JMS message indicating that the invoice has been updated.
`removeInvoice`	Sends a JMS message indicating that the invoice has been removed from the repository.

The Invoice Repository

The Invoice Repository uses a SQL repository to store customer invoices; see the SQL repository chapters in the *ATG Repository Guide* for more information. Its configuration file is located at `<ATG10dir>/DCS/src/config/atg/commerce/invoice/invoicerepository.xml`. This file defines the item descriptors for the invoice classes, which in turn define important properties for the given class. The repository itself appears at `/atg/commerce/invoice/InvoiceRepository` in Nucleus.

Invoice Repository Item

The invoice item descriptor includes these properties:

Property	Description
balanceDue	Amount to be paid by the invoice.
creationDate	Date the invoice was created.
deliveryInfo	Link to a deliveryInfo item.
id	Unique repository ID of the invoice.
invoiceNumber	System-generated identifying number. InvoiceNumber is unique as provided, but you can customize your system to generate multiple invoices with the same number.
lastModifiedDate	Date the invoice was last modified.
orderId	ID of the Order item in the Order repository to which the invoice is linked.
paymentDueDate	Date the invoice must be paid.
paymentGroupId	ID of the paymentGroup item in the Order repository; identifies which parts of the order (commerce items, shipping costs, etc.) the invoice pays for.
paymentTerms	Link to a paymentTerms item.
poNumber	Purchase order number assigned to the invoice by the user at the time of checkout; need not be unique.
requisitionNumber	Value copied from the requisitionNumber property of the payment group assigned to the invoice.
type	Provided for subclassing purposes; use to indicate if an item belongs to the superclass or a subclass. Read-only.
version	Integer that is incremented automatically each time the product is updated; used to prevent version conflict. Read-only.

The orderId allows you to find the Order being paid for with the invoice, and the paymentGroupId identifies the corresponding PaymentGroup, telling you which parts of the order the invoice pays for. This can be important if billing is split among multiple payment groups. For example, if you want to generate an itemized bill that includes only the items billed to a particular invoice, you need to find the corresponding PaymentGroup and iterate over its commerce items to create the bill.

Given the orderId and paymentGroupId, you can find the Order and PaymentGroup objects using the following code:

```
public void doSomethingWithInvoice(RepositoryItem invoiceItem)
    {
      OrderManager om = getOrderManager();
      Order order = null;
      PaymentGroup payment = null;
      try
      {
        String orderId   =(String)invoiceItem.getPropertyValue("orderId");
```

```
      String paymentId =
             (String)invoiceItem.getPropertyValue("paymentGroupId");
      if (orderId != null)
        order = om.getOrder(orderId);
      if (order != null && paymentId != null)
        payment = order.getPaymentGroup(paymentId);
      ... work with order and payment...
    }
    catch (Exception e) {
      ... handle exceptions...
    }
  }
```

DeliveryInfo Repository Item

The `deliveryInfo` item descriptor represents physical or electronic delivery information with a flexible repository item. Its descriptor includes these address and contact properties: `city`, `country`, `emailAddress`, `faxNumber`, `firstName`, `lastName`, `middleName`, `phoneNumber`, `postalCode`, `prefix`, `state`, and `title`. Additional properties are described in the table below:

Property	Description
`format`	Enumerated value indicating the format in which to deliver electronic items. Oracle ATG Web Commerce does not use this property; it is provided for integration with other systems. Examples: text, HTML, XML, EDI.
`id`	Repository ID of the `deliveryInfo` item.
`preferredDeliveryMode`	Enumerated value indicating the preferred shipping method for items ordered. Oracle ATG Web Commerce does not use this property; it is provided for integration with other systems. Examples: USPS, fax, e-mail.
`type`	Provided for subclassing purposes; use to indicate if an item belongs to the superclass or a subclass. Read-only.
`version`	Integer that is incremented automatically each time the product is updated; used to prevent version conflict. Read-only.

You can use the `format` and `preferredDeliveryMode` properties to let your customers indicate their delivery preferences, then consult these preferences when exporting invoices to other systems. By default Oracle ATG Web Commerce does not use these properties, but provides them as a convenience to developers integrating the invoice generation system with external financial or billing systems.

PaymentTerms Repository Item

The `paymentTerms` item descriptor represents payment terms for an order, expressed in terms of discount percentage, discount days, and net days. Its descriptor includes the following properties:

Property	Description
discountDays	Period within which the discountPercentage applies.
discountPercentage	Discount on the price if the invoice is paid within the discountDays period.
netDays	Time at which payment in full of the net price is due.
type	Provided for subclassing purposes; use to indicate if an item belongs to the superclass or a subclass. Read-only.
version	Integer that is incremented automatically each time the product is updated; used to prevent version conflict. Read-only.

Sending Invoice JMS Messages

The addInvoice(), updateInvoice(), and removeInvoice() methods of the InvoiceManager all generate JMS messages of class atg..invoice.messaging.InvoiceMessage. This class includes the following properties:

repositoryId (String)

orderId (String)

paymentGroupId (String)

profile (RepositoryItem)

invoiceNumber (String)

PONumber (String)

requisitionNumber (String)

billingAddress (atg.core.util.ContactInfo)

preferredFormat (String)

preferredDeliveryMode (String)

balanceDue (Date)

paymentDueDate (Date)

paymentNetDays (Integer)

paymentDiscountDays (Integer)

paymentDiscountPercent (Double)

The inclusion of basic invoice properties in the message allows you to build scenarios that are invoice-driven, while the inclusion of the orderId and paymentGroupId allows more complex Oracle ATG Web Commerce-based message sinks to retrieve the invoice using the InvoiceManager and perform any required data transformation (such as translation to XML) before passing the invoice information on to external systems.

By default, invoice messages are sent to the Scenario Manager via Local JMS, but you can change your configuration to use SQL JMS if your system requires perfect reliability (see the *Dynamo Message System* chapter

in the *ATG Platform Programming Guide* for more information on JMS and its alternatives in the Oracle ATG Web Commerce platform, and on the Patch Bay).

23 Using Requisitions and Contracts

Requisitions give your customers another way to track their purchases through your Web sites by attaching requisition numbers to their orders. Contracts allow you to associate a particular catalog, price list(s), and payment terms with a specific organization.

This chapter includes the following sections:

Requisitions

Requisition numbers provide another way for buying organizations to track orders from the purchase process through billing. A buyer can add a requisition number to **any payment group** when placing an order. Oracle ATG Web Commerce keeps track of the requisition number and carries it over to the billing process when invoices are generated.

For example, a customer could make a purchase using a corporate credit card, and include a requisition number with the order for the benefit of their company's finance department. The finance department could then link the credit card bill back to a requisition form filled out before making the purchase.

Since the requisition number is part of the payment group, your back-end code and JSPs can access it like any other property. Linking the requisition number to the payment group rather than the order allows purchasing departments to aggregate purchases involving multiple buyers into a single order without losing information. After the order is created, someone at the purchasing organization must approve the order and supply any payment information your system requires. You can override this functionality changing the `checkRequiresApproval` pipeline chain. See the *Managing the Order Approval Process* (page 445) chapter for more information.

Commerce classes include several features to permit use of requisitions. The `atg.commerce.order.PaymentGroup` and `PaymentGroupImpl` classes include the `requisitionNumber` property, as does the `PaymentGroup` item descriptor in the order repository (these objects are discussed in depth in the *Working With Purchase Process Objects* (page 223) chapter of this manual). The `requisitionNumber` is generated by the purchasing organization.

The database stores requisition numbers in the `dbcpp_pmt_req` table and includes an index on the `req_number` column, allowing quick searches.

Requisition number use is optional. If your sites do not need to support requisition-based purchasing, you can simply omit requisition number fields from your order processing and checkout pages.

Contract Repository Items

The Contract repository item includes these properties:

Property	Description
`catalog`	The catalog to associate with this contract. This property is commented out; to use, uncomment it.
`comments`	String holding additional comments on the contract.
`creationDate`	Date the contract was created.
`creator`	Profile of the user who created the contract.
`displayName`	Name of the contract.
`endDate`	Date the contract ends.
`negotiatorInfo`	A `ContactInfo` item with information on the entity that negotiated the contract.
`priceList`	If using `priceLists`, the `priceList` to associate with this contract. This property is commented out; to use, uncomment it.
`startDate`	Date the contract goes into effect.
`terms`	A `contractTerms` repository object.

The `contractTerms` repository item includes these properties:

Property	Description
`paymentDiscountPercent`	Discount on the price if the invoice is paid within the `discountDays` period.
`paymentDiscountDays`	Period within which the `discountPercentage` applies.
`paymentNetDays`	Time at which payment in full of the net price is due.

Using Contracts

Invoices are automatically set up to handle contracts appropriately. Whenever a new `InvoiceRequest` is created (see the *Generating Invoices* (page 477) chapter for further information), it consults the user's organization information to see if a contract exists, and copies over the `paymentNetDays`, `paymentDiscountDays`, and `paymentDiscountPercent` information.

24 Preparing to Use Commerce Reporting

Commerce Reporting draws on data stored in the Data Warehouse, described in detail in the *ATG Data Warehouse Guide*. Before you can use this feature, data must be gathered and loaded into the warehouse. Data is collected in the form of log files, which include information about orders placed, visits to your sites, and changes to the product catalog or user records. This chapter describes the components, processes, and configuration required to create these logs.

This chapter includes the following sections:

Setting Up Commerce Reporting Environments

A typical setup for Commerce Reporting involves three Oracle ATG Web Commerce environments:

- An asset management environment running Oracle ATG Web Commerce Merchandising. This environment is used for maintaining the versioned site data, which it deploys to the production environment.
- A production environment that runs the actual Oracle ATG Web Commerce site. This site logs data about customer activity.
- A data loading environment that loads the data logged by the production environment into the Data Warehouse database tables.

This section describes how to set up these three environments. For additional information about installing database tables for Oracle ATG Web Commerce, see the *Configuring and Populating a Production Database* (page 7) chapter.

Setting up the Asset Management Environment

The asset management environment runs the Oracle ATG Web CommerceMerchandising application (which also requires Commerce and ATG Content Administration). It manages the Commerce catalog and other data

in versioned repositories, and deploys data from these versioned repositories to nonversioned repositories on the production environment. Note that the data loading environment also accesses this data, so both the production environment and the data loading environment need to be deployment targets for the merchandising environment to ensure that the repository caches on these environments remain in synch.

For information about installing and running Oracle ATG Web Commerce Merchandising, see the*ATG Merchandising Administration Guide*. In addition to following the instructions found there, run the following import script to create the default segment lists for reporting:

```
<ATG10dir>/DCS/Versioned/install/importDCSVersioned
```

When you assemble your EAR file for the asset management environment, include the `ARF.base` application module.

See also, *Configuring an Asset Management Server* in the *ATG Business Intelligence Installation and Configuration Guide*.

Setting Up the Production Environment

The production environment runs your site application (that is, the web application that constitutes your online store). As customers interact with your site, the production environment logs data about orders placed, site visits, and other customer activities. No additional tables or data are required for this environment.

Setting Up the Data Loading Environment

The data loading environment processes the data logged by the production environment and stores results in the Data Warehouse, where it can be used for running reports.

For information on creating the database tables for the Data Warehouse and data loaders, configuring data sources, and assembling the loading environment EAR file, see the *ATG Business Intelligence Installation and Configuration Guide*.

Note that the data loading environment includes several repositories whose tables are part of the production environment. These repositories include the user profile repository, the product catalog, and the order repository. Make sure the datasources for these repositories are configured to point to the production database. For information about configuring datasources in general, see the *ATG Business Intelligence Installation and Configuration Guide*.

Configuring a Parent Catalog

Each product can have multiple parent categories, one for each catalog that includes the product. Therefore, you must designate a reporting catalog, which determines which catalog supplies the parent category information for a given product. You can do this in the `reportingCatalogId` property of the `/atg/reporting/datawarehouse/CommerceWarehouseConfiguration` component. This property is `null` by default. If the property is `null` and a product has only a single parent category, that category is used. If it is null and there is more than one potential parent category, the first category is used, and a warning is logged.

Logging Data for Commerce Reporting

Data logging takes place in two environments in an Oracle ATG Web Commerce context. Some data comes from your production environment, including site visits, orders, customer/visitor changes and segment definition changes. Other data comes from the merchandising environment, including changes to categories, products, SKUs and promotions.

Data logging components for users, segments, and site visits are located in `/atg/reporting/datacollection/userprofiling`. The logging components for orders and the product catalog are located in `/atg/reporting/datacollection/commerce`. All loggers have the following common characteristics:

1. A listener monitors for one or more event types that signify log-worthy activity in the listener's domain. The event is placed on a queue, which notifies its associated logging component of the event. The type of event that triggers logging differs among the various types of logged data.

2. Depending on the log type, filtering may be performed to ensure that only changes of interest for reporting are logged.

3. The data logger writes information about the events to a tab-delimited text file. Each log file has a unique name in the format *`log-type_timestamp_unique-ID`*`.data`. For example:

   ```
   catalog_12-04-2006_03-27-31-442_200055200055.data
   ```

   The unique ID is generated by the `/atg/dynamo/service/IdGenerator` component. Including the ID in the filename means that even if you configure your system to send all log files generated by any number of Dynamo instances to a central location, there is no risk that they will overwrite one another.

   All log files consist of one tab-delimited line per entry.

   Log files are rotated by the `FileLogger` components, either at a scheduled time or when a configured limit on the number of log entries is reached. During rotation, the currently open log file is closed and a new one opened. The logging component fires a JMS message, which is received by the `LogRotationSink`, which adds the closed log file and its type to the queue in the `DataWarehouseLoaderRepository`.

   Note: Rotation occurs only if the current log file contains data.

The sections that follow provide specific information on each logger. Since all loggers are configured in similar ways, for logger configuration, see the Data Logging Configuration (page 496) section.

Site Visit Data Logging

The `/atg/reporting/datacollection/userprofiling/SiteVisitManager` Nucleus component listens for session start, session end and page view events. It accumulates information about a site visit in a session-scoped component called `/atg/reporting/datacollection/userprofiling/SiteVisitRequest`. When the session ends, the `SiteVisitManager` sends a JMS message. The `siteVisit` message has a JMS type of `atg.dps.SiteVisit`, and is an instance of the `atg.userprofiling.dms.SiteVisitMessage` class.

This message is received by the `SiteVisitMessageSink`, which calls the `/atg/reporting/datacollection/userprofiling/SiteVisitQueue` component. The `SiteVisitQueue` component calls the `/atg/reporting/datacollection/userprofiling/SiteVisitFileLogger` component, which adds an entry to the `site-visit_`*`timestamp_unique-ID`*`.data` file.

The log file includes the following details for each visit:

- Session ID

- Profile ID
- Session start timestamp
- Session end timestamp
- Visit duration in seconds
- Number of pages viewed

The following is an example of the site visit log:

```
0385E85E9E8BC68D34354D132AFAB29C17000311/07/2006 14:38:46
11/07/2006 14:38:4601
```

```
65F444358B90D134E745E0E9859F397412000011/07/2006 14:35:27
11/07/2006 15:01:55158825
```

Order Submit Data Logging

To log order data, the `CommerceOrderSubmitPipeline` emits a `SubmitOrder` event. The event is received by the `/atg/reporting/datacollection/commerce/OrderLogEntryQueueSink` and forwarded on to the `/atg/reporting/datacollection/commerce/OrderFileLogger` component.

There is one log entry per order. The logging file is named `order_timestamp_`*`unique ID`*`.data`, and it includes the following:

- Timestamp
- Profile ID of the user who placed the order
- Order ID
- Set of segments the user was in when the `OrderSubmit` message was fired

  This means those segments for which both of the following apply:

  - The user is a member of the segment
  - The segment is part of the `CommerceReporting` segment list

If the logger encounters a null value, it writes the empty string to the log file. If you prefer the logger to write the string `null` instead, set the `skipNullOjbect` property of the `OrderFileLogger` component to `false`.

The following is an example of the order log:

```
12/04/2006 16:39:42o100041100025E47ED51379664EBA2077AEFCDC0F90D
     UserProfiles:ValueShoppers
```

Commerce Search Data Logging

This section applies if you are using Oracle ATG Web Commerce Search as part of your Commerce site, and want to report on the integration between these products. Search logs information when searches are performed,

and when customers click links in search results to view or purchase items. See the *ATG Search Administration Guide* for information on the former; this section discusses the latter.

To associate search terms with items that are viewed or purchased, your sites must record "click-through" events. These occur when a customer clicks on a product or SKU returned by a search, to view it or purchase it. The recording of these events works like this:

- For each search result, the `/atg/search/droplet/GetClickThroughId` servlet bean generates a click-through ID, which you append to the URL for that result using a query parameter. The servlet bean also adds the result document to a cache.
- When a customer clicks a link to view a search result, the `/atg/search/servlet/pipeline/SearchClickThroughServlet` examines the request URL, finds the click-through ID, and uses it to look up the document in the cache. If it finds the document, the servlet fires an `atg.search.events.SearchClickThroughMessage` JMS event containing the search request and response objects and the selected document. This event is logged to be used for reporting.

The `QueryFileLogger` component logs the search information; see *Data Logging for Reports* in the *ATG Search Administration Guide* for information on this component.

For more information about configuring your Commerce site to log data for Commerce Search reports, see the *ATG Search Administration Guide*.

Product Catalog Data Logging

When you make changes to category, product, SKU or promotion items in your catalog, ATG Content Administration creates a list of changed items to deploy. Once the deployment is completed and the publishing clients have switched their data sources, the `/atg/reporting/datacollection/commerce/ProductCatalogDeploymentListener` component on the ATG Content Administration server is notified of the switch and the list of changes.

The `sourceRepositoryPathToItemDecriptorNames` property of the `ProductCatalogDeploymentListener` component defines the item types for which changes should be logged and the Data Warehouse updated.

```
sourceRepositoryPathToItemDecriptorNames=\
      /atg/commerce/catalog/ProductCatalog=category;product;sku;promotion
```

You can use the `ignorableTargets` property to specify any Content Administration targets where changes should not be logged, such as your staging environment:

```
ignorableTargets=Staging
```

Note: If you do a full deploy of your product catalog, all items are logged as changed, though the data may not be any different. During its next scheduled run, the loader will then compare every item in the catalog with the dimension data in the Warehouse Repository, and only actual changes are loaded. For large catalogs, this process can take a long time.

The JMS message type used to identify log rotation is `atg.reporting.productCatalogUpdate`.

The `catalog_`*`timestamp_unique id`*`.data` file contains the following logged information:

- Timestamp when the logging component received notification of the successful data source switch, which approximates the time when the change went live on the production server
- Nucleus path to the repository component
- Type of item that changed
- Repository item ID for the changed item
- Change type (insert, update, or delete)

The following is an example of the product catalog log:

```
10/19/2006 10:08:39     /atg/commerce/catalog/ProductCatalog product
prod70002     insert
```

```
10/19/2006 10:08:39     /atg/commerce/catalog/ProductCatalog sku
     sku81007     update
```

```
10/19/2006 10:10:39     /atg/commerce/catalog/ProductCatalog product
prod70004     delete
```

User Data Logging

The `/atg/reporting/datacollection/userprofiling/UserEventListener` component is notified whenever an insert, update, or delete is performed on either User or ContactInfo repository item descriptors in the Profile Repository. The `user_`*`timestamp_unique ID`*`.data` log created by the `/atg/reporting/datacollection/userprofiling/UserFileLogger` component includes the following data:

- Time stamp
- Dynamo path to the repository.
- Type of item that changed.
- Repository item ID for the changed item.
- Change type (insert, update, or delete)

The `UserEventListener.properties` file includes several configurable properties that allow you to filter out information you do not want to log. The `sourceRepositoryPathToItemDescriptorNames` property specifies which repository items have change information logged for them:

```
sourceRepositoryPathToItemDecriptorNames=\
     /atg/userprofiling/ProfileAdapterRepository=user;contactInfo
```

The `excludedEventTypes` property allows you to filter out change types that you do not want logged. There are four types of change event: UPDATE, DELETE, INSERT, and CACHE_INVALIDATE. By default, CACHE_INVALIDATE is excluded from logging. The property is set as follows:

```
excludedEventTypes=CACHE_INVALIDATE
```

The `itemDescriptorToPropertyNames` property allows you to explicitly identify which properties you want to log UPDATE changes for:

```
itemDescriptorToPropertyNames=\
/atg/userprofiling/ProfileAdapterRepository:user=login
```

The following is an example of the user log:

```
10/19/2006 10:08:39     /atg/userprofiling/ProfileAdapterRepository
user     711     insert
10/19/2006 10:10:39     /atg/userprofiling/ProfileAdapterRepository
user     735     insert
10/19/2006 10:10:39     /atg/userprofiling/ProfileAdapterRepository
user     1710002     update
```

Segment Data Logging

Segments (also known as profile groups) represent different groups in your customer base. To log segment data for reporting, the `/atg/reporting/datacollection/userprofiling/SegmentChangeListenerService` listens for `ConfigurationEvents`. These events occur when updates, deletions, or insertions are made to segments. The listener sends details of segment changes to the `/atg/reporting/datacollection/userprofiling/SegmentFileLogger` via a data collection queue. The `SegmentFileLogger` creates the log entry.

The `segment_`*`timestamp_unique id`*`.data` file contains the following logged information:

- Timestamp
- Name of the repository to which segments are bound within the repository group container, specified in `/atg/registry/RepositoryGroups`
- Item type, which is always `segment`
- A *`repository:name`* combination, which uniquely identifies the segment. For example, `UserProfile:HighValueCustomers`.
- Change type (INSERT, UPDATE, or DELETE)
- Segment name
- Segment description from the `serviceInfo` property of the associated Nucleus component

The following is an example of the segment log:

```
10/19/2006 20:46:01  UserProfiles  segment  UserProfiles:DiscountShopper
  INSERT  DiscountShopper  "Prefers sale items"
10/19/2006 21:46:10  UserProfiles  segment  UserProfiles:DiscountShopper
  UPDATE  DiscountShopper  "Only sale items"
10/19/2006 21:50:13  UserProfiles  segment  UserProfiles:DiscountShopper
```

```
DELETE  DiscountShopper  "Only sale items"
```

Data Logging Configuration

This section describes those aspects of logging configuration that apply to all logging components.

Enabling Data Logging

Logging components can be enabled individually or as a group. By default, the `enabled` property of each individual file logger is set to refer to the `enabled` property of an `/atg/dynamo/service/` component. Therefore, data collection for the product catalog can be enabled by setting the `enabled` property of the `/atg/dynamo/service/DeploymentDWDataCollectionConfig` component to `true`. Data collection for all other file loggers can be enabled by setting the `enabled` property of the `/atg/dynamo/service/DWDataCollectionConfig` component to `true`.

Changing the `enabled` property of either component affects all of the loggers that point to that component. After enabling any disabled logging component, you must restart the Oracle ATG Web Commerce instance to begin logging.

Configuring Log File Rotation

You can schedule file rotation using the `schedule` property of each of the individual file logging components. For example, to rotate the segment fie log hourly, in the `/atg/reporting/datacollection/userprofiling/SegmentFileLogger.properties` set the following:

```
schedule=every 1 hour without catch up
```

For information on how to specify a schedule, see the *Scheduler Services* section of the *Core Dynamo Services* chapter in the *ATG Platform Programming GuideATG Platform Programming Guide*.

You can use the `dataItemThreshold` of each component to trigger log rotation when the specified number of log records is exceeded. If you set both properties, the log will be rotated whenever it exceeds the `dataItemThreshold` and at the scheduled time, as long as the log file contains at least one log entry.

By default, the loggers are scheduled to rotate as follows:

- Product Catalog—Every 15 minutes
- User—Hourly
- Segment--Hourly
- Site Visit—Hourly
- Order--Hourly

Configuring Timestamp Formats in Log Entries

To change the timestamp formatting in log entries, edit the `formatFields` property in the file logging component. For example, to change the timestamp format for the user file logger, edit the `timestampDateFormat` property in `/atg/reporting/datacollection/userprofiling/UserFileLogger.properties`:

```
timestampDateFormat=MM-dd-yyyy_HH-mm-ss-SS
```

Configuring Log File Location

Both the logging and file loading components need to know the location of the log files. Both components must be configured to point to the same physical location, though the logical names may differ depending on your network configuration.

The log file location is configured by the `defaultRoot` property in all logging and loading components. By default, all of these components are configured to refer to the `<ATG10dir>/home/server/`*`server_name`*`/logs` directory.

Initial Data Logging for Catalogs, Users, and Segments

For catalogs, users, and segments, in order for reporting to be useful you need a warehouse record of the initial state of the data. To create this initial record, you can do one of two things.

- On the Dynamo Administration page, invoke the `doWalk` method of the `/atg/reporting/datawarehouse/service/UserService` and `/atg/reporting/datawarehouse/service/ProductCatalogService` components, and the `bulkLoad` method of the `/atg/reporting/datacollection/userprofiling/SegmentChangeListenerService`.

  Note: Before you can use this method, you must set the `enabled` property of the `/atg/dynamo/service/LogRotationMessageSource` component to `true`. You can disable it again when initial logging is finished. See the JMS Message Information for Data Logging (page 498) section for more information on this component.

- If you use ATG Content Administration, you can do a full deployment of your catalog. This creates a log entry for every catalog item, which is then treated as a change and loaded into the Data Warehouse.

Note: Either method will take considerable time, especially if you have a large catalog.

The `UserService` and `ProductCatalogService` components walk the entire product catalog and user repository and write a log entry for each item, with a null timestamp and an event type of `INSERT`. The regular data loading process can then load the data into the Data Warehouse.

Both components must be configured with the repository to be loaded and the item descriptors to include in the walk. For example, in `UserService.properties`, configure the following:

```
repository=/atg/userprofiling/ProfileAdapterRepository
itemDescriptorNames=user
```

For `ProductCatalogService.properties`, configure the following:

```
repository=/atg/commerce/catalog/ProductCatalog
itemDescriptorNames=category,product,sku,promotion
```

No configuration is required for `SegmentChangeListenerService`.

JMS Message Information for Data Logging

Oracle ATG Web Commerce includes the Patch Bay configuration required for data logging. Logging components use the `messageSource` property to specify a component to use as the source for logging messages, as shown in this example:

```
messageSource=/atg/dynamo/service/LogRotationMessageSource
```

The `LogRotationMessageSource` component is responsible for sending out logging-related JMS messages. The message source definition for this component in `dynamoMessagingSystem.xml` is:

```
    <message-source>
      <nucleus-name>
        /atg/dynamo/service/LogRotationMessageSource
      </nucleus-name>

      <output-port>
        <port-name>
         DEFAULT
        </port-name>
        <output-destination>
         <provider-name>
             local
         </provider-name>
         <destination-name>
            localdms:/local/Reporting/LogRotation
          </destination-name>
          <destination-type>
            Queue
          </destination-type>
        </output-destination>
      </output-port>
</message-source>
```

The message sink for the `localdms:/local/Reporting/LogRotation` queue has the following configuration:

```
  <message-sink>
      <nucleus-name>
        /atg/reporting/datawarehouse/loaders/LogRotationSink
      </nucleus-name>
      <input-port>
        <port-name>
         DEFAULT
        </port-name>
        <input-destination>
          <provider-name>
             local
         </provider-name>
          <destination-name>
             localdms:/local/Reporting/LogRotation
          </destination-name>
          <destination-type>
             Queue
          </destination-type>
```

```
        </input-destination>
    </input-port>
</message-sink>
```

The table that follows lists the `logRotationMessageType` for each logging component:

Logging Component	Message Type
`OrderFileLogger`	`atg.reporting.submitOrder`
`ProductCatalogLogger`	`atg.reporting.productCatalogUpdate`
`SegmentFileLogger`	`atg.reporting.segmentUpdate`
`SiteVisitFileLogger`	`atg.reporting.siteVisit`
`UserFileLogger`	`atg.reporting.userUpdate`
`UserServiceFileLogger`	`atg.reporting.userUpdate`
`ProductCatalogServiceFileLogger`	`atg.reporting.productCatalogUpdate`

Appendix A. Web Services

This appendix includes descriptions of all of the Oracle ATG Web Commerce Web services and security information specific to the Oracle ATG Web Commerce Web services. For more information on Web services, see the *ATG Web Services and Integration Framework Guide*.

The Oracle ATG Web Commerce Web services are separated into the following groups:

Note: Some Web services descriptions include URLs with the variables *`hostname:port`*, in which *`hostname`* signifies the name of the machine running your application and *`port`* signifies the port on that machine designated by your application server for handling HTTP requests.

Order Management Web Services

All order management web services are included in `commerceWebServices.ear` in the `orderManagement.war` web application.

Ear file	`commerceWebServices.ear`
War file	`orderManagement.war`
Context-root	`commerce/orderManagment`

For an example of a client calling an order management web service, see the Order Management Web Services Example (page 517) section. For the recommended security policies to be associated with order management web services, see the Commerce Web Services Security (page 536) section.

This section includes information on the following Order Web services.

addCreditCardToOrder Web Service

The `addCreditCardToOrder` Web service adds the credit card information to an order.

Servlet Name	`addCreditCardToOrder`
Input Parameters	`orderId` – The ID of the order. `creditCard` – The credit card to be added to the order.
Output	The ID of newly created payment group.

Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/addCreditCardToOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/addCreditCardToOrder/addCreditCardToOrder`
Method	`addCreditCardToOrder(String orderId, BasicCreditCardInfoImpl creditCard)`
Security `FunctionalName`	`orderManagement`

addItemToOrder Web Service

The `addItemToOrder` Web service adds the given product/SKU to the order. If the product/SKU already exists, the quantity is increased.

Servlet Name	`addItemToOrder`
Input Parameters	`orderId` – The ID of the order to which the item will be added. `productId` – The ID of the product to be added to the order. `skuId` – The ID of the SKU to be added to the order. `quantity` – The number of the specified products or SKUs to be added to the order.
Output	The ID of the commerce item that was either added or updated.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/addItemToOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/addItemToOrder/addItemToOrder`
Method	`addItemToOrder(String orderId, String productId, String skuId, long quantity)`
Security `FunctionalName`	`orderManagement`

addItemToShippingGroup Web Service

The `addItemToShippingGroup` Web service adds the given product/SKU to a shipping group. If the product/SKU already exists in the shipping group, the quantity is increased.

Servlet Name	`addItemToShippingGroup`
Input Parameters	`orderId` – The ID of the order to which the item will be added. `productId` – The ID of the product to be added to the shipping group. `skuId` – The ID of the SKU to be added to the shipping group. `quantity` – The number of the specified products or SKUs to be added to the shipping group. `shippingGroupId` – The ID of the shipping group to which the item will be added.
Output	The ID of the commerce item that was either added or updated.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/order/addItemToShippingGroup?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/order/addItemToShippingGroup/addItemToShippingGroup`
Method	`addItemToShippingGroup(String orderId, String productId, String skuId, long quantity, String shippingGroupId)`
Security `FunctionalName`	`orderManagement`

addShippingAddressToOrder Web Service

The `addShippingAddressToOrder` Web service adds a new shipping address to the order.

Servlet Name	`addShippingAddressToOrder`
Input Parameters	`orderId` – The ID of the order. `address` – The address to be added to the order.
Output	The ID of the newly created shipping group.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`

Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/addShippingAddressToOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/addShippingAddressToOrder/addShippingAddressToOrder`
Method	`addShippingAddressToOrder(String orderId, ContactInfo address)`
Security `FunctionalName`	`orderManagement`

cancelOrder Web Service

The `cancelOrder` Web service cancels the specified Order.

Servlet Name	`cancelOrder`
Input Parameters	`orderId` - The ID of the order to be cancelled
Output	An integer with one of three values: 1 – the request was ignored 2 – the order was deleted 3 – a JMS message was sent requesting that the order be cancelled
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/cancelOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/cancelOrder/cancelOrder`
Method	`cancelOrder(String orderId)`
Security `FunctionalName`	`orderManagement`

createOrder Web Service

The `createOrder` Web service creates a new order and assigns the current profile ID to the order.

Servlet Name	`createOrder`

Input Parameters	`orderType` – (Optional) the type of order to create. Null indicates the default order type.
Output	The ID of the newly created order.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/createOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/createOrder/createOrder`
Method	`createOrder(String orderType)`
Security `FunctionalName`	`orderCreation`

createOrderForUser Web Service

The `createOrderForUser` Web service creates a new order of the given type for the given user. If `orderType` is null, it creates an order of the default order type. This operation is intended to be used by applications that need to create orders on behalf of another user, such as a CSR application.

Servlet Name	`createOrderForUser`
Input Parameters	`ProfileId` - The user who will own the order `OrderType` - The type of order to create. Null indicates the default order type.
Output	`OrderId` - The order ID of the new order.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/createOrderForUser?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/createOrderForUser/createOrderForUser`
Method	`createOrderForUser(String orderType, String profileId)`
Security `FunctionalName`	`orderCreationForUser`

createOrderFromXML Web Service

The `createOrderFromXML` Web service takes the order specified in the parameter and saves it to the order repository. The XML must follow a schema that represents an order. For more information, see the *Repository to XML Data Binding* chapter in the *ATG Web Services and Integration Framework Guide*.

Servlet Name	`createOrderFromXML`
Input Parameters	`OrderAsXML` - The XML document represent ting the entire order. `ProfileId` – The ID of the profile to be associated with the new order.
Output	`OrderId` – The ID of the new order.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Method	`createOrderFromXML(String pOrderXML, String pProfileId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/order/createOrderFromXML?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/order/createOrderFromXML/createOrderFromXML`
Security `FunctionalName`	`orderCreation`

getCurrentOrderId Web Service

The `getCurrentOrderId` Web service returns the ID of the current user's current order from the shopping cart. If there is no order in the shopping cart, null is returned.

Servlet Name	`getCurrentOrderId`
Input Parameters	`none`
Output	The ID of the current order or null if there is no current order.
Web Service Class	`atg.commerce.order.OrderService`
Nucleus Component	`/atg/commerce/order/orderServices`

Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/getCurrentOrderId?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/getCurrentOrderId/getCurrentOrderId`
Method	`getCurrentOrderId()`
Security `FunctionalName`	`getCurrentOrderId`

getDefaultPaymentGroupId Web Service

The `getDefaultPaymentGroupId` Web service retrieves the first payment group ID from the order.

Servlet Name	`getDefaultPaymentGroupId`
Input Parameters	`OrderId` - The ID of the order.
Output	The default payment group ID of the order
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/getDefaultPaymentGroupId?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/getDefaultPaymentGroupId/getDefaultShippingGroupId`
Method	`getDefaultPaymentGroupId(String pOrderId)`
Security `FunctionalName`	`orderManagement`

getDefaultShippingGroupId Web Service

The `getDefaultShippingGroupId` Web service retrieves the first shipping group ID from the order.

Servlet Name	`getDefaultShippingGroupId`
Input Parameters	`OrderId` - The ID of the order.
Output	The default shipping group ID of the order
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/getDefaultShippingGroupId?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/getDefaultShippingGroupId/getDefaultShippingGroupId`
Method	`getDefaultShippingGroupId(String pOrderId)`
Security `FunctionalName`	`orderManagement`

getOrderAsXML Web Service

The `getOrderAsXML` Web service returns the given order as an XML string using the `GetService`. The mapping file used is the `outboundOrderMappingFileName`. The XML must follow a schema that represents an order. For more information, see the *Repository to XML Data Binding* chapter in the *ATG Web Services and Integration Framework Guide*.

Servlet Name	`getOrderAsXML`
Input Parameters	`orderId` - The ID of the order to be returned as an XML string.
Output	An XML string valid against the schema.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/getOrderAsXML?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/getOrderAsXML/getOrderAsXML`

Method	`getOrderAsXML(String pOrderId)`
Security `FunctionalName`	`orderManagement`
Repository Component	`/atg/commerce/order/OrderRepository`
Item Descriptor	`order`

getOrdersAsXML Web Service

The `getOrdersAsXML` Web service will look up orders for a particular user. The possible search types are:

- incomplete - return the orders that are incomplete
- open - return the orders that are in one of the configured open states
- closed - return the orders that are in one of the configured closed states.

Servlet Name	`getOrdersAsXML`
Input Parameters	`profileId` – The ID of the user whose orders are returned. `startIndex` – Which order should be the first order returned. If you want to start at the beginning, use 0. `numOrders` – How many orders should be returned. If all orders, use -1. `searchType` – Which orders should be returned: incomplete, open, or closed. See above description for details. `sortProperty` – (Optional) Which property should the results should be sorted on. If null, the orders are not sorted. `ascending` – Which direction should the orders be sorted in. If true, then orders are sorted in ascending order. If false, then orders are sorted in descending order.
Output	An array of XML strings, each representing an order.
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/OrderServices`
Method	`getOrdersAsXML(String pProfileId, int pStartIndex, int pEndIndex, String pSearchType, String pSortProperty, Boolean pAscending)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/order/getOrdersAsXML?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/order/getOrdersAsXML/getOrdersAsXML`

Security `FunctionalName`	`orderLookupOperation`
Repository Component	`/atg/commerce/order/OrderRepository`
Item Descriptor	`order`

getOrderStatus Web Service

The `getOrderStatus` Web service returns the `OrderStatus` associated with the specified ID.

Servlet Name	`getOrderStatus`
Input Parameters	`orderId` - The ID of the order.
Output	A complete `OrderStatus` object. This includes: String `orderId` Date `dateSubmitted` String `orderState` List of items and their states List of shipping groups and their states List of payment groups
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Method	`getOrderStatus(String orderId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/order/getOrderStatus?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/order/getOrderStatus/getOrderStatus`
Security `FunctionalName`	`orderManagement`

moveItemBetweenShippingGroups Web Service

The `moveItemBetweenShippingGroups` Web service moves the given quantity of the commerce item from one shipping group into another within an order.

Servlet Name	`moveItemBetweenShippingGroups`

Input Parameters	`orderId` - The ID of the order. `commerceItemId` - The ID of the item to be moved between shipping groups. `quantity` – the number of items to be moved. `sourceShippingGroupId` - The ID of the shipping group *from* which the item will be moved. `targetShippingGroupId` - The ID of the shipping group *to* which the item will be moved.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/moveItemBetweenShippingGroups?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/moveItemBetweenShippingGroups/moveItemBetweenShippingGroups`
Method	`moveItemBetwaeenShippingGroups(String orderId, String commerceItemId,long quantity, String sourceShippingGroupId,String targetShippingGroupId)`
Security `FunctionalName`	`orderManagement`

removeCreditCardFromOrder Web Service

The `removeCreditCardFromOrder` Web service removes the credit card with the given number from the order.

Servlet Name	`removeCreditCardFromorder`
Input Parameters	`orderId` - The ID of the order from which the credit card will be removed. `creditCardNumber` – The credit card number to be removed from the order.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/removeCreditCardFromOrder?WSDL`

Endpoint URL	`http://hostname:port/commerce/order/removeCreditCardFromOrder/removeCreditCardFromOrder`
Method	`removeCreditCardFromOrder(String orderId, String creditCardNumber)`
Security `FunctionalName`	`orderManagement`

removeItemFromOrder Web Service

The `removeItemFromOrder` Web service decreases the quantity of the given commerce item. If the new quantity is 0 or less, the item is removed.

Servlet Name	`removeItemFromOrder`
Input Parameters	`orderId` - The ID of the order from which the item will be removed. `commerceItemId` - The ID of the item to remove.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/removeItemFromOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/removeItemFromOrder/removeItemFromOrder`
Method	`removeItemFromOrder(String orderId, String commerceItemId)`
Security `FunctionalName`	`orderManagement`

removeItemQuantityFromShippingGroup Web Service

The `removeItemQuantityFromShippingGroup` Web service removes a quantity of the given item from the shipping group. The commerce item can be completely removed from the order in this way.

Servlet Name	`removeItemQuantityFromShippingGroup`

Input Parameters	`orderId` - The ID of the order. `commerceItemId` - The ID of the item to be removed. `quantity` – the number of items to be removed. `shippingGroupId` - The ID of the shipping group from which the item will be removed.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/removeItemQuantityFromShippingGroup?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/removeItemQuantityFromShippingGroup/removeItemQuantityFromShippingGroup`
Method	`removeItemQuantityFromShippingGroup(String orderId, String commerceItemId, long quantity, String shippingGroupId)`
Security `FunctionalName`	`orderManagement`

removePaymentGroupFromOrder Web Service

The `removePaymentGroupFromOrder` Web service removes the payment group from the order.

Servlet Name	`removePaymentGroupFromOrder`
Input Parameters	`orderId` - The ID of the order from which the payment group will be removed. `paymentGroupId` - The ID of the payment group to be removed.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/removePaymentGroupFromOrder?WSDL`

Endpoint URL	`http://hostname:port/commerce/order/removePaymentGroupFromOrder/removePaymentGroupFromOrder`
Method	`removePaymentGroupFromOrder(String orderId, String paymentGroupId)`
Security `FunctionalName`	`orderManagement`

removeShippingGroupFromOrder Web Service

The `removeShippingGroupFromOrder` Web service removes the payment shipping group from the order.

Servlet Name	`removeShippingGroupFromOrder`
Input Parameters	`orderId` – The ID of the order from which the shipping group will be removed. `shippingGroupId` – The ID of the shipping group to be removed from the order.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/removeShippingGroupFromOrder?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/removeShippingGroupFromOrder/removeShippingGroupFromOrder`
Method	`removeShippingGroupFromOrder(String orderId, String shippingGroupId)`
Security `FunctionalName`	`orderManagement`

setItemQuantity Web Service

The `setItemQuantity` Web service sets the item quantity for the given commerce item in the given shipping group. The default shipping group is used if the shipping group ID argument is null. If the quantity is set to zero, the item is removed from the shipping group (and the order if the item is part of only one shipping group).

Servlet Name	`setItemQuantity`

Input Parameters	`orderId` - The ID of the order containing the item. `commerceItemId` - The ID of the item whose quantity will be modified. `quantity` - The new quantity of the item.
Output	
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/setItemQuantity?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/setItemQuantity/setItemQuantity`
Method	`setItemQuantity(String orderId, String commerceItemId, long quantity, String pShippingGroupId)`
Security `FunctionalName`	`orderManagement`

setOrderAmountToPaymentGroup Web Service

The `setOrderAmountToPaymentGroup` Web service assigns a given amount to the payment group.

Servlet Name	`setOrderAmountToPaymentGroup`
Input Parameters	`orderId` - The ID of the order. `paymentGroupId`- The ID of the payment group to which the amount will be added. `amount` - The amount of money to be added to the payment group.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/order/setOrderAmountToPaymentGroup?WSDL`
Endpoint URL	`http://hostname:port/commerce/order/setOrderAmountToPaymentGroup/setOrderAmountToPaymentGroup`

Method	`setOrderAmountToPaymentGroup(String orderId, String paymentGroupId, double amount)`
Security `FunctionalName`	`orderManagement`

submitOrderWithReprice Web Service

The `submitOrderWithReprice` Web service takes the order specified in the parameter and puts it into the order process. The order is priced and authorized by Oracle ATG Web Commerce.

Note: The currency is normally driven by the locale in the profile associated with the order.

Servlet Name	`submitOrderWithReprice`
Input Parameters	`OrderId` - The ID of the order to be submitted. `Locale` - The locale to use. This may be null.
Output	*none*
Web Service Class	`atg.commerce.order.OrderServices`
Nucleus Component	`/atg/commerce/order/orderServices`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/order/submitOrderWithReprice?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/order/submitOrderWithReprice/submitOrderWithReprice`
Method	`submitOrderWithReprice(String orderId, String locale)`
Security `FunctionalName`	`orderManagement`

Order Management Web Services Example

The following is an example Apache Axis client calling an order management web service. The `createOrder` web service is used in this example.

The first step in calling the order management web service using Apache is to generate the client stubs:

```
java org.apache.axis.wsdl.WSDL2Java
```

```
http://hostname:port/commerce/order/createOrder?WSDL
```

Next, the following code executes the `createOrder` web service:

```
CreateOrderSEIService webService = new CreateOrderSEIServiceLocator();
CreateOrderSEI createStub = webService.getCreateOrderSEIPort();
String orderId = createStub.createOrder(null);
```

Pricing Web Services

All pricing web services are included in `commerceWebServices.ear` in the `pricing.war` web application.

Ear file	`commerceWebServices.ear`
War file	`pricing.war`
Context-root	`commerce/pricing`

For an example of a client calling a pricing web service, see the Pricing Web Services Example (page 520) section. For the recommended security policies to be associated with pricing web services, see the Commerce Web Services Security (page 536) section.

This section includes information on the following Pricing Web services.

calculateOrderPrice Web Service

The `calculateOrderPrice` Web service takes an order ID and a locale, prices the order, creates and returns an `OrderPrice` object based on the information in the priced order.

Servlet name	`calculateOrderPrice`
Input Parameters	`orderId` – The ID of the order to be priced. `locale` – (Optional) String representing the locale to be considered when pricing. If this parameter is null, then the order owner's locale will be used. This will usually be null.

Output	`OrderPrice` - an object containing the complete price information for the given order
Web Service Class	`atg.commerce.pricing.PricingServices`
Nucleus Component	`/atg/commerce/pricing/PricingServices`
Method	`calculateOrderPrice(String pOrderId, String pLocale)`
Executes within a session	Yes
WSDL URL	`http://host:port/commerce/pricing/calculateOrderPrice?WSDL`
Endpoint URL	`http://host:port/commerce/pricing/calculateOrderPrice/calculateOrderPrice`
Security `FunctionalName`	`orderPricing`

calculateOrderPriceSummary Web Service

The `calculateOrderPriceSummary` Web service takes an order Id and a locale, prices the order, creates and returns an `OrderPriceSummary` object based on the information in the priced order.

Servlet name	`calculateOrderPriceSummary`
Input Parameters	`orderId` – The ID of the order to be priced. `locale` – (Optional) String representing the locale to be considered when pricing. If this parameter is null, then the order owner's locale will be used. This will usually be null.
Output	`OrderPriceSummary` - An object containing a summary of the price information for the given order.
Web Service Class	`atg.commerce.pricing.PricingServices`
Nucleus Component	`/atg/commerce/pricing/PricingServices`
Method	`calculateOrderPriceSummary(String pOrderId, String pLocale)`
Executes within a session	Yes
WSDL URL	http://*`host:port`*/commerce/pricing/calculateOrderPriceSummary?WSDL
Endpoint URL	http://*`host:port`*/commerce/pricing/calculateOrderPriceSummary/calculateOrderPriceSummary
Security `FunctionalName`	`orderPricing`

calculateItemPriceSummary Web Service

The `calculateItemPriceSummary` Web service takes a SKU ID, product ID, quantity, profile ID, and a locale, and prices the item based on the information given. The information is returned in an `ItemPriceSummary` object.

Servlet name	`calculateItemPriceSummary`
Input Parameters	`skuId` – (Optional) The SKU ID of the item to be priced. If `productId` is null, then `skuId` is required. `productId` – (Optional) The product Id of the item to be priced. If `skuId` is null, then `productId` is required. `quantity` - The quantity of the item to be considered when pricing `profileId` – (Optional) The ID of the profile to be considered when pricing. If this is null, then the current user is used. `locale` – (Optional) String representing the locale to be considered when pricing. If this is null, the locale of the current user is used.
Output	`ItemPriceSummary` - An object containing the price information for the given item
Web Service Class	`atg.commerce.pricing.PricingServices`
Nucleus Component	`/atg/commerce/pricing/PricingServices`
Method	`calculateItemPriceSummary(String pSkuId, String pProductId, long pQuantity, String pProfileId, String pLocale)`
Executes within a session	Yes
WSDL URL	`http://host:port/commerce/pricing/calculateItemPriceSummary?WSDL`
Endpoint URL	`http://host:port/commerce/pricing/calculateItemPriceSummary/calculateItemPriceSummary`
Security `FunctionalName`	`itemPricing`

Pricing Web Services Example

The following is an example Apache Axis client calling a pricing Web service. The `calculateOrderPrice` web service is used in this example.

The first step in calling a pricing web service using Apache is to generate the client stubs:

```
java org.apache.axis.wsdl.WSDL2Java
http://hostname:port/commerce/pricing/calculateOrderPrice?WSDL
```

Next, the following code executes the `calculateOrderPrice` web service:

```
CalculateOrderPriceSEIService webService = new
        CalculateOrderPriceSEIServiceLocator();

CalculateOrderPriceSEI pricingStub =
        webService.getCalculateOrderPriceSEIPort();

OrderPrice price = pricingStub.calculateOrderPrice(myOrderId, null);
```

Promotion Web Services

All promotion web services are included in `commerceWebServices.ear` in the `promotions.war` web application.

Ear file	`commerceWebServices.ear`
War file	`promotions.war`
Context-root	`commerce/promtions`

For an example of a client calling a promotion web service, see the Promotion Web Services Example (page 524) section. For the recommended security policies to be associated with promotion web services, see the Commerce Web Services Security (page 536) section.

This section includes information on the following Promotions Web services.

claimCoupon Web Service

The `claimCoupon` Web service claims the given coupon for the current user.

Servlet Name	`claimCoupon`
Input Parameters	`profileId` - The ID of the customer's profile. `couponClaimCode` – The code of the coupon being claimed. The repository ID of the coupon item is the same as the claim code.
Output	*none*

Web Service Class	`atg.commerce.claimable.ClaimableManager`
Nucleus Component	`/atg/commerce/cliamable/ClaimableManager`
Method	`claimCoupon(String pProfileId, String pCouponClaimCode)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/promotion/claimCoupon?WSDL`
Endpoint URL	`http://hostname:port/commerce/promotion/claimCoupon/claimCoupon`
Security `FunctionalName`	`couponClaims`

getPromotionsAsXML Web Service

The `getPromotionsAsXML` Web service looks up the given profile ID and for each available promotion and returns an xml representation of the promotion using the `GetService` and the `mappingFileName` as configured on `PromotionTools`. Both active promotions in the profile and global promotions are returned.

Servlet Name	`getPromotionsAsxML`
Input Parameters	`profileId` – The ID of the profile for the customer whose promotions will be retrieved.
Output	An XML representation of each active promotion in the profile
Web Service Class	`atg.commerce.promotion.PromotionTools`
Nucleus Component	`/atg/commerce/promotion/PromotionTools`
Method	`getPromotionsAsXML(String pProfileId)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/promotion/getPromotionsAsXML?WSDL`
Endpoint URL	`http://hostname:port/commerce/promotion/getPromotionsAsXML/getPromotionsAsXML`
Security `FunctionalName`	`profileOwnerOperation`
Repository Component	`/atg/commerce/catalog/ProductCatalog`
Item Descriptor	`promotion`

grantPromotion Web Service

The `grantPromotion` Web service grants a promotion to a customer using customer's profile ID and promotion ID.

Servlet Name	`grantPromotion`
Input Parameters	`promotionId` – The ID of the promotion to be granted to the customer. `profileId` – The ID of the of the customer's profile.
Output	*none*
Web Service Class	`atg.commerce.promotion.PromotionTools`
Nucleus Component	`/atg/commerce/promotion/PromotionTools`
Method	`grantPromotion(String pProfileId, String pPromotionId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/promotion/grantPromotion?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/promotion/grantPromotion/grantPromotion`
Security FunctionalName	`promotionManagement`

revokePromotion Web Service

The `revokePromotion` Web service invalidates a customer's eligibility for a promotion using the customer's profile ID and promotion ID.

Servlet Name	`revokePromotion`
Input Parameters	`promotionId` – The ID of the promotion to be granted to the customer. `profileId` – The ID of the of the customer's profile. `removeAllInstances`– If this is set to true and the promotion has more than one instance of the given promotion, all copies of the promotion are removed. Otherwise, only the first copy is removed.
Output	*none*
Web Service Class	`atg.commerce.promotion.PromotionTools`
Nucleus Component	`/atg/commerce/promotion/PromotionTools`

Method	`revokePromotion(String pProfileId, String pPromotionId, boolean pRemoveAllInstances)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/promotion/revokePromotion?WSDL`
Endpoint URL	`http://hostname:port/commerce/promotion/revokePromotion/revokePromotion`
Security FunctionalName	`promotionManagement`

Promotion Web Services Example

The following is an example Apache Axis client calling a promotions web service. The `claimCoupon` web service is used in this example.

The first step in calling a promotion web service using Apache is to generate the client stubs:

```
java org.apache.axis.wsdl.WSDL2Java
http://hostname:port/commerce/promotion/claimCoupon?WSDL
```

Next, the following code executes the `claimCoupon` web service:

```
ClaimCouponSEIService webService = new ClaimCouponSEIServiceLocator();
ClaimCouponSEI couponStub = webService.getClaimCouponSEIPort();
CouponStub.claimCoupon(myProfileId, someCouponClaimCode);
```

Inventory Web Services

All inventory web services are included in `commerceWebServices.ear` in the `inventory.war` web application.

Ear file	`commerceWebServices.ear`
War file	`inventory.war`
Context-root	`commerce/inventory`

For an example of a client calling an inventory web service, see the Inventory Web Services Example (page 527) section. For the recommended security policies to be associated with inventory web services, see the Commerce Web Services Security (page 536) section.

This section includes information on the following Inventory Web services.

getInventory Web Service

This service will return inventory information for each of the given SKUs. The information returned is in a `SimpleInventoryInfo` class.

Servlet name	`getInventory`
Input Parameters	`skuIds` – An array of `skuIds`.
Output	`SimpleInventoryInfo[]` – An array of `SimpleInventoryInfo` objects, one for each SKU that was passed in as input.
Web Service Class	`atg.commerce.inventory.InventoryServices`
Nucleus Component	`/atg/commerce/inventory/InventoryServices`
Method	`getInventory(String[] skuIds) throws InventoryException`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/inventory/getInventory?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/inventory/getInventory/getInventory`
Security `FunctionalName`	`inventory`

getInventoryStatus Web Service

This method will return an inventory status message describing the inventory status of each of the given `skuIds`. The choices are: `inStock`, `outOfStock`, `backorderable`, `preorderable`, or `discontinued`.

Servlet name	`getInventoryStatus`
Input Parameters	`skuIds` – An array of `skuIds`.
Output	An array of Stings (one of the above status messages).
Web Service Class	`atg.commerce.inventory.InventoryServices`

Nucleus Component	`/atg/commerce/inventory/InventoryServices`
Method	`getInventoryStatus(String[] skuIds) throws InventoryException`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/inventory/getInventoryStatus?WSDL`
Endpoint URL	`http://hostname:port/commerce/inventory/getInventoryStatus/getInventoryStatus`
Security `FunctionalName`	`Inventory`

setStockLevels Web Service

This service will update the stock level for each of the given SKUs.

Servlet name	`setStockLevels`
Input Parameters	`skuIds` – An array of SKU IDs. `stockLevels` – An array of new stock level quantities. Both arrays must be the same length.
Output	void
Web Service Class	`atg.commerce.inventory.InventoryServices`
Nucleus Component	`/atg/commerce/inventory/InventoryServices`
Method	`setStockLevels(String[] skuIds, long[] stockLevels)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/inventory/setStockLevels?WSDL`
Endpoint URL	`http://hostname:port/commerce/inventory/setStockLevels/setStockLevels`
Security FunctionalName	`inventoryAdministration`

setStockLevel Web Service

This service will update the stock level for each of the given SKUs.

Servlet name	`setStockLevel`

Input Parameters	`skuId` – The ID of the SKU being updated. `stockLevel` – The new stock level for `skuId`.
Output	void
Web Service Class	`atg.commerce.inventory.InventoryServices`
Nucleus Component	`/atg/commerce/inventory/InventoryServices`
Method	`setStockLevel(String skuId, long stockLevel)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/inventory/setStockLevel?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/inventory/setStockLevel/setStockLevel`
Security `FunctionalName`	`inventoryAdministration`

Inventory Web Services Example

The following is an example Apache Axis client calling an inventory web service. The `getInventory` web service is used in this example.

The first step in calling the inventory web service using Apache is to generate the client stubs:

```
java org.apache.axis.wsdl.WSDL2Java
http://hostname:port/commerce/inventory/getInventory?WSDL
```

Next, the following code executes the `getInventory` web service:

```
GetInventorySEIService webService = new GetInventoryServiceLocator();
GetInventorySEI inventoryStub = webService.getGetInventorySEIPort();
String[] skuIds = { "sku1", "sku2" };
SimpleInventoryInfo[] infos = inventoryStub.getInventory(skuIds);
```

Catalog Web Services

All catalog web services are included in `commerceWebServices.ear` in the `catalog.war` web application.

Ear file	`commerceWebServices.ear`

War file	`catalog.war`
Context-root	`commerce/catalog`

For an example of a client calling a catalog web service, see the Catalog Web Services Example (page 531) section. For the recommended security policies to be associated with catalog web services, see the Commerce Web Services Security (page 536) section.

This section includes information on the following Catalog Web services.

catalogItemViewed Web Service

The `catalogItemViewed` Web service indicates that a particular item was viewed by the current user. A new `ViewItemMessage` will be created and sent.

Servlet name	`catalogItemViewed`
Input Parameters	`profileId` – (Optional) The ID of the user who viewed the item. If null, current user is assumed. `itemId` - The ID of the item. `itemType` - The type of the item viewed (for example: category or product).
Output	none
Web Service Class	`atg.commerce.catalog.CatalogServices`
Nucleus Component	`/atg/commerce/catalog/CatalogServices`
Method	`catalogItemViewed(String pProfileId, String pItemId, String pItemType)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/catalog/catalogItemViewed?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/catalog/catalogItemViewed/catalogItemViewed`
Security `FunctionalName`	`catalog`

getProductSkusXML Web Service

The `getProductSkusXML` Web service retrieves the Product item in XML form as specified by the `ProductId`. Catalog items are retrieved as specified by `CatalogId`. If no catalog item is found, then the user in the current session is accessed and the catalog item is retrieved from the user.

Servlet name	`getProductSkusXML`
Input Parameters	`productId` - Specifies the product to retrieve `catalogId` - (Optional) Catalog item to verify the product. If this parameter is null, then the current user's catalog is used.
Output	SKU items present in the specified product in XML format.
Web Service Class	`atg.commerce.catalog.CatalogServices`
Nucleus Component	`/atg/commerce/catalog/CatalogServices`
Method	`getProductSkusXML(String pProductId, String pCatalogId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/catalog/getProductSkusXML?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/catalog/getProductSkusXML/getProductSkusXML`
Security `FunctionalName`	`Catalog`
Repository Component	`/atg/commerce/catalog/ProductCatalog`
Item Descriptor	`sku`

getProductXMLByDescription Web Service

The `getProductXMLByDescription` Web service retrieves the product whose properties contain `SearchString`, as specified by `SearchPropertyNames` or `getSearchPropertyNames()`.

Servlet name	`getProductXMLByDescription`
Input Parameters	`searchString` - Search string to search for in the properties of Product. `searchPropertyNames` – (Optional) Array of property names to search for. If this is null, then the property names configured at `CatalogServices.searchPropertyNames` will be used. `catalogId` – (Optional) The catalog item to verify the product. If this parameter is null, then the current user's catalog will be used.
Output	Product items in XML format

Web Service Class	`atg.commerce.catalog.CatalogServices`
Nucleus Component	`/atg/commerce/catalog/CatalogServices`
Method	`getProductXMLByDescription(String pSearchString, String[] pSearchPropertyNames, String pCatalogId)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/catalog/getProductXMLByDescription?WSDL`
Endpoint URL	`http://hostname:port/commerce/catalog/getProductXMLByDescription/getProductXMLByDescription`
Security `FunctionalName`	`Catalog`
Repository Component	`/atg/commerce/catalog/ProductCatalog`
Item Descriptor	`product`

getProductXMLById Web Service

The `getProductXMLById` Web service retrieves the Product item in XML form as specified by the `ProductId`. The catalog item is retrieved as specified by `CatalogId`. If no catalog item is found then the user in the current session is accessed and the catalog item is retrieved from the user.

Servlet name	`getProductXMLById`
Input Parameters	`productId` - Specifies the product to retrieve `catalogId` – (Optional) Specifies the `catalogItem` to check for the product. If this parameter is null, then the current users catalog will be used.
Output	Product item in XML form.
Web Service Class	`atg.commerce.catalog.CatalogServices`
Nucleus Component	`/atg/commerce/catalog/CatalogServices`
Method	`getProductXMLById(String pProductId, String pCatalogId)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/catalog/getProductXMLById?WSDL`
Endpoint URL	`http://hostname:port/commerce/catalog/getProductXMLById/getProductXMLById`
Security `FunctionalName`	`catalog`

Repository Component	`/atg/commerce/catalog/ProductCatalog`
Item Descriptor	`product`

getProductXMLByRQL Web Service

The `getProductXMLByRQL` Web service parses the `pRQLQuery` and executes the `RqlStatement` to retrieve the product items. Product repository items are checked against specified catalog to include only items present in catalog.

Servlet name	`getProductXMLByRQL`
Input Parameters	`RQLQuery` - RQL query string to execute against product `RepositoryView`. `catalogId` - (Optional) The catalog item to verify the product. If this parameter is null, then the current users catalog will be used.
Output	An array of strings containing product repository items in XML format.
Web Service Class	`atg.commerce.catalog.CatalogServices`
Nucleus Component	`/atg/commerce/catalog/CatalogServices`
Method	`getProductXMLByRQL(String pRQLQuery, String pCatalogId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/catalog/getProductXMLByRQL?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/catalog/getProductXMLByRQL/getProductXMLByRQL`
Security `FunctionalName`	`Catalog`
Repository Component	`/atg/commerce/catalog/ProductCatalog`
Item Descriptor	`product`

Catalog Web Services Example

The following is an example Apache Axis client calling an catalog web service. The `getProductXMLById` web service is used in this example.

The first step in calling the inventory web service using Apache is to generate the client stubs:

```
java org.apache.axis.wsdl.WSDL2Java
http://hostname:port/commerce/order/getProductXMLById?WSDL
```

Next, the following code executes the `getProductXMLById` web service:

```
GetProductXMLByIdSEIService webService = new
                    GetProductXMLByIdServiceLocator();
GetProductXMLByIdSEI catalogStub =
                    webService.getGetProductXMLByIdSEIPort();
String productXML = catalogStub.getProductXMLById("product1", null);
```

Profile Web Services

All profile web services are included in `commerceWebServices.ear` in the `commerceProfile.war` web application.

Ear file	`commerceWebServices.ear`
War file	`commerceProfile.war`
Context-root	`commerce/commerceProfile`

For an example of a client calling a profile web service, see the Profile Web Services Example (page 536) section. For the recommended security policies to be associated with profile web services, see the Commerce Web Services Security (page 536) section.

This section includes information on the following Commerce Profile Web services.

getDefaultShippingAddress Web Service

The `getDefaultShippingAddress` Web service retrieves the `shippingAddressPropertyName` from the profile.

Servlet Name	`getDefaultShippingAddress`
Input Parameters	`profileId` - The profile ID of the user whose shipping address is returned.

Output	`contactInfo` - Contains the user's default shipping address.
Web Service Class	`atg.commerce.profile.CommerceProfileServices`
Nucleus Component	`/atg/userprofiling/ProfileServices`
Method	`getDefaultShippingAddress(String pProfileId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/userprofiling/getDefaultShippingAddress?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/userprofiling/getDefaultShippingAddress/getDefaultShippingAddress`
Security `FunctionalName`	`profileOwnerOperation`

getDefaultBillingAddress Web Service

The `getDefaultBillingAddress` Web service retrieves the `billingAddressPropertyName` from the profile.

Servlet Name	`getDefaultBillingAddress`
Input Parameters	`profileId` - The profile ID of the user whose billing address is returned.
Output	`contactInfo` - Contains the user's default shopping address.
Web Service Class	`atg.commerce.profile.CommerceProfileServices`
Nucleus Component	`/atg/userprofiling/ProfileServices`
Method	`getDefaultBillingAddress(String pProfileId)`
Executes within a session	Yes
WSDL URL	`http://`*`hostname:port`*`/commerce/userprofiling/getDefaultBillingAddress?WSDL`
Endpoint URL	`http://`*`hostname:port`*`/commerce/userprofiling/getDefaultBillingAddress/getDefaultBillingAddress`
Security `FunctionalName`	`profileOwnerOperation`

getDefaultCreditCard Web Service

The `getDefaultCreditCard` Web service retrieves the `creditCardPropertyName` from the profile.

Servlet Name	`getDefaultCreditCard`
Input Parameters	`profileId` - The ID of the customer profile that contains the credit card.
Output	`BasicCreditCardInfo` - Contains the user's default shopping address.
Web Service Class	`atg.commerce.profile.CommerceProfileServices`
Nucleus Component	`/atg/userprofiling/ProfileServices`
Method	`getDefaultCreditCard(String pProfileId)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/userprofiling/getDefaultCreditCard?WSDL`
Endpoint URL	`http://hostname:port/commerce/userprofiling/getDefaultCreditCard/getDefaultCreditCard`
Security `FunctionalName`	`profileOwnerOperation`

setDefaultBillingAddress Web Service

The `setDefaultBillingAddress` Web service sets the user's `billingAddressPropertyName` to the given address

Servlet Name	`setDefaultBillingAddress`
Input Parameters	`profileId` - The ID of the customer profile to be changed. `address` - The new billing address.
Output	The ID of the newly created address item
Web Service Class	`atg.commerce.profile.CommerceProfileServices`
Nucleus Component	`/atg/userprofiling/ProfileServices`
Method	`setDefaultBillingAddress(String pProfileId, ContactInfo pAddress)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/userprofiling/setDefaultBillingAddress?WSDL`
Endpoint URL	`http://hostname:port/commerce/userprofiling/setDefaultBillingAddress/setDefaultBillingAddress`

Security FunctionalName	profileOwnerOperation

setDefaultCreditCard Web Service

The setDefaultCreditCard Web service sets the user's creditCardPropertyName to the given address.

Servlet Name	setDefaultCreditCard
Input Parameters	profileId - The ID of the customer profile to be changed. creditCard - The new credit card.
Output	The ID of the newly created credit card item.
Web Service Class	atg.commerce.profile.CommerceProfileServices
Nucleus Component	/atg/userprofiling/ProfileServices
Method	setDefaultCreditCard(String pProfileId, BasicCreditCardInfoImpl pCreditCardInfo)
Executes within a session	Yes
WSDL URL	http://*hostname:port*/commerce/userprofiling/setDefaultCreditCard?WSDL
Endpoint URL	http://*hostname:port*/commerce/userprofiling/setDefaultCreditCard/setDefaultCreditCard
Security FunctionalName	profileOwnerOperation

setDefaultShippingAddress Web Service

The setDefaultShippingAddress Web service sets the user's shippingAddressPropertyName to the given address

Servlet Name	setDefaultShippingAddress
Input Parameters	profileId - The ID of the customer profile to be changed. address - The new shipping address.
Output	The ID of the newly created shipping address item.
Web Service Class	atg.commerce.profile.CommerceProfileServices
Nucleus Component	/atg/userprofiling/ProfileServices

Method	`setDefaultShippingAddress(String pProfileId, ContactInfo pAddress)`
Executes within a session	Yes
WSDL URL	`http://hostname:port/commerce/userprofiling/setDefaultShippingAddress?WSDL`
Endpoint URL	`http://hostname:port/commerce/userprofiling/setDefaultShippingAddress/setDefaultShippingAddress`
Security `FunctionalName`	`profileOwnerOperation`

Profile Web Services Example

The following is an example Apache Axis client calling a profile web service. The `getDefaultShippingAddress` web service is used in this example.

The first step in calling the profile web service using Apache is to generate the client stubs:

```
java org.apache.axis.wsdl.WSDL2Java
http://hostname:port/commerce/profile/getDefaultShippingAddress?WSDL
```

Next, the following code executes the `getDefaultShippingAddress` web service:

```
GetDefaultShippingAddressSEIService webService = new
GetDefaultShippingAddressSEIServiceLocator();
GetDefaultShippingAddressSEI addressStub =
webService.getGetDefaultShippingAddressSEIPort();
OrderPrice price = pricingStub.calculateOrderPrice(myOrderId, null);
```

Commerce Web Services Security

Web service security is controlled by the security policies associated with the Web service's security functional name. For more information on general web service security see the *Web Service Security* section of the *Creating Custom Web Services* chapter in the *ATG Web Services and Integration Framework Guide*.

The security functional name for each Web service is included in the sections about each web services in this chapter. The standard security policy is described in the *Managing Access Control* chapter of the *ATG Platform Programming GuideATG Platform Programming Guide*. The `ProfileOwnerPolicy` and `RelativeRoleByProfileOrgPolicy` are described in the *Profile-Related Security Policies* section in the *Web Services for Personalization and Scenarios* chapter of the *ATG Personalization Programming Guide*. The `OrderOwnerPolicy` is described in the Using the Order Owner Security Policy (page 537) section of this chapter.

The following table lists the recommended security policy for each security functional name.

Security Functional Name	Recommended Security Policy
`catalog`	Standard security policy with an ACL that lists everyone that is allowed to view the catalog. This ACL will probably include all users.
`couponClaims`	`ProfileOwnerPolicy`
`getCurrentOrderId`	none recommended
`inventory`	Standard `SecurityPolicy` with an ACL that lists those that are allowed to call the inventory services. This ACL will probably include all users.
`inventoryAdministration`	Standard `SecurityPolicy` with an ACL that lists all users that are allowed to manage inventory.
`itemPricing`	`ProfileOwnerPolicy`
`orderCreation`	Standard `SecurityPolicy` with ACL that lists users that are allowed to create orders.
`orderCreationForUser`	Standard `SecurityPolicy` with ACL that lists users that are allowed to create orders for other users such as administrators and customer service representatives. Another option is to use the `RelativeRoleByProfileOrgPolicy` to define access relative to a user's organization.
`orderLookupOperation`	`ProfileOwnerPolicy`
`orderManagement`	`OrderOwnerPolicy`
`orderPricing`	`OrderOwnerPolicy`
`profileOwnerOperation`	`ProfileOwnerPolicy`
`profileOwnerOperation`	`ProfileOwnerPolicy`
`promotionManagement`	`ProfileOwnerPolicy`

Using the Order Owner Security Policy

The Order Owner Security Policy extends the Standard Security Policy, which has all the base functionality for interpreting the Access Control Lists (ACL). ACLs grant or deny access to secure objects. The `atg.security.StandardSecurityPolicy` class is provided as part of the Oracle ATG Web Commerce platform. For more information on the Standard Security Policy, see the *Managing Access Control* chapter of the *ATG Platform Programming GuideATG Platform Programming Guide*.

The Order Owner Security Policy appends the ACL returned by Standard Security Policy with additional ACLs that either grant or deny access to specific personas. Personas can be users, roles or organizations. The Order Owner Security Policy appends the ACL with the persona of the order owner. The order object is an incoming method parameter.

Appendix B. Oracle ATG Web Commerce Databases

The Oracle ATG Web Commerce database schema includes the following types of tables:

Core Oracle ATG Web Commerce Functionality Tables

The following sections describe the database tables specific to core Oracle ATG Web Commerce functionality:

- Product Catalog Tables (page 540)

Product Catalog Tables

Oracle ATG Web Commerce uses the following tables to store product catalog information:

dcs_allroot_cats

This table contains a list of all root categories

Column	Data Type	Constraint
`catalog_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog. References `dcs_catalog(catalog_id)`.	
`root_cat_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`.	

dcs_cat_anc_cats

This table defines the ancestor categories of each category. Used by the `ancestorCategories` property of the category item.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	not null
(primary key)	The ID of the category whose ancestor category is defined by the `anc_category_id` column.	
`sequence_num`	INTEGER	not null
(primary key)	Sequence number used to differentiate rows that are otherwise the same. An ancestor can appear more than once for a given category if there is more than one way to traverse up the catalog tree to reach the ancestor.	
`anc_category_id`	VARCHAR(40)	not null
	The ID of the ancestor category of the category defined by the `category_id` column.	

dcs_cat_ancestors

This table contains information about category ancestors.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category.	
`anc_cat_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of a category that is an ancestor in the category. A category has multiple rows in this table representing all the ancestors of a category. A query of this value determines whether a category is a child of another category.	

dcs_cat_aux_media

This table contains information about an auxiliary media image associated with a category.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`	
`tag`	VARCHAR(42)	NOT NULL

Column	Data Type	Constraint
(primary key)	Represents the key.	
`media_id`	VARCHAR(40)	NOT NULL
	The value that points to a media item. References `dcs_media(media_id)`	

dcs_cat_catalogs

This table defines all the catalogs that a given category can be viewed in. Used by the `catalogs` property of the `category` item.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	not null
(primary key)	The ID of the category that can be viewed in the catalog defined by the `catalog_id` column	
`catalog_id`	VARCHAR(40)	not null
(primary key)	The ID of the catalog within which the category defined in the `category_id` column can be viewed	

dcs_cat_catinfo

This table contains information about a specific category in the catalog.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`.	
`catalog_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog.	
`category_info_id`	VARCHAR(40)	NOT NULL
	Unique identifier of the `categoryInfo` to associate with this category when it is viewed as part of this catalog.	

dcs_cat_groups

This table contains information about groups in a category.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`	
`child_prd_group`	VARCHAR(254)	NULL
	Stores the name of a Content Group that should return a list of products that should be children of this category.	
`child_cat_group`	VARCHAR(254)	NULL
	Stores the name of a Content Group that should return a list of categories that should be children of this category.	
`related_cat_group`	VARCHAR(254)	NULL
	Stores the name of a Content Group that should return a list of categories that are related to this category.	

dcs_cat_chldcat

This table contains information about children of categories.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Used to order rows in this table.	
`child_cat_id`	VARCHAR(40)	NOT NULL
	A category ID that should be considered a child of this category. References `dcs_category(category_id)`	

dcs_cat_chldprd

This table contains information about child products within a category.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Used to order rows in the table.	
`child_prd_id`	VARCHAR(40)	NOT NULL
	A product ID that should be considered a child of this category. References `dcs_product(product_id)`	

dcs_cat_keywrds

This table contains information about category keywords.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Used to order rows in this table.	
`Keyword`	VARCHAR(254)	NOT NULL
	A String value used in searches.	

dcs_cat_media

This table contains information about media in a category.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the category. References `dcs_category(category_id)`	
`template_id`	VARCHAR(40)	NULL

Column	Data Type	Constraint
	The ID of a media item that represents the template that renders the category. References `dcs_media(media_id)`	
`thumbnail_image_id`	VARCHAR(40)	NULL References `dcs_media(media_id)`
	The ID of a media item that represents a thumbnail image of a category that can be displayed in the template.	
`small_image_id`	VARCHAR(40)	NULL
	The ID of a media item that represents a small image of a category that can be displayed in the template. References `dcs_media(media_id)`	
`large_image_id`	VARCHAR(40)	NULL
	The ID of a media item that represents a large image of a category that can be displayed in the template. References `dcs_media(media_id)`	

dcs_cat_prnt_cats

This table stores information related to a catalog's parent categories.

Column	Data Type	Constraint
`Category_id`	VARCHAR(40)	not null
(primary key)	The ID of a category	
`Catlog_id`	VARCHAR(40)	not null
(primary key)	The ID of a catalog.	
`parent_ctgy_id`	VARCHAR(40)	not null
	The ID of the catalog's parent category.	

dcs_cat_rltdcat

This table contains information about category relationships.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category.	

Column	Data Type	Constraint
sequence_num	INTEGER	NOT NULL
(primary key)	Used to order rows in this table.	
related_cat_id	VARCHAR(40)	NOT NULL
	A category ID that should be considered related to the category. References dcs_category(category_id)	

dcs_cat_subcats

This table contains information about the sub-categories for a specific category in the catalog.

Column	Data Type	Constraint
category_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References dcs_category(category_id).	
sequence_num	INTEGER	NOT NULL
(primary key)	This number is used to order the sub catalogs.	
catalog_id	VARCHAR(40)	NOT NULL
	The unique identifier associated with the catalog. References dcs_catalog(catalog_id).	

dcs_cat_subroots

This table contains information about the root subcategories for a specific category in the catalog.

Column	Data Type	Constraint
category_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References dcs_category(category_id).	
sequence_num	INTEGER	NOT NULL
(primary key)	This number is used to order the sub catalogs.	
sub_category_id	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
	The unique identifier associated with the subcategory.	

dcs_catalog

This table contains information that describes catalogs.

Column	Data Type	Constraint
`catalog_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`display_name`	VARCHAR(254)	NULL
	The name of the catalog displayed in the ACC.	
`creation_date`	TIMESTAMP	NULL
	The date the catalog was created.	
`last_mod_date`	TIMESTAMP	NULL
	The last date the catalog was modified.	
`migration_status`	INTEGER	NULL
	If the catalog was migrated from a standard product, this represents the status of the migration.	
`migration_index`	INTEGER	NULL
	This is the index of the last successful migration step. Used if `CatalogMigration` needs to restart.	
`item_acl`	LONG VARCHAR	NULL
	The security for the catalog.	

dcs_catalog_sites

This table stores information related to site ownership of catalogs. Used by the multisite feature.

Column	Data Type	Constraint
catalog_id	VARCHAR(40)	not null
(primary key)	The ID of a catalog.	
site_id	VARCHAR(40)	not null
(primary key)	The ID of a site to which the catalog is associated.	

dcs_category

This table contains information that describes a category.

Column	Data Type	Constraint
category_id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the category.	
catalog_id	VARCHAR(40)	
	The unique identifier associated with the catalog that contains this category.	
version	INTEGER	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
creation_date	DATE	NULL
	The date this category was created.	
start_date	DATE	NULL
	The date on which this category will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
end_date	DATE	NULL
	The last date on which this category will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
display_name	VARCHAR(254)	NULL
	The name of the category that displays in the ACC.	
description	VARCHAR(254)	NULL

Column	Data Type	Constraint
	A text description of the category.	
`long_description`	LONG VARCHAR	NULL
	A long text description of the category.	
`parent_cat_id`	VARCHAR(40)	NULL
	The ID of the immediate and default parent category. References `dcs_category(category_id)`	
`parent_cat_id`	VARCHAR(40)	NULL
	The ID of the immediate and default parent category.	
`category_type`	INTEGER	NULL
	An enumerated value used for defining sub-types of the initial Oracle ATG Web Commerce category item descriptor.	
`root_category`	NUMERIC(1)	NULL CHECK`(root_category in (01))`
	A boolean (1 or 0) indicator that represents which categories should be considered the root of the catalog hierarchy.	

dcs_category_info

This table contains information about categories in catalogs.

Column	Data Type	Constraint
`category_info_id`	VARCHAR(40)	NOT NULL
(primary key)	Unique identifier associated with the `categoryInfo` object.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`item_acl`	LONG VARCHAR	NULL
	The security for the `category_info`.	

dcs_category_sites

This table stores information related to site ownership of categories. Used by the multisite feature.

Column	Data Type	Constraint
`category_id`	VARCHAR(40)	not null
(primary key)	The ID of a category.	
`site_id`	VARCHAR(40)	not null
(primary key)	The ID of a site to which the category is associated.	

dcs_catinfo_anc

This table contains information about ancestor categories for `categoryInfo` objects.

Column	Data Type	Constraint
`category_info_id`	VARCHAR(40)	NOT NULL
(primary key)	Identifier for the `categoryInfo`.	
`anc_cat_id`	VARCHAR(40)	NOT NULL
(primary key)	Identifier for the ancestor category.	

dcs_catfol_chld

This table contains information about the folders in which catalogs are located.

Column	Data Type	Constraint
`catfol_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcs_gen_fol_cat(folder_id)`.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	This number is used to order the catalogs.	
`catalog_id`	VARCHAR(40)	NOT NULL
	The ID of the catalog contained in the folder.	

dcs_catfol_sites

This table contains information about the sites with which catalog folders are associated.

Column	Data Type	Constraint
catfol_id	VARCHAR(40)	NOT NULL
(primary key)	References dcs_gen_fol_cat(folder_id).	
site_id	VARCHAR(40)	NOT NULL
(primary key)	References the ID of the site with with the catalog folder is associated.	

dcs_child_fol_cat

This table contains information about the child folders for a specific catalog.

Column	Data Type	Constraint
folder_id	VARCHAR(40)	NOT NULL
(primary key)	References dcs_gen_fol_cat(folder_id).	
sequence_num	INTEGER	NOT NULL,
(primary key)	This number is used to order the child folders.	
child_folder_id	VARCHAR(40)	NOT NULL,
	The ID of the child folder.	

dcs_conf_options

The following table contains information related to configuration options.

Column	Data Type	Constraint
config_prop_id	VARCHAR(40)	NOT NULL
(primary key)	References dcs_config_prop(config_prop_id)	
config_options	VARCHAR(40)	NOT NULL
	The configuration options associated with the configurable property.	
sequence_num	INTEGER	NOT NULL
(primary key)	The sequence number of the configurable option in a list.	

dcs_config_prop

The following table contains information related to configurable properties.

Column	Data Type	Constraint
`config_prop_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository configurable property ID.	
`version`	INTEGER	NOT NULL
	The repository version number.	
`display_name`	VARCHAR(40)	NULL
	The display name of the configurable property.	
`description`	VARCHAR(255)	NULL
	A description of the configurable property.	
`item_acl`	LONG VARCHAR	NULL
	The item access control list for this item.	

dcs_config_opt

The following table contains information related to configuration options.

Column	Data Type	Constraint
`config_opt_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository configuration option ID.	
`Version`	INTEGER	NOT NULL
	The repository version number.	
`display_name`	VARCHAR(40)	NULL
	The configuration option display name.	
`description`	VARCHAR(255)	NULL
	The configuration option description.	
`sku`	VARCHAR(40)	NULL
	The configuration option SKU.	

Column	Data Type	Constraint
`product`	VARCHAR(40)	NULL
	The configuration option product.	
`price`	DOUBLE PRECISION	NULL
	The configuration option price.	
`item_acl`	LONG VARCHAR	NULL
	The item access control list for this item.	

dcs_ctlg_anc_cats

This table contains information that defines the ancestor categories of each catalog. Used for the `ancestorCategories` property of the catalog item.

Column	Data Type	Constraint
`catalog_id`	VARCHAR(40)	not null
(primary key)	The ID of the catalog whose ancestor category is defined by the `anc_category_id` column.	
`sequence_num`	INTEGER	not null
(primary key)	Sequence number used to differentiate rows that are otherwise the same. An ancestor can appear more than once for a given catalog if there is more than one way to traverse up the catalog tree to reach the ancestor.	
`category_id`	VARCHAR(40)	not null
	The ID of the ancestor category of the catalog defined in the `catalog_id` column.	

dcs_dir_anc_ctlgs

This table contains information that defines a direct-ancestor relationship between two catalogs. A catalog is considered a direct ancestor of another catalog if there are no categories separating the two catalogs. In other words, the tree between the two catalogs will only include catalogs.

A "self" ancestor relationship is defined in this database table. Each catalog should have a row where the ancestor catalog is itself in the `directAncestorCatalogsAndSelf` property of the catalog item.

Column	Data Type	Constraint
`catalog_id`	VARCHAR(40)	not null

Column	Data Type	Constraint
(primary key)	The ID of the catalog whose direct ancestor is defined in the `anc_catalog_id` column.	
`sequence_num`	INTEGER	not null
(primary key)	Sequence number used to differentiate rows that are otherwise the same – an ancestor can appear more than once for a given catalog if there is more than one way to traverse up the catalog tree to reach the ancestor.	
`anc_catalog_id`	VARCHAR(40)	not null
	The direct ancestor catalog of the catalog defined in the `catalog_id` column.	

dcs_folder

This table contains information that describes a folder.

Column	Data Type	Constraint
`folder_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the folder.	
`version`	INTEGER	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce platform, you should also increment the version number.	
`creation_date`	DATE	NULL
	The date this folder was created.	
`start_date`	DATE	NULL
	The date on which this folder will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`end_date`	DATE	NULL
	The last date on which this folder will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`description`	VARCHAR(254)	NULL
	A text description of the folder.	

Column	Data Type	Constraint
name	VARCHAR(254)	NOT NULL
	The name of the folder.	
path	VARCHAR(254)	NOT NULL
	A String that represents the folder in the context of all the ancestor folders. This value is similar to the complete absolute path of a file.	
parent_folder_id	VARCHAR(40)	NOT NULL
	The ID of the folder that contains this folder in the catalog hierarchy. References dcs_folder(folder_id).	

dcs_foreign_cat

The following table contains information related to a foreign catalog (remote catalog) that Oracle ATG Web Commerce would integrate with to support configurable commerce items.

Column	Data Type	Constraint
catalog_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the catalog.	
type	INTEGER	NOT NULL
	The type field is used for sub-typing this catalog.	
version	INTEGER	NOT NULL
	Version property that is used internally by the Repository to maintain data consistency.	
name	VARCHAR(100)	NULL
	The name for this catalog that will appear in the ACC.	
description	VARCHAR(255)	NULL
	A description of this catalog that is appropriate in a UI context.	
host	VARCHAR(100)	NULL
	The host that this foreign catalog lives at.	
port	INTEGER	NULL
	The port that this catalog can be located on at the host.	

Column	Data Type	Constraint
base_url	VARCHAR(255)	NULL
	The base URL to locate this catalog, at a given host.	
return_url	VARCHAR(255)	NULL
	The URL that can be used for return.	
item_acl	LONG VARCHAR	NULL
	Maintains an ACL for security information purposes on instances of this item-descriptor.	

dcs_gen_fol_cat

This table contains information about the base folders for a specific catalog.

Column	Data Type	Constraint
folder_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the folder item.	
type	INTEGER	NOT NULL
	The type of folder.	
name	VARCHAR(40)	NOT NULL
	The name of the folder that is displayed in the ACC.	
parent	VARCHAR(40)	NULL
	The parent folder of the folder described in this table.	
description	VARCHAR(254)	NULL
	A description of this folder.	
item_acl	LONG VARCHAR	NULL
	Security information for this folder.	

dcs_ind_anc_ctlgs

This table contains information that defines an indirect-ancestor relationship between two catalogs. A catalog is considered an indirect ancestor of another catalog if there is at least one category separating the two catalogs. This table defines the `indirectAncestorCatalogs` property of the catalog item.

Column	Data Type	Constraint
catalog_id	VARCHAR(40)	not null
(primary key)	The ID of the catalog whose indirect ancestor is defined by the anc_catalog_id column.	
sequence_num	INTEGER	not null
(primary key)	Sequence number used to differentiate rows that are otherwise the same – an ancestor can appear more than once for a given catalog if there is more than one way to traverse up the catalog tree to reach the ancestor.	
anc_catalog_id	VARCHAR(40)	not null
	The ID of the indirect ancestor catalog of the catalog defined by the catalog_id column.	

dcs_media

This table contains information that describes a media item.

Column	Data Type	Constraint
media_id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the media item.	
version	INTEGER	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
creation_date	DATE	NULL
	The date this media item was created.	
start_date	DATE	NULL
	The date on which this media item will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
end_date	DATE	NULL
	The last date on which this media item will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	

Column	Data Type	Constraint
`description`	VARCHAR(254)	NULL
	A text description of the media item.	
`name`	VARCHAR(254)	NOT NULL
	The name of the media item.	
`path`	VARCHAR(254)	NOT NULL
	A String, which represents the folder in the context of all the ancestor folders. This value is similar to the complete absolute path of a file.	
`parent_folder_id`	VARCHAR(40)	NOT NULL
	The ID of the folder that contains this media item in the catalog hierarchy. References `dcs_folder(folder_id)`	
`media_type`	INTEGER	NULL
	Used as an enumerated data type that indicates what form of media is stored. By default this includes external, internal binary, and internal text media items.	

dcs_media_ext

This table contains information that describes extended attributes of a media item.

Column	Data Type	Constraint
`media_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the media. References `dcs_media(media_id)`.	
`url`	VARCHAR(254)	NOT NULL
	The external URL that references media content.	

dcs_media_bin

This table contains information that describes the size of a media item.

Column	Data Type	Constraint
`media_id`	VARCHAR(40)	NOT NULL UNIQUE

Column	Data Type	Constraint
(primary key)	The unique identifier associated with the media. References `dcs_media(media_id)`.	
`length`	INTEGER	NOT NULL
	The number of bytes of data stored in the data column.	
`last_modified`	DATE	NOT NULL
	The date this item was last modified.	
`data`	LONG VARBINARY	NOT NULL
	The raw bytes of content.	

dcs_media_txt

This table contains information about the text features of a media item.

Column	Data Type	Constraint
`media_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the media. References `dcs_media(media_id)`.	
`length`	INTEGER	NOT NULL
	The number of bytes of data stored in the data column.	
`last_modified`	DATE	NOT NULL
	The date this item was last modified.	
`data`	LONG VARCHAR	NOT NULL
	Text content that can be indexed by a search engine.	

dcs_prd_anc_cats

This table defines the ancestor categories of each product. Used by the `ancestorCategories` property of the product item.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	not null

Column	Data Type	Constraint
(primary key)	The ID of the product whose ancestor category is defined in the `category_id` column.	
`sequence_num`	INTEGER	not null
(primary key)	Sequence number used to differentiate rows that are otherwise the same. An ancestor can appear more than once for a given product if there is more than one way to traverse up the catalog tree to reach the ancestor.	
`category_id`	VARCHAR(40)	not null
	The ancestor category of the product defined in the `product_id` column.	

dcs_prd_ancestors

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product.	
`anc_cat_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of a category that is an ancestor of the product. A product has multiple rows in this table representing all the ancestors of the product. . A query of this value determines whether a product is a child of another category.	

dcs_prd_aux_media

This table contains information about an auxiliary media image associated with a product.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`	
`tag`	VARCHAR(42)	NOT NULL
(primary key)	Represents the key.	
`media_id`	VARCHAR(40)	NOT NULL
	The value that points to a media item. References `dcs_media(media_id)`	

dcs_prd_catalogs

This table defines all the catalogs that a given product can be viewed in. Used by the `catalogs` property of the `product` item.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	not null
(primary key)	The ID of the product that can be viewed in the catalog defined by the `catalog_id` column	
`catalog_id`	VARCHAR(40)	not null
(primary key)	The ID of the catalog within which the product defined by the `product_id` column can be viewed	

dcs_prd_keywrds

This table contains information about product keywords.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Used to order rows in this table.	
`keyword`	VARCHAR(254)	NOT NULL
	A String value used in searches.	

dcs_prd_media

This table contains information about product media items.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`	

Column	Data Type	Constraint
template_id	VARCHAR(40)	NULL
	The ID of a media item that represents the template that renders the product. References dcs_media(media_id)	
thumbnail_image_id	VARCHAR(40)	NULL
	The ID of a media item that represents a thumbnail image of a product that can be displayed in the template. References dcs_media(media_id)	
small_image_id	VARCHAR(40)	NULL
	The ID of a media item that represents a small image of a category that can be displayed in the template. References dcs_media(media_id)	
large_image_id	VARCHAR(40)	NULL
	The ID of a media item that represents a large image of a category that can be displayed in the template. References dcs_media(media_id)	

dcs_prd_chldsku

This table contains information about children of products.

Column	Data Type	Constraint
product_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product. References dcs_product(product_id)	
sequence_num	INTEGER	NOT NULL
(primary key)	Used to order rows in the table.	
sku_id	VARCHAR(40)	NOT NULL
	The ID of a SKU that is a child of this product. References dcs_sku(sku_id)	

dcs_prd_groups

This table contains information about related product groups.

Column	Data Type	Constraint
product_id	VARCHAR(40)	NOT NULL UNIQUE

Column	Data Type	Constraint
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`	
`related_prd_group`	VARCHAR(254)	NULL
	Stores the name of a Content Group that should return a list of products that are related to this product.	

dcs_prd_prdinfo

This table contains information about the `productInfos` to associate with a given product in a given catalog.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`.	
`catalog_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with a catalog.	
`product_info_id`	VARCHAR(40)	NOT NULL
	The unique identifier of the `productInfo` to associate with this product when it is viewed from this catalog.	

dcs_prd_prnt_cats

This table defines the parent category for each product within each catalog. It defines the `parentCategoriesForCatalog` property of the product item.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	not null
(primary key)	The ID of the product whose parent category in the catalog defined by the `catalog_id` column is defined in the `category_id` column.	
`catalog_id`	VARCHAR(40)	not null
(primary key)	The ID of the catalog within which the category defined by the `category_id` column is the parent category of the product defined in the `product_id` column.	
`category_id`	VARCHAR(40)	not null

Column	Data Type	Constraint
	The ID of the parent category of the product defined by the `product_id` column within the catalog defined by the `catalog_id` column.	

dcs_prd_rltdprd

This table contains information about related products.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`	
`sequence_num`	INTEGER	NOT NULL
	Used to order rows in the table.	
`related_prd_id`	VARCHAR(40)	NOT NULL
	A product ID that should be considered related to the product. References `dcs_product(product_id)`	

dcs_prd_skuattr

This table contains information about which attributes of a SKU should be displayed.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product. References `dcs_product(product_id)`	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Used to order rows in the table.	
`attribute_name`	VARCHAR(40)	NOT NULL
	The name of a SKU property that should be used to display relevant information about a SKU. For example not all the attributes of a SKU need to be displayed, perhaps only color, size, and display-name are necessary.	

dcs_prdinfo_anc

This table contains information about the ancestor categories to associate with `productInfos`.

Column	Data Type	Constraint
`product_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the `productInfo` item.	
`anc_cat_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the ancestor category.	

dcs_prdinfo_rdprd

This table contains information about related products.

Column	Data Type	Constraint
`product_info_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcs_product_info(product_info_id)`.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	This number is used to order the related products.	
`related_prd_id`	VARCHAR(40)	NOT NULL
	References `dcs_product(product_id)`.	

dcs_product

This table contains information that describes a product item.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the product.	
`version`	INTEGER	NOT NULL

Column	Data Type	Constraint
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
`creation_date`	DATE	NULL
	The date this product was created.	
`start_date`	DATE	NULL
	The date on which this product will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`end_date`	DATE	NULL
	The last date on which this product will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`display_name`	VARCHAR(254)	NULL
	The name of the product that displays in the ACC.	
`description`	VARCHAR(254)	NULL
	A text description of the product.	
`long_description`	LONG VARCHAR	NULL
	A long text description of the product.	
`parent_cat_id`	VARCHAR(40)	NULL
	The ID of the immediate and default parent category. References `dcs_category(category_id)`	
`parent_cat_id`	VARCHAR(40)	NULL
	The ID of the immediate and default parent category.	
`product_type`	INTEGER	NULL
	An enumerated value used for defining sub-types of the initial Oracle ATG Web Commerce product item descriptor.	
`manufacturer`	VARCHAR 40	NULL
	A reference to the manufacturer of this particular product. References `dcs_manufacturer(manufacturer_id)`.	

dcs_product_acl

The table stores security information for each product repository item.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the product.	
`item_acl`	LONG VARCHAR	NULL
	Stores the actual access control list. The access control list defines who can read, edit and delete an item.	

dcs_product_info

This table contains information about the products in catalogs.

Column	Data Type	Constraint
`product_info_id`	VARCHAR(40)	NOT NULL
(primary key)	Unique identifier associated with the `productInfo` object.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`parent_cat_id`	VARCHAR(40)	NULL
	Identifier for the parent category of the `productInfo`.	
`item_acl`	LONG VARCHAR	NULL
	The security for the `product_info`.	

dcs_product_sites

This table stores information related to site ownership of catalogs. Used by the multisite feature.

Column	Data Type	Constraint
`product_id`	VARCHAR(40)	not null
(primary key)	The ID of a product.	

Column	Data Type	Constraint
site_id	VARCHAR(40)	not null
(primary key)	The ID of a site to which the product is associated.	

dcs_root_cats

This table contains a list of root categories.

Column	Data Type	Constraint
catalog_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog. References dcs_catalog(catalog_id).	
root_cat_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the category. References dcs_category(category_id).	

dcs_root_subcats

This table contains a list of root sub-catalogs.

Column	Data Type	Constraint
catalog_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog. References dcs_catalog(catalog_id)	
sub_catalog_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the subcatalog. References dcs_catalog(catalog_id).	

dcs_sku

This table contains information that describes a SKU item.

Column	Data Type	Constraint
sku_id	VARCHAR(40)	NOT NULL UNIQUE

Column	Data Type	Constraint
(primary key)	The unique identifier associated with the SKU.	
`version`	INTEGER	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
`creation_date`	DATE	NULL
	The date this SKU was created.	
`start_date`	DATE	NULL
	The date on which this SKU will become available. This optional field can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`end_date`	DATE	NULL
	The last date on which this SKU will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`display_name`	VARCHAR(254)	NULL
	The name of the SKU that displays in the ACC.	
`description`	VARCHAR(254)	NULL
	A text description of the SKU.	
`sku_type`	INTEGER	NULL
	An enumerated value used for defining sub-types of the initial Oracle ATG Web Commerce SKU item descriptor.	
`wholesale_price`	DOUBLE PRECISION	NULL
	The wholesale price of the SKU.	
`list_price`	DOUBLE PRECISION	NULL
	The list price of the SKU.	
`sale_price`	DOUBLE PRECISION	NULL
	The sale price of the SKU.	
`on_sale`	NUMERIC(1)	NULL CHECK`(on_sale in (01))`
	Determines whether the SKU is on sale.	

Column	Data Type	Constraint
tax_status	INTEGER	NULL
	An optional field that may be used to determine the taxable status of the SKU.	
fulfiller	INTEGER	NULL
	An enumerated value that indicates which Oracle ATG Web Commerce fulfiller should attempt to process the SKU in the submitted order.	
manuf_part_num	WVARCHAR(254)	NULL
	A String property that represents the manufacturers part number for this SKU.	

dcs_sku_attr

This table holds information about an attribute map associated with a SKU.

Column	Data Type	Constraint
sku_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the SKU. References dcs_sku(sku_id)	
attribute_name	VARCHAR(42)	NOT NULL
(primary key)	Represents the key.	
attribute_value	VARCHAR(254)	NOT NULL
	The value that allows arbitrary name property values to be associated with a SKU.	

dcs_sku_aux_media

This table contains information about an auxiliary media image associated with a SKU.

Column	Data Type	Constraint
sku_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the SKU. References dcs_sku(sku_id)	

Column	Data Type	Constraint
tag	VARCHAR(42)	NOT NULL
(primary key)	Represents the key.	
media_id	VARCHAR(40)	NOT NULL
	The value that points to a SKU. References dcs_media(media_id)	

dcs_sku_catalogs

This table defines all the catalogs that a given SKU can be viewed in. Used by the catalogs property of the sku item.

Column	Data Type	Constraint
sku_id	VARCHAR(40)	not null
(primary key)	The ID of the SKU that can be viewed within the catalog defined by the catalog_id column	
catalog_id	VARCHAR(40)	not null
(primary key)	The ID of the catalog within which the SKU defined by the sku_id column can be viewed	

dcs_sku_link

This table describes SKU links, which are used to represent SKU bundles.

Column	Data Type	Constraint
sku_link_id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier of the SKU link, which represents a bundle of SKUs.	
version	INTEGER	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
creation_date	DATE	NULL
	The date this SKU was created.	

Column	Data Type	Constraint
start_date	DATE	NULL
	The date on which this SKU will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
end_date	DATE	NULL
	The last date on which this SKU will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
display_name	VARCHAR(254)	NULL
	The name of the SKU link that displays in the ACC.	
description	VARCHAR(254)	NULL
	A text description of the SKU.	
quantity	INTEGER	NOT NULL
	The number of items to include in the bundle.	
bundle_item	VARCHAR(40)	NOT NULL
	The specific SKU to include in the bundle. References dcs_sku(sku_id)	

dcs_sku_bndllnk

This table contains information that associated SKU links with SKU objects.

Column	Data Type	Constraint
sku_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the SKU. References dcs_sku(sku_id)	
sequence_num	INTEGER	NOT NULL
(primary key)	Used to order rows in the table.	
sku_link_id	VARCHAR(40)	NOT NULL
	The ID of the SKU link that should be included in the SKU bundle. References dcs_sku_link(sku_link_id)	

dcs_sku_conf

The following table contains information related to configurable SKUs.

Column	Data Type	Constraint
`sku_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcs_sku(sku_id)`.	
`config_props`	VARCHAR(40)	NOT NULL
	The configurable properties associated with this configurable SKU.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	The sequence number of the configurable properties in a list.	

dcs_sku_media

This table contains information about SKU media items.

Column	Data Type	Constraint
`sku_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the SKU. References `dcs_sku(sku_id)`	
`template_id`	VARCHAR(40)	NULL
	The ID of a SKU that represents the template that renders the category. References `dcs_media(media_id)`	
`thumbnail_image_id`	VARCHAR(40)	NULL
	The ID of a media item that represents a thumbnail image of a category that can be displayed in the template. References `dcs_media(media_id)`	
`small_image_id`	VARCHAR(40)	NULL
	The ID of a media item that represents a small image of a product that can be displayed in the template. References `dcs_media(media_id)`	
`large_image_id`	VARCHAR(40)	NULL
	The ID of a media item that represents a large image of a SKU that can be displayed in the template. References `dcs_media(media_id)`	

dcs_sku_replace

This table contains information about replacing SKUs.

Column	Data Type	Constraint
`sku_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the SKU. References `dcs_sku(sku_id)`	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Used to order the rows in the table.	
`replacement`	VARCHAR(40)	NOT NULL
	The ID of a SKU that should be used as a replacement for another SKU if it is not available for purchase.	

dcs_sku_sites

This table stores information related to site ownership of catalogs. Used by the multisite feature.

Column	Data Type	Constraint
`sku_id`	VARCHAR(40)	not null
(primary key)	The ID of a SKU.	
`site_id`	VARCHAR(40)	not null
(primary key)	The ID of a site to which the SKU is associated.	

dcs_sku_skuinfo

This table contains information about `skuInfo` items associated with SKU items.

Column	Data Type	Constraint
`sku_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the SKU. References `dcs_sku(sku_id)`.	
`catalog_id`	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	Unique identifier associated with the catalog.	
`sku_info_id`	VARCHAR(40)	NOT NULL
	Unique identifier to associate with this SKU when viewed as part of this catalog.	

dcs_sku_info

This table contains information about the SKUs in catalogs.

Column	Data Type	Constraint
`sku_info_id`	VARCHAR(40)	NOT NULL
(primary key)	Unique identifier associated with the `skuInfo` object.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`item_acl`	LONG VARCHAR	NULL
	The security for the `sku_info`.	

dcs_skuinfo_rplc

This table contains information about the replacement items to associate with a `skuInfo`.

Column	Data Type	Constraint
`sku_info_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcs_sku_info(sku_info_id)`.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	This number is used to order the replacement products.	
`replacement`	VARCHAR(40)	NOT NULL
	Identifier for the replacement item to associate with this `skuInfo`.	

dcs_sub_catalogs

This table contains a list of all sub catalogs (and their sub catalogs).

Column	Data Type	Constraint
`catalog_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog.	
`sub_catalog_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the catalog.	

dcs_user_catalog

This table defines the catalog assigned to a given user's profile. Used by the `catalog` property of the `user` item in the profile repository.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	not null
(primary key)	The ID of the user profile whose catalog is defined in the `user_catalog` column	
`user_catalog`	VARCHAR(40)	null
	The ID of the catalog that is used by the user defined by the `user_id` column	

dcs_manufacturer

The following table contains information related to manufacturers. Each product that appears in the product catalog can be associated with a particular manufacturer. These tables are used for information purposes by the buyer.

Column	Data Type	Constraint
`manufacturer_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of the manufacturer.	
`manufacturer_name`	WVARCHAR(254)	NULL
	A string name that identifies the manufacturer.	
`Description`	WVARCHAR(254)	NULL

Column	Data Type	Constraint
	A short description of this manufacturer.	
long_description	LONG WVARCHAR	NULL
	A long description of this manufacturer.	
Email	VARCHAR(30)	NULL
	An e-mail address for this manufacturer.	

dcs_measurement

The following table contains information related to the measurements of a particular item.

Column	Data Type	Constraint
sku_id	VARCHAR(40)	NOT NULL
(primary key	The unique SKU ID for an item	
unit_of_measure	INT	NULL
	The unit of measurement that is used to quantify this item. Meters, liters etc.	
quantity	DOUBLE PRECISION	NULL
	The quantity of the particular unit of measurement.	

Commerce Users Tables

Oracle ATG Web Commerce uses the following tables to store information about commerce users:

- dps_credit_card (page 579)
- dcs_user (page 580)
- dps_usr_creditcard (page 581)

dps_credit_card

This table contains information that describes a credit card.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the credit card.	

Column	Data Type	Constraint
`credit_card_number`	VARCHAR(40)	NULL
	The credit card number.	
`credit_card_type`	VARCHAR(40)	NULL
	The type of the credit card.	
`expiration_month`	VARCHAR(20)	NULL
	The month the credit card expires.	
`exp_day_of_month`	VARCHAR(20)	NULL
	The day of the month the credit card expires.	
`expiration_year`	VARCHAR(20)	NULL
	The year the credit card expires.	
`billing_addr`	VARCHAR(40)	NULL
	The billing address of the credit card. References `dps_contact_info(id)`	

dcs_user

This table contains information about a user's credit card and price lists.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user. References `dps_user(id)`	
`allow_partial_ship`	TINYINT	NULL CHECK `(allow_partial_ship in (01))`
	Determines whether the user will allow partial shipment of an order.	
`default_creditcard`	VARCHAR(40)	NULL
	The Unique identifier associated with the user's default credit card. References `dps_credit_card(id)`	
`daytime_phone_num`	VARCHAR(20)	NULL
	The user's daytime phone number.	
`express_checkout`	TINYINT	NULL CHECK`(express_checkout in (01))`

Column	Data Type	Constraint
	Determines whether the user has chosen the express checkout option.	
`default_carrier`	VARCHAR(256)	NULL
	The default mail carrier for the order.	
`price_list`	VARCHAR(40)	NULL
	The price list assigned to the user; reference to `dcs_price_list.price_list_id`.	
`sale_price_list`	VARCHAR(40)	NULL
	The sale price list assigned to the user; reference to `dcs_price_list.price_list_id`.	

dps_usr_creditcard

This table models a `java.util.Map` object. It allows a user to store a collection of named credit cards.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user. References `dps_user(id)`	
`tag`	VARCHAR(42)	NOT NULL
(primary key)	The user's chosen name for the credit card.	
`credit_card_id`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the credit card. References `dps_credit_card(id)`	

Claimable Tables

Oracle ATG Web Commerce uses the following table to store claimable information:

- dcspp_claimable (page 582)
- dcspp_giftcert (page 582)
- dcs_storecred_clm (page 583)
- dcspp_coupon (page 584)
- dcspp_coupon_info (page 584)

- dcspp_cp_folder (page 585)

dcspp_claimable

Column	Data Type	Constraint
`claimable_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository id of the claimable repository item.	
`version`	INTEGER	NOT NULL
	Used by the repository to detect "out of date" updates. This information is managed internally by the Oracle ATG Web Commerce repositories.	
`type`	INTEGER	NOT NULL
	Indicates whether a particular instance of a repository item is a coupon or a gift certificate.	
`status`	INTEGER	NULL
	Indicates if a particular item has been claimed or not.	
`expiration_date`	DATE	NULL
	If there is an expiration date, then an item cannot be claimed after this date.	
`last_modified`	DATETIME	NULL
	Indicates when the repository item was last changed.	

dcspp_giftcert

Column	Data Type	Constraint
`giftcertificate_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the gift certificate. References `dcspp_claimable(claimable_id)`	
`amount`	DOUBLE PRECISION	NOT NULL
	The amount of the gift certificate.	
`amount_authorized`	DOUBLE PRECISION	NOT NULL
	The amount of the gift certificate authorized to be used at the time a purchase is made.	

Column	Data Type	Constraint
amount_remaining	DOUBLE PRECISION	NOT NULL
	The amount of the gift certificate remaining after it has been used. Before the gift certificate has been claimed, this amount is equal to the amount property.	
purchaser_id	VARCHAR(40)	NULL
	The profile id of the person who purchased the gift certificate.	
purchase_date	DATE	NULL
	The date on which the gift certificate is purchased	
last_used	DATE	NULL
	The date on which the gift certificate was last used.	

dcs_storecred_clm

This table includes information on store credit.

Column	Data Type	Constraint
store_credit_id	VARCHAR(40)	NOT NULL
(primary key)	References dcspp_claimable(claimable_id)	
amount	DOUBLE PRECISION	NOT NULL
	Contains the original amount of the store credit	
amount_authorized	DOUBLE PRECISION	NOT NULL
	Contains the amount of the store credit authorized.	
amount_remaining	DOUBLE PRECISION	NOT NULL
	The amount of the store credit which has not been consumed yet.	
owner_id	VARCHAR(40)	NULL
	The profile id of the owner of this store credit.	
issue_date	DATETIME	NULL
	The date the store credit was issued.	
expiration_date	DATETIME	NULL

Column	Data Type	Constraint
	The date the store credit expires. Can be NULL.	
`last_used`	DATETIME	NULL
	The date the store credit was last used.	

dcspp_coupon

This table relates coupons to one or more promotions.

Column	Data Type	Constraint
`coupon_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the coupon. References `dcspp_claimable(claimable_id)`	
`promotion_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the promotion.	

dcspp_coupon_info

This table is used for coupon validation.

Column	Data Type	Constraint
`coupon_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the coupon. References `dcspp_claimable(claimable_id)`	
`display_name`	VARCHAR(254)	NULL
	The name a user sees when viewing this coupon.	
`use_promo_site`	INT	NULL
	Boolean indicating whether or not the coupon validation process should check for site or site group associations.	
`parent_folder`	VARCHAR(40)	NULL
	Parent folder for the coupon. References `dcspp_cp_folder(folder_id)`.	
`uses`	INT	NULL

Column	Data Type	Constraint
	Number of times the coupon has been claimed.	

dcspp_coupon_batch

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the coupon batch.	
`code_prefix`	VARCHAR(254)	NOT NULL
`number_of_coupons`	INT	NULL
`seed_value`	LONG	NULL

dcspp_batch_claimable

Column	Data Type	Constraint
`coupon_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the batch claimable.	
`coupon_batch`	VARCHAR(254)	NULL

dcspp_cp_folder

This table holds information about coupon folders.

Column	Data Type	Constraint
`folder_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the folder.	
`name`	VARCHAR(254)	NOT NULL
	The name a user sees when viewing this coupon.	

Column	Data Type	Constraint
parent_folder	VARCHAR(40)	NULL
	Parent folder for the coupon, if one is present. References dcspp_cp_folder(folder_id).	

Shopping Cart Events Table

Oracle ATG Web Commerce uses the following table to store information about shopping cart events:

dcs_cart_event

This table contains information about a shopping cart event, either adding or removing an item.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
	Describes whether this event was an add to the cart, or a remove from the cart.	
timestamp	DATE	NULL
	Date and time the item was added or removed from the cart	
orderid	VARCHAR(40)	NULL
	The order ID associated with the given shopping cart	
itemid	VARCHAR(40)	NULL
	The SKU ID of the item that was added or removed	
quantity	INTEGER	NULL
	The quantity of the item that was added or removed	
amount	NUMBER(19,7)	NULL
	The total price of the item(s) added or removed	
profileid	VARCHAR(40)	NULL
	The profile ID of the user associated with the request when this message is sent in the context of an HTTP request.	
sessionid	VARCHAR(100)	NULL
	The current session ID associated with the request when this message is sent in the context of an HTTP request.	

Column	Data Type	Constraint
parentsessionid	VARCHAR(100)	NULL
	The parent session ID. This ID may be different from the request's current session ID on application servers that use a separate session ID for each Web application.	

Inventory Tables

Oracle ATG Web Commerce uses the following tables to store inventory information:

dcs_inventory

This table stores all the inventory information for each SKU in the product catalog.

Column	Data Type	Constraint
inventory_id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with this inventory.	
version	INTEGER	NOT NULL
	The version of this row. The GSA uses this value.	
creation_date	DATE	NULL
	The date this inventory was created.	
start_date	DATE	NULL
	The date on which this inventory will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
end_date	DATE	NULL
	The last date on which this inventory will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
display_name	VARCHAR(254)	NULL
	The name a user sees when viewing this row.	
description	VARCHAR(254)	NULL
	A text description of the inventory.	
catalog_ref_id	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
	The inventory of this SKU in the product catalog.	
avail_status	INTEGER	NOT NULL
	Indicates whether this inventory is in stock, out of stock, backorderable, preorderable, or if the value should be derived at runtime based on stock_level, backorder_level, and preorder_level.	
availability_date	DATE	NULL
	The date on which the inventory is expected to be available.	
stock_level	INTEGER	NULL
	The number of items in stock.	
backorder_level	INTEGER	NULL
	The number of items that may be backordered.	
preorder_level	INTEGER	NULL
	The number of items that may be preorderable.	
stock_thresh	INTEGER	NULL
	If the stock_level value dips below this value, a ThresholdReached event is sent.	
backorder_thresh	INTEGER	NULL
	If the backorder_level value dips below this value, a ThresholdReached event is sent.	
preorder_thresh	INTEGER	NULL
	If the preorder_level value dips below this value, a ThresholdReached event is sent.	

Order Tables

Oracle ATG Web Commerce uses the following tables to store order information:

- dcspp_order (page 591)
- dcspp_ship_group (page 593)
- dcspp_pay_group (page 594)
- dcspp_item (page 595)
- dcspp_relationship (page 596)

dcspp_order

This table stores information associated with a user' s order such as the submitted date, the state, and the price.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID associated with the order.	
`type`	INTEGER	NOT NULL
	An INTEGER representing the specific type of item descriptor.	
`version`	INTEGER	NOT NULL
	The version of the data within the row.	
`order_class_type`	VARCHAR(40)	NULL
	A string representing the type of order.	
`profile_id`	VARCHAR(40)	NULL
	The ID of the profile to which the order belongs.	
`description`	VARCHAR(64)	NULL
	A text description of the order.	
`state`	VARCHAR(40)	NULL
	The state of the order.	
`state_detail`	VARCHAR(254)	NULL
	Detailed state information about the order.	
`created_by_order`	VARCHAR(40)	NULL
	Contains the ID of the order from which this order was created. Used by the Customer Service Module in situations such as exchanges.	
`origin_of_order`	ENUM(10)	NULL

	Enumerated property that provides information about where the order came from. The default options are:
	`default`
	`scheduledOrder`
	`contactCenter`
	You can extend this to track orders from other sources.	

`creation_date`	DATE	NULL

Column	Data Type	Constraint
	The date and time the order was created.	
`submitted_date`	DATE	NULL
	The date and time the order was submitted for processing.	
`last_modified_date`	DATE	NULL
	The date the order was last modified.	
`completed_date`	DATE	NULL
	The date the order was completed.	
`price_info`	VARCHAR(40)	NULL
	The price data for the order.	
`tax_price_info`	VARCHAR(40)	NULL
	The price data for the tax of the order.	
`explicitly_saved`	BOOLEAN	NULL
	Customers can save orders without checking out to start a fresh order. If this is done, `explicitly_saved` is set to `true`. When the customer starts another session, the current active shopping cart is identified as an incomplete order where `explicitly_saved` is `false`..	
`agent_id`	VARCHAR(40)	NULL
	ID of the agent associated with the order, if any.	
`sales_channel`	NUMBER(10)	NULL
	Identifier of the sales channel through which the order was placed, if any.	
`creation_site_id`	VARCHAR(40)	NULL
	Contains the ID of the site on which this order was created. Used by the multisite feature.	
`site_id`	VARCHAR(40)	NULL
	Contains the ID of the site the user was on for the most recent activity on this order. Used by the multisite feature.	
`gwp`	BOOLEAN	NULL
	Indicates whether the order includes an item added as a result of a gift with purchase promotion.	

dcspp_ship_group

This table stores information associated with a shipping group such as the shipping date, the state, and the shipping cost.

Column	Data Type	Constraint
`shipping_group_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID associated with the shipping group.	
`type`	INTEGER	NOT NULL
	An INTEGER representing the specific type of item descriptor.	
`version`	INTEGER	NOT NULL
	The version of the data within the row.	
`shipgrp_class_type`	VARCHAR(40)	NULL
	A string representing the type of shipping group.	
`shipping_method`	VARCHAR(40)	NULL
	The shipping method, for example Ground, Next Day, etc.	
`description`	VARCHAR(64)	NULL
	A text description of the shipping order.	
`ship_on_date`	DATE	NULL
	The date to ship the items within the shipping group.	
`actual_ship_date`	DATE	NULL
	The actual date that the items within the shipping group were shipped.	
`state`	VARCHAR(40)	NULL
	The state of the shipping group.	
`state_detail`	VARCHAR(254)	NULL
	Detailed state information about the shipping group.	
`submitted_date`	DATE	NULL
	The date that the shipping group was submitted for processing.	
`price_info`	VARCHAR(40)	NULL
	The price data for the shipping group.	

Column	Data Type	Constraint
order_ref	VARCHAR(40)	NULL
	The ID of the order to which this shipping group belongs.	

dcspp_pay_group

This table stores information associated with a payment group such as the amount, the state, and the submitted date.

Column	Data Type	Constraint
payment_group_id	VARCHAR(40)	NOT NULL
(primary key)	The ID associated with the payment group.	
type	INTEGER	NOT NULL
	An INTEGER representing the specific type of item descriptor.	
Version	INTEGER	NOT NULL
	The version of the data within the row.	
paygrp_class_type	VARCHAR(40)	NULL
	A string representing the type of payment group.	
payment_method	VARCHAR(40)	NULL
	The payment method, for example Credit Card, Gift Certificate, etc.	
amount	NUMERIC (19,7)	NULL
	The amount of the order this payment group represents.	
amount_authorized	NUMERIC (19,7)	NULL
	The amount of the order that was authorized.	
amount_debited	NUMERIC (19,7)	NULL
	The amount of the order was been debited.	
amount_credited	NUMERIC (19,7)	NULL
	The amount of the order that was credited.	
currency_code	VARCHAR(10)	NULL
	The currency code for the payment group.	

Column	Data Type	Constraint
state	VARCHAR(40)	NULL
	The state of the payment group.	
state_detail	VARCHAR(254)	NULL
	Detailed state information about the payment group.	
submitted_date	DATE	NULL
	The date and time the payment group was submitted for processing.	
order_ref	VARCHAR(40)	NULL
	The ID of the order to which this payment group belongs.	

dcspp_item

This table stores all the properties of an item in a cart such as the SKU ID, the quantity, and the price.

Column	Data Type	Constraint
commerce_item_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the item.	
type	INTEGER	NOT NULL
	An integer representing the specific type of item descriptor.	
version	INTEGER	NOT NULL
	The version of the data within the row.	
item_class_type	VARCHAR(40)	NULL
	A string representing the type of commerce item.	
catalog_id	VARCHAR(40)	NULL
	The ID of the catalog associated with this commerce item.	
catalog_ref_id	VARCHAR(40)	NULL
	The catalog reference ID of the item that this item represents (the SKU).	
catalog_key	VARCHAR(40)	NULL
	An optional property that can be used to signify from which catalog the item was added. Usually contains a locale.	

Column	Data Type	Constraint
product_id	VARCHAR(40)	NULL
	The ID of the product that this item represents.	
quantity	INTEGER	NULL
	The quantity of the item.	
state	VARCHAR(40)	NULL
	The state of the commerce item.	
state_detail	VARCHAR(254)	NULL
	Detailed state information about the commerce item.	
price_info	VARCHAR(40)	NULL
	The price data for the commerce item.	
order_ref	VARCHAR(40)	NULL
	The ID of the order to which this commerce item belongs.	
site_id	VARCHAR(40)	NULL
	ID of the site from which the item was added. Used by the multisite feature.	

dcspp_relationship

This table stores information associated with the relationship of an order such as the relationship type.

Column	Data Type	Constraint
relationship_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the relationship.	
type	INTEGER	NOT NULL
	An INTEGER representing the specific type of item descriptor.	
version	INTEGER	NOT NULL
	The version of the data within the row.	
rel_class_type	VARCHAR(40)	NULL
	A string representing the type of relationship.	

Column	Data Type	Constraint
relationship_type	VARCHAR(40)	NULL
	The type of the relationship. For example, SHIPPINGQUANTITY.	
order_ref	VARCHAR(40)	NULL
	The ID of the order to which this relationship belongs.	

dcspp_rel_orders

This table stores the individual entries of the `specialInstructions` property of the `atg.commerce.order.Order` object.

Column	Data Type	Constraint
order_id	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_order(order_id)`	
related_orders	VARCHAR(40)	NOT NULL
	The id of another order which was created in reference to this order.	
seq_num	VARCHAR(40)	NOT NULL
(primary key)	The sequence number in the list of related orders.	

dcspp_order_inst

This table stores the individual entries of the `specialInstructions` property of the `atg.commerce.order.Order` object.

Column	Data Type	Constraint
order_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the order to which this instruction belongs. References `dcspp_order(order_id)`	
tag	VARCHAR(42)	NOT NULL
(primary key)	The identifier for the order instruction.	
special_inst	VARCHAR(254)	NULL
	The instruction information.	

dcspp_order_sg

This table stores the list of `ShippingGroup` ids that are contained within an Order.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the order to which the shipping group belongs. References `dcspp_order(order_id)`	
`shipping_groups`	VARCHAR(40)	NOT NULL
	The ID of the shipping group.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	The element number within the list to which the shipping group belongs.	

dcspp_order_pg

This table stores the list of `PaymentGroup` IDs that are contained within an order.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the order to which the payment group belongs. References `dcspp_order(order_id)`	
`payment_groups`	VARCHAR(40)	NOT NULL
	The ID of the payment group.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	The element number within the list to which the payment group belongs.	

dcspp_order_item

This table stores the list of `CommerceItem` IDs that are contained within an order.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	The unique identifier of the order to which the commerce item belongs. References `dcspp_order(order_id)`	
`commerce_items`	VARCHAR(40)	NOT NULL
	The ID of the commerce item.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	The element number within the list to which the commerce item belongs.	

dcspp_order_rel

This table stores the list of `Relationship` IDs that are contained within an order.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the order to which the relationship belongs. References `dcspp_order(order_id)`	
`relationships`	VARCHAR(40)	NOT NULL
	The ID of the relationship.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	The element number within the list to which the relationship belongs.	

dbcpp_sched_order

This table contains information related to scheduled orders.

Column	Data Type	Constraint
`scheduled_order_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of the scheduled order entry.	
`type`	INT	NOT NULL
	The repository type of the item.	
`version`	INT	NOT NULL

Column	Data Type	Constraint
	The repository version number.	
name	WVARCHAR(32)	NULL
	The user defined name of a scheduled order.	
profile_id	VARCHAR(40)	NULL
	The profile ID of the owner of a scheduled order.	
create_date	TIMESTAMP	NULL
	The creation date of the scheduled order.	
start_date	TIMESTAMP	NULL
	The date that the scheduled order is to begin submission.	
end_date	TIMESTAMP	NULL
	The date that the scheduled order is to terminate submission.	
template_order	VARCHAR(32)	NULL
	The order ID of the template order.	
state	INT	NULL
	The state of the scheduled order.	
next_scheduled	TIMESTAMP	NULL
	The time of the next scheduled submission.	
schedule	VARCHAR(255)	NULL
	The schedule of when the order is to be submitted.	
siteId	VARCHAR(40)	NULL
	ID of the site with which this scheduled order is associated.	

dcspp_ship_inst

This table contains information about any special instructions associated with the shipping group.

Column	Data Type	Constraint
shipping_group_id	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	The unique identifier of the shipping group to which this instruction belongs. References `dcspp_ship_group(shipping_group_id)`	
`tag`	VARCHAR(42)	NOT NULL
(primary key)	The identifier for the shipping group instruction.	
`special_inst`	VARCHAR(254)	NULL
	The instruction information.	

dcspp_hrd_ship_grp

This table contains information about the hardgood shipping group such as the tracking number.

Column	Data Type	Constraint
`shipping_group_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the shipping group. References `dcspp_ship_group(shipping_group_id)`	
`tracking_number`	VARCHAR(40)	NULL
	The tracking number for the hardgood shipping group.	

dcspp_ele_ship_grp

This table contains information about an electronic shipping group such as the e-mail address.

Column	Data Type	Constraint
`shipping_group_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the shipping group. References `dcspp_ship_group(shipping_group_id)`	
`email_address`	VARCHAR(40)	NULL
	The e-mail address of the user to which this shipping group belongs.	

dcspp_ship_addr

This table contains information about the shipping address of a shipping group.

Column	Data Type	Constraint
`shipping_group_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the shipping group. References `dcspp_ship_group(shipping_group_id)`	
`prefix`	VARCHAR(40)	NULL
	The prefix associated with the name in the shipping address. (Miss, Mrs., etc.)	
`first_name`	VARCHAR(40)	NULL
	The first name of the user at this shipping address.	
`middle_name`	VARCHAR(40)	NULL
	The middle name of the user at this shipping address.	
`last_name`	VARCHAR(40)	NULL
	The last name of the user at this shipping address.	
`suffix`	VARCHAR(40)	NULL
	The suffix associated with the name in the shipping address. (Jr. Sr., etc.)	
`job_title`	VARCHAR(40)	NULL
	The job title of the person in the shipping address.	
`address_1`	VARCHAR(50)	NULL
	The street and number of this shipping address.	
`address_2`	VARCHAR(50)	NULL
	The street and number of this shipping address.	
`address_3`	VARCHAR(50)	NULL
	The street and number of this shipping address.	
`city`	VARCHAR(40)	NULL
	The city of this shipping address.	
`county`	VARCHAR(50)	NULL
	The county of this shipping address.	
`state`	VARCHAR(40)	NULL
	The state of this shipping address.	

Column	Data Type	Constraint
postal_code	VARCHAR(10)	NULL
	The postal code of this shipping address.	
country	VARCHAR(40)	NULL
	The country of this shipping address.	
phone_number	VARCHAR(40)	NULL
	The phone number of the user at this shipping address.	
fax_number	VARCHAR(40)	NULL
	The fax number of the user at this shipping address.	
email	VARCHAR(40)	NULL
	The e-mail of the user at this shipping address.	

dcspp_hand_inst

This table stores all the properties of the `atg.commerce.order.HandlingInstruction` object such as the item and quantity to which the handling instruction belongs.

Column	Data Type	Constraint
handling_inst_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the handling instruction.	
type	INTEGER	NOT NULL
	An integer representing the specific type of item descriptor.	
version	INTEGER	NOT NULL
	The version of the data within the row.	
hndinst_class_type	VARCHAR(40)	NULL
	A string representing the type of handling instruction.	
handling_method	VARCHAR(40)	NULL
	The handling method.	
shipping_group_id	VARCHAR(40)	NULL
	The ID of the shipping group to which this handling instruction belongs.	

Column	Data Type	Constraint
commerce_item_id	VARCHAR(40)	NULL
	The ID of the commerce item that this handling instruction represents	
quantity	INTEGER	NULL
	The quantity of the commerce item to which this handling instruction applies.	

dcspp_gift_inst

This table stores all the properties of a gift list handling instruction object such as the gift list ID.

Column	Data Type	Constraint
handling_inst_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the gift list handling instruction. References dcspp_hand_inst(handling_inst_id)	
giftlist_id	VARCHAR(40)	NULL
	The ID of the gift list that this handling instruction represents.	
giftlist_item_id	VARCHAR(40)	NULL
	The ID of the item within the gift list that this handling instruction represents.	

dcspp_sg_hand_inst

This table stores the list of HandlingInstruction IDs that are contained within a shipping group.

Column	Data Type	Constraint
shipping_group_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the shipping group to which the handling instruction belongs. References dcspp_ship_group(shipping_group_id)	
handling_instrs	VARCHAR(40)	NOT NULL
	The ID of the handling instruction.	
sequence_num	INTEGER	NOT NULL

Column	Data Type	Constraint
(primary key)	The element number within the list to which the handling instruction belongs.	

dcspp_pay_inst

This table stores the individual entries of the `specialInstructions` property of the `atg.commerce.order.PaymentGroup` object. The table contains special order instructions for each item in a certain payment group.

Column	Data Type	Constraint
`payment_group_id`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the payment group. References `dcspp_pay_group(payment_group_id)`	
`Tag`	VARCHAR(42)	NOT NULL
	The identifier for the payment group instruction.	
`special_inst`	VARCHAR(254)	NULL
	The instruction information.	

dcspp_config_item

This table includes information on configurable commerce items.

Column	Data Type	Constraint
`config_item_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_item(commerce_item_id)`	
`reconfig_data`	VARCHAR(255)	NULL
	Optional reconfiguration data about this configurable commerce item.	
`notes`	VARCHAR(255)	NULL
	Optional notes about this configurable commerce item.	

dcspp_subsku_item

This table includes information on a subSKU.

Column	Data Type	Constraint
subsku_item_id	VARCHAR(40)	NOT NULL
(primary key)	References dcspp_item(commerce_item_id)	
ind_quantity	INTEGER	NULL
	The quantity of this subSKU that is contained within a single configurable commerce item.	

dcspp_item_ci

This table includes information on a configurable commerce item.

Column	Data Type	Constraint
commerce_item_id	VARCHAR(40)	NOT NULL
(primary key)	References dcspp_item(commerce_item_id)	
commerce_items	VARCHAR(40)	NOT NULL
	The ID of the subSKU commerce item within this configurable commerce item.	
sequence_num	INTEGER	NOT NULL
(primary key)	The sequence number in the list of subSKU commerce items.	

dcspp_gift_cert

This table stores all the properties of a gift certificate in an order such as the profile ID and the gift certificate number.

Column	Data Type	Constraint
payment_group_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the payment group. References dcspp_pay_group(payment_group_id)	
profile_id	VARCHAR(40)	NULL
	The profile ID to which this gift certificate payment group belongs.	
gift_cert_number	VARCHAR(50)	NULL

Column	Data Type	Constraint
	The gift certificate number that this gift certificate payment group represents.	

dcspp_store_cred

This table includes information on store credit.

Column	Data Type	Constraint
`payment_group_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_payment_group(payment_group_id)`	
`profile_id`	VARCHAR(40)	NULL
	The profile id of the user associated with the store credit referenced in the `store_cred_number`.	
`store_cred_number`	VARCHAR(50)	NULL
	The store credit number associated with this store credit..	

dcspp_credit_card

This table stores all the properties of a credit card in an order such as the credit card number, the expiration month, the day of the month, and year, and the credit card type.

Column	Data Type	Constraint
`payment_group_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the payment group. References `dcspp_pay_group(payment_group_id)`	
`credit_card_number`	VARCHAR(40)	NULL
	The credit card number.	
`credit_card_type`	VARCHAR(40)	NULL
	The type of the credit card.	
`expiration_month`	VARCHAR(20)	NULL
	The month the credit card expires.	

Column	Data Type	Constraint
exp_day_of_month	VARCHAR(20)	NULL
	The day of the month the credit card expires.	
expiration_year	VARCHAR(20)	NULL
	The year the credit card expires.	

dcspp_bill_addr

This table stores the billing address information for a certain payment group.

Column	Data Type	Constraint
payment_group_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the payment group. References dcspp_pay_group(payment_group_id)	
prefix	VARCHAR(40)	NULL
	The prefix associated with the name in the billing address. (Miss, Mrs., etc.)	
first_name	VARCHAR(40)	NULL
	The first name of the user at this billing address.	
middle_name	VARCHAR(40)	NULL
	The middle name of the user at this shipping address.	
last_name	VARCHAR(40)	NULL
	The last name of the user at this shipping address.	
suffix	VARCHAR(40)	NULL
	The suffix associated with the name in the billing address. (Jr. Sr., etc.)	
job_title	VARCHAR(40)	NULL
	The job title of the person in the billing address.	
address_1	VARCHAR(50)	NULL
	The street and number of this billing address.	
address_2	VARCHAR(50)	NULL
	The street and number of this billing address.	

Column	Data Type	Constraint
address_3	VARCHAR(50)	NULL
	The street and number of this billing address.	
city	VARCHAR(40)	NULL
	The city of this billing address.	
county	VARCHAR(40)	NULL
	The county of this billing address.	
state	VARCHAR(40)	NULL
	The state of this billing address.	
postal_code	VARCHAR(10)	NULL
	The postal code of this billing address.	
country	VARCHAR(40)	NULL
	The country of this billing address.	
phone_number	VARCHAR(40)	NULL
	The phone number of the user at this billing address.	
fax_number	VARCHAR(40)	NULL
	The fax number of the user at this billing address.	
email	VARCHAR(40)	NULL
	The e-mail of the user at this billing address.	

dcspp_pay_status

This table stores all the properties of a payment status in a payment group such as the amount, the transaction success, and the transaction timestamp.

Column	Data Type	Constraint
status_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the payment status.	
type	INTEGER	NOT NULL
	An integer representing the specific type of item descriptor.	

Column	Data Type	Constraint
`version`	INTEGER	NOT NULL
	The version of the data within the row.	
`trans_id`	VARCHAR(50)	NULL
	The transaction ID of the payment status.	
`amount`	NUMERIC (19,7)	NULL
	The amount this payment status represents.	
`trans_success`	NUMERIC (1)	NULL CHECK `(trans_success IN (01))`
	A flag indicating the success of the transaction.	
`error_message`	VARCHAR(254)	NULL
	The error message of the transaction, if any.	
`trans_timestamp`	DATE	NULL
	The timestamp of the transaction.	

dcspp_cc_status

This table stores all the properties of a credit card status in a payment group.

Column	Data Type	Constraint
`status_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the credit card status. References `dcspp_pay_status(status_id)`	
`auth_expiration`	TIMESTAMP	NULL
	The time and date the authorization expires.	
`avs_code`	VARCHAR(40)	NULL
	The address verification code.	
`avs_desc_result`	VARCHAR(254)	NULL
	The address verification descriptive result.	

dcspp_gc_status

This table stores all the properties of a gift certificate status in a payment group.

Column	Data Type	Constraint
`status_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the gift certificate status. References `dcspp_pay_status(status_id)`	
`auth_expiration`	TIMESTAMP	NULL
	The time and date the authorization expires.	

dcspp_sc_status

This table stores all the properties of a store credit status in a payment group.

Column	Data Type	Constraint
`status_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the store credit status. References `dcspp_pay_status(status_id)`	
`auth_expiration`	TIMESTAMP	NULL
	The time and date the authorization expires.	

dcspp_auth_status

This table stores the list of `PaymentStatus` IDs that are contained within a `PaymentGroup` that represent authorizations.

Column	Data Type	Constraint
`payment_group_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the payment group which the authorization status belongs. References `dcspp_pay_group(payment_group_id)`	
`auth_status`	VARCHAR(40)	NOT NULL
	The ID of the authorization status.	

Column	Data Type	Constraint
sequence_num	INTEGER	NOT NULL
(primary key)	The element number within the list to which the authorization status belongs.	

dcspp_debit_status

This table stores the list of `PaymentStatus` IDs that are contained within a `PaymentGroup` that represent debits.

Column	Data Type	Constraint
payment_group_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the payment group which the debit status belongs. References `dcspp_pay_group(payment_group_id)`	
debit_status	VARCHAR(40)	NOT NULL
	The ID of the debit status.	
sequence_num	INTEGER	NOT NULL
(primary key)	The element number within the list to which the debit status belongs.	

dcspp_cred_status

This table stores the list of `PaymentStatus` IDs that are contained within a `PaymentGroup` that represent credits.

Column	Data Type	Constraint
payment_group_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the payment group which the credit status belongs. References `dcspp_pay_group(payment_group_id)`	
credit_status	VARCHAR(40)	NOT NULL
	The ID of the credit status.	
sequence_num	INTEGER	NOT NULL
(primary key)	The element number within the list to which the credit status belongs.	

dcspp_shipitem_rel

This table contains information about relationships between shipping groups and commerce items. This table stores all the properties of a `ShippingGroupCommerceItemRelationship` such as shipping group ID, commerce item ID, and relationship quantity.

Column	Data Type	Constraint
`relationship_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the shipping group commerce item relationship. References `dcspp_relationship(relationship_id)`	
`shipping_group_id`	VARCHAR(40)	NULL
	The ID of the shipping group for this relationship.	
`commerce_item_id`	VARCHAR(40)	NULL
	The ID of the commerce item for this relationship.	
`quantity`	INTEGER	NULL
	The quantity of the commerce item to which this relationship applies.	
`returned_qty`	NUMERIC (19, 7)	NULL
	The returned quantity of the commerce item.	
`amount`	NUMERIC (19,7)	NULL
	The amount of the commerce item to which this relationship applies.	
`state`	VARCHAR(40)	NULL
	The state of the relationship.	
`state_detail`	VARCHAR(254)	NULL
	Detailed state information about the relationship.	

dcspp_rel_range

This table includes information on the ranges associated with a particular relationship. Specifically, `ShippingGroupCommerceItemRelationship` objects refer to some quantity of a `CommerceItem`. The range tells you *which* commerce items are in the given shipping group.

Column	Data Type	Constraint
`relationship_id`	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	The relationship that uses this range.	
`low_bound`	INTEGER	NULL
	The index (inclusive) of the first item in the range.	
`high_bound`	INTEGER	NULL
	The index (inclusive) of the last item in the range.	

dcspp_det_range

This table includes information on the ranges associated with a particular item price detail. Specifically, `DetailedItemPriceInfo` objects refer to some quantity of a `CommerceItem`. The range tells you *which* commerce items are priced in a particular way.

Column	Data Type	Constraint
`relationship_id`	VARCHAR(40)	NOT NULL
(primary key)	The relationship that uses this range.	
`low_bound`	INTEGER	NULL
	The index (inclusive) of the first item in the range.	
`high_bound`	INTEGER	NULL
	The index (inclusive) of the last item in the range.	

dcspp_payitem_rel

This table contains information about relationships between payment groups and commerce items. This table stores all the properties of a `PaymentGroupCommerceItemRelationship` such as payment group ID, commerce item ID, and relationship amount.

Column	Data Type	Constraint
`relationship_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the payment group commerce item relationship. References `dcspp_relationship(relationship_id)`	
`payment_group_id`	VARCHAR(40)	NULL
	The ID of the payment group for this relationship.	

Column	Data Type	Constraint
commerce_item_id	VARCHAR(40)	NULL
	The ID of the commerce item for this relationship.	
quantity	NUMERIC (19,7)	NULL
	The quantity of the commerce item to which this relationship applies.	
amount	NUMERIC (19,7)	NULL
	The amount of the commerce item to which this relationship applies.	

dcspp_payship_rel

This table contains information about relationships between payment and shipping groups. This table stores all the properties of a `PaymentGroupShippingGroupRelationship` such as payment group ID, shipping group ID, and relationship amount.

Column	Data Type	Constraint
relationship_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the payment group shipping group relationship. References `dcspp_relationship(relationship_id)`	
payment_group_id	VARCHAR(40)	NULL
	The ID of the payment group for this relationship.	
shipping_group_id	VARCHAR(40)	NULL
	The ID of the shipping group for this relationship.	
amount	NUMERIC (19,7)	NULL
	The amount of the shipping group to which this relationship applies.	

dcspp_payorder_rel

This table stores all the properties of a `PaymentGroupOrderRelationship` such as payment group ID, order ID, and relationship amount.

Column	Data Type	Constraint
relationship_id	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	The unique identifier associated with the payment group order relationship. References `dcspp_relationship(relationship_id)`	
`payment_group_id`	VARCHAR(40)	NULL
	The ID of the payment group for this relationship.	
`order_id`	VARCHAR(40)	NULL
	The ID of the order for this relationship.	
`amount`	NUMERIC (19,7)	NULL
	The amount of the order to which this relationship applies.	

dcspp_amount_info

This table stores all the common properties of the `priceInfo` object.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the amount info.	
`type`	INTEGER	NOT NULL
	An integer representing the specific type of item descriptor.	
`version`	integer	NOT NULL
	The version of the data within the row.	
`currency_code`	VARCHAR(10)	NULL
	The currency code of the amount info.	
`amount`	NUMERIC (19,7)	NULL
	The amount of the amount info.	
`discounted`	NUMERIC (1)	NULL CHECK(`discounted` IN (01))
	Flag indicating if a discount has been applied.	
`amount_is_final`	TINYINT	NULL CHECK(`amount_is_final` IN (0,1))
	Flag indicating if amount is final amount.	

dcspp_order_price

This table contains information about the price it costs to put together an order. This table stores all the properties of an `OrderPriceInfo` object such as raw subtotal, tax, and shipping.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the order price info. References `dcspp_amount_info(amount_info_id)`	
`raw_subtotal`	NUMERIC (19,7)	NULL
	The raw subtotal of the order.	
`tax`	NUMERIC (19,7)	NULL
	The tax for the order.	
`shipping`	NUMERIC (19,7)	NULL
	The shipping cost for the order.	
`manual_adj_total`	NUMERIC (19,7)	NULL
	Total amount of all manual adjustments associated with the order.	

dcspp_item_price

This table contains information about the price of a specific item. This table stores the properties of an `ItemPriceInfo` object such as list price, raw total price, and sale price.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the item price info. References `dcspp_amount_info(amount_info_id)`	
`list_price`	NUMERIC (19,7)	NULL
	The list price of the item.	
`raw_total_price`	NUMERIC (19,7)	NULL
	The raw total price of the item.	
`sale_price`	NUMERIC (19,7)	NULL
	The sale price of the item.	

Column	Data Type	Constraint
on_sale	NUMERIC (1)	CHECK(on_sale IN (01))
	Flag indicating whether item is on sale.	
qty_discounted	NUMERIC (19,0)	NULL
	The quantity of the items that were discounted.	
qty_as_qualifier	NUMERIC (19,0)	NULL
	The quantity of the items that are being used as qualifiers.	
order_discount	DOUBLE PRECISION	NULL
	The discount associated with the item.	
price_list	VARCHAR(40)	NULL
	The price list associated with the item.	

dcspp_tax_price

This table contains all the tax information associated with an order. This table stores all the properties of a `TaxPriceInfo` object such as city tax, county tax, state tax, and country tax.

Column	Data Type	Constraint
amount_info_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the tax price info. References `dcspp_amount_info(amount_info_id)`	
city_tax	NUMERIC (19,7)	NULL
	The amount of the city tax.	
county_tax	NUMERIC (19,7)	NULL
	The amount of the county tax.	
state_tax	NUMERIC (19,7)	NULL
	The amount of the state tax.	
country_tax	NUMERIC (19,7)	NULL
	The amount of the country tax.	

dcspp_ship_price

This table stores information about the as raw shipping cost of an order

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the shipping price info. References `dcspp_amount_info(amount_info_id)`	
`raw_shipping`	NUMERIC (19,7)	NULL
	The amount of the raw shipping cost.	

dcspp_amtinfo_adj

This table stores the list of `PricingAdjustment` IDs that are contained within a `PriceInfo` object.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the amount info to which the adjustment belongs. References `dcspp_amount_info(amount_info_id)`	
`adjustments`	VARCHAR(40)	NOT NULL
	The unique identifier of the adjustment.	
`sequence_num`	integer	NOT NULL
(primary key)	The element number within the list to which the adjustment belongs.	

dcspp_price_adjust

This table stores all the properties of a `PricingAdjustment` object such as pricing model and adjustment.

Column	Data Type	Constraint
`adjustment_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the adjustment.	
`version`	integer	NOT NULL
	The version of the data within the row.	

Column	Data Type	Constraint
adj_description	VARCHAR(254)	NULL
	A description of the adjustment.	
pricing_model	VARCHAR(40)	NULL
	The unique identifier of the pricing model used for this adjustment.	
Adjustment	DOUBLE PRECISION	NULL
	The amount of the adjustment.	
qty_adjusted	integer	NULL
	The quantity of items this adjustment represents.	
manual_adjustment	VARCHAR(40)	NULL
	The amount of the adjustment.	

dcspp_shipitem_sub

This table contains the shipping subtotal for a shipping group in an order.

Column	Data Type	Constraint
amount_info_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the amount info to which the shipping subtotal info belongs. References dcspp_amount_info(amount_info_id)	
shipping_group_id	VARCHAR(42)	NOT NULL
(primary key)	The ID associated with the shipping group.	
ship_item_subtotal	VARCHAR(40)	NOT NULL
	The subtotal of the shipping charge.	

dcspp_taxshipitem

This table contains the tax information for a shipping group in an order.

Column	Data Type	Constraint
amount_info_id	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	The unique identifier of the amount info to which the shipping tax info belongs. References `dcspp_amount_info(amount_info_id)`	
`shipping_group_id`	VARCHAR(42)	NOT NULL
(primary key)	The ID associated with the shipping group.	
`tax_ship_item_sub`	VARCHAR(40)	NOT NULL
	The amount of the shipping tax.	

dcspp_ntaxshipitem

This table contains information about the non-taxable items in a shipping group in an order.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_amount_info(amount_info_id)`	
`shipping_group_id`	VARCHAR(42)	NOT NULL
(primary key)	The ID associated with the shipping group.	
`non_tax_item_sub`	VARCHAR(40)	NOT NULL
	The amount of the shipping group that is non-taxable..	

dcspp_itmprice_det

This table stores the list of `DetailedItemPriceInfo` IDs contained within a `PriceInfo` object.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the amount info to which the detailed price info belongs. References `dcspp_amount_info(amount_info_id)`	
`cur_price_details`	VARCHAR(40)	NOT NULL
	The detailed price info.	
`sequence_num`	integer	NOT NULL

Column	Data Type	Constraint
(primary key)	The element number within the list to which the detailed price info belongs.	

dcspp_det_price

This table stores detailed item price information such as quantity and quantity as qualifier.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the detailed item price info.	
`quantity`	NUMERIC (19,0)	NULL
	The quantity of items to which this detailed item price info applies.	
`qty_as_qualifier`	NUMERIC (19,0)	NULL
	The quantity of items that acted as a qualifier.	
`order_discount`	DOUBLE PRECISION	NULL
	The amount of the order discount.	
`tax`	DOUBLE PRECISION	NULL
	The amount of the tax on the order.	

dcs_submt_ord_evt

This table contains information about Order Submitted events.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The ID of the event.	
`clocktime`	TIMESTAMP	NULL
	The date and time that the event was sent.	
`orderid`	VARCHAR(40)	NULL
	The ID of the order that was submitted.	
`profileid`	VARCHAR(40)	NULL

Column	Data Type	Constraint
	The profile ID of the user associated with the request when this message is sent in the context of an HTTP request.	
`sessionid`	VARCHAR(100)	NULL
	The current session ID associated with the request when this message is sent in the context of an HTTP request.	
`parentsessionid`	VARCHAR(100)	NULL
	The parent session ID. This ID may be different from the request's current session ID on application servers that use a separate session ID for each Web application.	

dcs_ord_merge_evt

This table contains information about Orders Merged events.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The ID of the event.	
`clocktime`	TIMESTAMP	NULL
	The date and time that the event was sent.	
`sourceorderid`	VARCHAR(40)	NULL
	The ID of the source order that was merged into the destination order.	
`destorderid`	VARCHAR(40)	NULL
	The ID of the destination order into which the source order was merged.	
`sourceremoved`	TINYINT	NULL CHECK `(sourceremoved in (0,1))`
	Determines whether the source order is removed after it is merged with the destination order.	
`profileid`	VARCHAR(40)	NULL
	The profile ID of the user associated with the request when this message is sent in the context of an HTTP request.	
`sessionid`	VARCHAR(100)	NULL

Column	Data Type	Constraint
	The current session ID associated with the request when this message is sent in the context of an HTTP request.	
`parentsessionid`	VARCHAR(100)	NULL
	The parent session ID. This ID may be different from the request's current session ID on application servers that use a separate session ID for each Web application.	

dcspp_order_adj

This table contains information about manual adjustments performed by agents using CSC.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key component)	The ID of the order that received the adjustment; foreign key to the `dcspp_order` table and a component of the primarykey.	
`adjustment_id`	VARCHAR(40)	NOT NULL
	The ID of the adjustment.	
`sequence_num`	INTEGER	NOT NULL
(primary key component)	A sequentially generated number that combines with the `order_id` to create a unique primary key..	

dcspp_manual_adj

This table contains information about manual adjustments performed by agents using CSC.

Column	Data Type	Constraint
`manual_adjust_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of the manual adjustment.	
`type`	INTEGER	NOT NULL
	The repository item type of the manual adjustment. Possible values are: 0 – `OrderFixedAmountAdjustment`	
`adjustment_type`	INTEGER	NOT NULL

Column	Data Type	Constraint
	The type of adjustment made. Possible values are: 0 – `amountOff` (credit) 1 – `amountIncrease` (debit)	
`reason`	INTEGER	NOT NULL
	The reason for applying the adjustment. Possible values are: 0 – `reasonAppeasement` 1 -- `reasonOther`	
`amount`	DOUBLE	NULL
	Amount of the adjustment.	
`notes`	VARCHAR(255)	NULL
	Agent notes related to this adjustment.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	

dcspp_shipitem_tax

That table holds the `shippingItemsTaxPriceInfos` property of the `taxPriceInfo` item descriptor in the order repository.

Column	Data Type	Constraint
`amount_info_id`	VARCHAR(40)	NOT NULL
(primary key component)	The unique identifier associated with the tax PriceInfo.	
`shipping_group_id`	VARCHAR(40)	NOT NULL
(primary key component)	The ID of the shipping group described by the PriceInfo.	
`ship_item_tax`	VARCHAR(40)	NOT NULL
	The ID of a PriceInfo that describes the tax amount for the items in a particular shipping group in the order.	

dbcpp_approverids

This table contains profile ids of users who have approved the order.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_order(order_id)`.	
`approver_ids`	VARCHAR(40)	NOT NULL
	Profile ID for an approver who approved the order.	
`sequence_num`	INT	NOT NULL
(primary key)	The sequence number of the approver ID in a list.	

dbcpp_authapprids

This table contains profile ids of users who have authorization to approver the order.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_order(order_id)`.	
`auth_appr_ids`	VARCHAR(40)	NOT NULL
	Profile ID for a valid approver for an order.	
`sequence_num`	INT	NOT NULL
(primary key)	The sequence number of the approver ID in a list.	

dbcpp_apprsysmsgs

This table contains system-generated messages for order approvals.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_order(order_id)`.	
`appr_sys_msgs`	VARCHAR(254)	NOT NULL
	System generated message for an order requiring approval.	
`sequence_num`	INT	NOT NULL
(primary key)	The sequence number of the system message in a list.	

dbcpp_appr_msgs

This table contains user messages for order approvals.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_order(order_id)`.	
`approver_msgs`	VARCHAR(254)	NOT NULL
	Approver message for an order, which has been approved or rejected.	
`sequence_num`	INT	NOT NULL
(primary key)	The sequence number of the approver message in a list.	

dbcpp_invoice_req

This table contains information related to the invoice request payment method.

Column	Data Type	Constraint
`payment_group_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_pay_group(payment_group_id)`	
`po_number`	VARCHAR(40)	NULL
	The user-supplied purchase order number that was used to create this invoice request.	
`pref_format`	VARCHAR(40)	NULL
	The preferred format for delivery of the invoice created from this invoice request (e.g., text, HTML, XML DTD, etc.). **Note:** Oracle ATG Web Commerce does not currently use this field. The field is provided as a placeholder for applications that may wish to let the user set a preferred delivery format and then try to honor that preference when delivering an invoice.	
`pref_delivery`	VARCHAR(40)	NULL

Column	Data Type	Constraint
	The preferred mode for delivery of the invoice created from this invoice request (e.g., postal mail, e-mail, electronic, etc.). **Note:** Oracle ATG Web Commerce does not currently use this field. The field is provided as a placeholder for applications that may wish to let the user set a preferred delivery channel and then try to honor that preference when delivering an invoice.	
`disc_percent`	NUMERIC(19, 7)	NULL
	The discount percentage offered as part of the payment terms for this invoice, if the invoice is paid within `disc_days`. This is part of the conventional representation of payment terms, which consists of discount percentage, discount days, and net days. For example, payment terms of 2/10/net 30 means that a 2% discount is offered if payment is made within 10 days, but payment in full must be received within 30 days.	
`disc_days`	INT	NULL
	The discount days part of the payment terms – i.e., the number of days within which the invoice must be paid in order to qualify for the specified discount percentage.	
`net_days`	INT	NULL
	The net days part of the payment terms – i.e. the number of within which the invoice must be paid in full.	
`pmt_due_date`	TIMESTAMP	NULL
	The actual date on which payment is due.	

dbcpp_cost_center

This table contains information related to cost centers.

Column	Data Type	Constraint
`cost_center_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of the cost center.	
`type`	INT	NOT NULL
	The repository type of the item.	
`version`	INT	NOT NULL
	The repository version number.	

Column	Data Type	Constraint
costctr_class_type	VARCHAR(40)	NULL
	The mapped name of the class type of the cost center	
identifier	VARCHAR(40) NULL	
	The name of the cost center.	
amount	NUMERIC(19, 7)	NULL
	The amount assigned to the cost center.	
order_ref	VARCHAR(40) NULL	
	The ID of the order associated with this cost center.	

dbcpp_order_cc

This table contains information about which cost centers are contained in which orders.

Column	Data Type	Constraint
order_id	VARCHAR(40)	NOT NULL
(primary key)	References dcspp_order(order_id).	
cost_centers	VARCHAR(40)	NOT NULL
	Cost center ID of the cost center within the order.	
sequence_num	INT	NOT NULL
(primary key)	The sequence number of the cost center in a list.	

dbcpp_sched_clone

This table contains information related to which cloned orders are associated with scheduled orders.

Column	Data Type	Constraint
scheduled_order_id	VARCHAR(40)	NOT NULL
(primary key)	References dbcpp_sched_order(scheduled_order_id).	
cloned_order	VARCHAR(40)	NOT NULL
	The order ID of the cloned order which is derived from a template order.	

Column	Data Type	Constraint
sequence_num	INTEGER	NOT NULL
(primary key)	The sequence number of the cloned order in a list.	

dbcpp_ccitem_rel

This table contains information related to a cost center/commerce item relationship object. It ties commerce items to cost centers.

Column	Data Type	Constraint
relationship_id	VARCHAR(40)	NOT NULL
(primary key)	References dcspp_relationship(relationship_id).	
cost_center_id	VARCHAR(40)	NULL
	The ID of the cost center which the relationship references.	
commerce_item_id	VARCHAR(40)	NULL
	The ID of the commerce item which the relationship references.	
Quantity	NUMERIC(19, 0)	NULL
	The quantity of commerce items which are assigned to the cost center.	
Amount	NUMERIC(19, 7)	NULL
	The amount which is assigned to the cost center.	

dbcpp_ccship_rel

This table contains information related to a cost center/shipping group relationship object. It ties shipping amounts to cost centers.

Column	Data Type	Constraint
relationship_id	VARCHAR(40)	NOT NULL
(primary key)	References dcspp_relationship(relationship_id).	
Cost_center_id	VARCHAR(40)	NULL
	The ID of the cost center which the relationship references.	

Column	Data Type	Constraint
`shipping_group_id`	VARCHAR(40)	NULL
	The ID of the shipping group which the relationship references.	
`Amount`	NUMERIC(19, 7)	NULL
	The shipping amount which is assigned to the cost center.	

dbcpp_ccorder_rel

This table contains information related to a cost center/order relationship object. It ties order amounts to cost centers.

Column	Data Type	Constraint
`relationship_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_relationship(relationship_id)`.	
`Cost_center_id`	VARCHAR(40)	NULL
	The ID of the cost center which the relationship references.	
`order_id`	VARCHAR(40)	NULL
	The ID of the order which the relationship references.	
`amount`	NUMERIC(19, 7)	NULL
	The order amount which is assigned to the cost center.	

dbcpp_pmt_req

This table contains requisition information related to a payment group.

Column	Data Type	Constraint
`payment_group_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcspp_pay_group(payment_group_id)`.	
`req_number`	VARCHAR(40)	NULL
	A requisition number for a payment group.	

Promotion Tables

Oracle ATG Web Commerce uses the following tables to store information about promotions:

dcs_promotion

This table contains information about promotions.

Column	Data Type	Constraint
`promotion_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the promotion.	
`version`	INT	NOT NULL
	The version of the promotion. Oracle ATG Web Commerce uses this value to allow several people to edit the same promotion at the same time.	
`creation_date`	DATE	NULL
	The date the promotion was created.	
`start_date`	DATE	NULL
	The date on which the promotion will become available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	
`end_date`	DATE	NULL
	The last date on which the promotion will be available. This is an optional field that can be used by the SQL Repository as part of an RQL filter to prevent items from being loaded from the database.	

Column	Data Type	Constraint
`display_name`	VARCHAR(254)	NULL
	The name of the promotion that is displayed in the ACC.	
`description`	VARCHAR(254)	NULL
	A text description of the promotion.	
`promotion_type`	INT	NULL
	Indicates the type of promotion.	
`enabled`	NUMERIC(1)	NULL CHECK(`enabled in (01)`)
	Determines whether or not the promotion is valid.	
`begin_usable`	DATE	NULL
	The date Oracle ATG Web Commerce will begin using this promotion.	
`end_usable`	DATE	NULL
	The date ATG Commerce will stop using this promotion.	
`priority`	INT	NULL
	The order in which the promotion should be applied. Low priority takes precedence.	
`global`	NUMERIC(1)	NULL CHECK(`global in (01)`)
	Determines whether this promotion is global.	
`anon_profile`	NUMERIC(1)	NULL CHECK(`anon_profile in (01)`)
	Determines whether the promotion should be given to users with anonymous profiles.	
`allow_multiple`	NUMERIC(1)	NULL CHECK(`allow_multiple in (01)`)
	Determines whether a user can receive more than one copy of the promotion.	
`uses`	INT	NULL
	Determines how many times the promotion can be used by a single customer.	
`rel_expiration`	NUMERIC(1)	NULL CHECK(`rel_expiration in (0,1)`)
`time_until_expire`	integer	NULL

Column	Data Type	Constraint
	The time left until the promotion expires.	

dcs_promo_media

This table contains information about media used in promotions.

Column	Data Type	Constraint
`promotion_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the promotion to which this belongs. References `dcs_promotion(promotion_id)`	
`tag`	VARCHAR(42)	NOT NULL
(primary key)	The identifier for the promotion instruction.	
`media_id`	VARCHAR(40)	NOT NULL
	The unique identifier associated with this media item. References `dcs_media(media_id)`	

dcs_discount_promo

This table contains information about promotion discounts. Oracle ATG Web Commerce uses this information to deduct the correct amount from a user's order, based on the rules of the promotion.

Column	Data Type	Constraint
`promotion_id`	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The unique identifier associated with the promotion. References `dcs_promotion(promotion_id)`	
`calculator`	VARCHAR(254)	NOT NULL
	The Oracle ATG Web Commerce calculator that interprets the promotion and applies the discount.	
`adjuster`	DOUBLE PRECISION	NOT NULL
	The number by which the promotion discounts. For example, the percent off the purchase.	
`pmdl_rule`	LONG VARCHAR	NOT NULL

Column	Data Type	Constraint
	The promotion rule that specifies under what conditions the promotion applies.	
one_use	NUMERIC (1, 0)	DEFAULT NULL
	Determines whether a promotion can be used more than once for a given user.	

dcs_promo_upsell

This table contains information about upsell promotions. Oracle ATG Web Commerce uses this information to determine if the upsell feature is enabled in a promotion.

Column	Data Type	Constraint
promotion_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the promotion. References dcs_promotion(promotion_id)	
upsell	NUMERIC (1,0)	DEFAULT NULL
	Determines whether upselling is enabled for a promotion.	

dcs_upsell_action

This table contains information about dynamic products used in Upsell Actions.

Column	Data Type	Constraint
action_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the Upsell Action.	
version	integer	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
name	VARCHAR(40)	NOT NULL
	The name of the Upsell Action.	
upsell_prd_grp	LONG VARCHAR	DEFAULT NULL

Column	Data Type	Constraint
	The content group associated with an Upsell Action.	

dcs_close_qualif

This table contains information about Closeness Qualifiers.

Column	Data Type	Constraint
`close_qualif_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of the Closeness Qualifier.	
`version`	integer	NOT NULL
	Manages the optimistic locking feature of the SQL Repository. This value is automatically incremented by the SQL Repository when any value of the item is modified. If you change rows directly outside of the Oracle ATG Web Commerce framework, you should also increment the version number.	
`name`	VARCHAR(40)	NOT NULL
	The name of the Closeness Qualifier.	
`priority`	integer	DEFAULT NULL
	The priority given to a Closeness Qualifier in the context of a promotion. Closeness Qualifiers are evaluated in the order specified by their priority.	
`qualifier`	LONG VARCHAR	DEFAULT NULL
	The PMDL rule that describes under which circumstances the Closeness Qualifier applies.	
`upsell_media`	VARCHAR(40)	DEFAULT NULL
	The media item associated with the Closeness Qualifier.	
`upsell_action`	VARCHAR(40)	DEFAULT NULL
	The Upsell Action assigned to the Closeness Qualifier.	

dcs_prm_cls_qlf

This table associates Closeness Qualifiers with promotions.

Column	Data Type	Constraint
promotion_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the promotion. References dcs_promotion(promotion_id)	
closeness_qualif	NUMERIC (1,0)	NOT NULL
	The unique identifier associated with the Closeness Qualifier. References dcs_close_qualif(close_qualif_id)	

dcs_upsell_prods

This table associates fixed products with Upsell Actions.

Column	Data Type	Constraint
action_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the Upsell Action. References dcs_upsell_action(action_id)	
product_id	VARCHAR(40)	NOT NULL
	The ID of a product associated with the Upsell Action.	
sequence_num	integer	NOT NULL
	Used to order rows in this table.	

dcs_prom_used_evt

This table contains information about Uses Promotion events.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The ID of the event.	
clocktime	TIMESTAMP	NULL
	The date and time that the event was sent.	
orderid	VARCHAR(40)	NULL
	The ID of the order for which the promotion was used.	

Column	Data Type	Constraint
promotionid	VARCHAR(40)	NULL
	The ID of the promotion that was used.	
order_amount	NUMERIC(26,7)	NULL
	The amount of the order for which the promotion was used.	
discount	NUMERIC(26,7)	NULL
	The amount discounted as a result of the promotion that was used.	
profileid	VARCHAR(40)	NULL
	The profile ID of the user associated with the request when this message is sent in the context of an HTTP request.	
sessionid	VARCHAR(100)	NULL
	The current session ID associated with the request when this message is sent in the context of an HTTP request.	
parentsessionid	VARCHAR(100)	NULL
	The parent session ID. This ID may be different from the request's current session ID on application servers that use a separate session ID for each Web application.	

dcs_promo_rvkd

This table contains information about Promotion Revoked events.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The ID of the event.	
time_stamp	TIMESTAMP	NULL
	The date and time that the event was sent.	
promotionid	VARCHAR(254)	NOT NULL
	The ID of the promotion that was revoked.	
profileid	VARCHAR(254)	NOT NULL
	The profile ID of the user associated with the request when this message is sent in the context of an HTTP request.	

Column	Data Type	Constraint
sessionid	VARCHAR(100)	NULL
	The current session ID associated with the request when this message is sent in the context of an HTTP request.	
parentsessionid	VARCHAR(100)	NULL
	The parent session ID. This ID may be different from the request's current session ID on application servers that use a separate session ID for each Web application.	

dcs_promo_grntd

This table contains information about Promotion Offered events.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL UNIQUE
(primary key)	The ID of the event.	
time_stamp	TIMESTAMP	NULL
	The date and time that the event was sent.	
promotionid	VARCHAR(254)	NOT NULL
	The ID of the promotion that was offered.	
profileid	VARCHAR(254)	NOT NULL
	The profile ID of the user associated with the request when this message is sent in the context of an HTTP request.	
sessionid	VARCHAR(100)	NULL
	The current session ID associated with the request when this message is sent in the context of an HTTP request.	
parentsessionid	VARCHAR(100)	NULL
	The parent session ID. This ID may be different from the request's current session ID on application servers that use a separate session ID for each Web application.	

User Promotion Tables

Oracle ATG Web Commerce uses the following tables to store information about user promotions:

dcs_usr_promostat

This table contains information about the status of promotions owned by specific users and the number of remaining uses of those promotions.

Column	Data Type	Constraint
`status_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the promotion status.	
`profile_id`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the user who owns this promotion status.	
`promotion`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the promotion. References `dcs_promotion(promotion_id)`	
`num_uses`	INT	none
	The number of uses the promotion has remaining.	
`expirationDate`	DATE	none
	The date the promotion expires.	
`grantedDate`	DATE	none
	The date the promotion was granted.	

dcs_usr_actvpromo

This table contains information about a user's active promotions.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user. References `dps_user(id)`	
`sequence_num`	INT	NOT NULL
(primary key)	Used to order rows in the table.	

Column	Data Type	Constraint
promo_status_id	VARCHAR(40)	NOT NULL
	The ID of the promo status Object (a promotion and its number of uses) associated with the active promotion.	

dcs_usr_usedpromo

This table contains information about promotions that have been used.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user.	
promotion_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the promotion.	

Gift List Tables

Oracle ATG Web Commerce uses the following tables to store information about gift lists and wish lists:

- dcs_giftlist (page 641)
- dcs_giftinst (page 643)
- dcs_giftitem (page 643)
- dcs_giftlist_item (page 644)
- dcs_user_wishlist (page 644)
- dcs_user_giftlist (page 645)
- dcs_user_otherlist (page 645)

dcs_giftlist

This table contains information about gift lists.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the gift list.	
owner_id	VARCHAR(40)	NULL

Column	Data Type	Constraint
	The ID of the user who owns the gift list. References `dps_user(id)`	
`is_public`	NUMERIC(1)	NOT NULL CHECK `(is_public in (0,1))`
	Specifies whether or not this gift list is public or private.	
`is_published`	NUMERIC(1)	NOT NULL CHECK `(is_published in (0,1))`
	Specifies whether or not the gift list is published.	
`event_name`	VARCHAR(64)	NULL
	The name of an event.	
`event_type`	INT	NULL
	Enumerated type of events, for example, a birthday.	
`event_date`	DATE	NULL
	The date on which the event took place.	
`comments`	VARCHAR(254)	NULL
	Comments associated with the gift list.	
`descriptions`	VARCHAR(254)	NULL
	A text description of the gift list.	
`instructions`	VARCHAR(254)	NULL
	Any special instructions associated with the gift list.	
`creation_date`	DATE	NULL
	The date the gift list was created.	
`last_modified`	DATE	NULL
	The date the gift list was last modified.	
`shipping_addr_id`	VARCHAR(40)	NULL
	The ID of the shipping address of the user who owns the gift list. References `dps_contact_info(id)`	
`site_id`	VARCHAR(40)	NULL
	ID of the site on which the gift list was created. Used by the multisite feature.	

dcs_giftinst

This table contains information about instructions associated with gifts.

Column	Data Type	Constraint
giftlist_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the gift list. References dcs_giftlist(id)	
tag	VARCHAR(42)	NOT NULL
(primary key)	The identifier for the gift list instruction.	
special_inst	VARCHAR(254)	NULL
	Any special instructions associated with the gift.	

dcs_giftitem

This table contains information about an item in a gift list.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of an item on the gift list.	
catalog_ref_id	VARCHAR(40)	NULL
	The SKU ID.	
product_id	VARCHAR(40)	NULL
	The product ID.	
display_name	VARCHAR(254)	NULL
	The visible name of the item on the gift list.	
description	VARCHAR(254)	NULL
	A short text description of the item on the gift list.	
quantity_desired	INT	NULL
	The number of items that the person creating the list would like to receive..	
quantity_purchased	INT	NUL

Column	Data Type	Constraint
	The number of items that have already been purchased from the list.	
site_id	VARCHAR(40)	NUL
	ID of the site with which the item is associated, if any.	

dcs_giftlist_item

This table stores a map of items for a given gift list.

Column	Data Type	Constraint
giftlist_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the gift list. References dcs_giftlist (id)	
sequence_num	INT	NOT NULL
(primary key)	Used to order rows in this table.	
giftitem_id	VARCHAR(40)	NOT NULL
	The ID associated with an item on the gift list. References dcs_giftitem(id)	
site_id	VARCHAR(40)	NULL
	ID of the site on which the gift list was created. Used by the multisite feature.	

dcs_user_wishlist

This table stores a map of gift lists created by a given user.

Column	Data Type	Constraint
user_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user who owns the wish list. References dps_user(id)	
giftlist_id	VARCHAR(40)	NULL
	The ID of the user's gift list. References dcs_giftlist(id)	

dcs_user_giftlist

This table stores the map of other gift lists for which a user is currently shopping.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user who owns the gift list. References `dps_user(id)`	
`sequence_num`	INT	NOT NULL
(primary key)	Used to order rows in this table.	
`giftlist_id`	VARCHAR(40)	NULL
	The ID associated with the user's gift list.	

dcs_user_otherlist

This table contains information about a user's other list of items.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the user. References `dps_user(id)`	
`sequence_num`	INT	NOT NULL
(primary key)	Used to order rows in this table.	
`giftlist_id`	VARCHAR(40)	NULL
	The ID associated with the user's gift list. References `dcs_giftlist(id)`	

Price List Tables

The following database tables store information related to price lists.

- dcs_price_list (page 646)
- dcs_complex_price (page 647)
- dcs_price (page 647)
- dcs_price_levels (page 648)

- dcs_price_level (page 648)
- dcs_gen_fol_pl (page 649)
- dcs_child_fol_pl (page 650)
- dcs_plfol_chld (page 650)

dcs_price_list

This table contains information related to basic price list functionality.

Column	Data Type	Constraint
price_list_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the price list	
version	integer	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
display_name	VARCHAR(254)	NULL
	The name of the price list displayed in the ACC.	
description	VARCHAR(254)	NULL
	The description of the price list displayed in the ACC.	
creation_date	TIMESTAMP	NULL
	The date the price list was created.	
last_mod_date	TIMESTAMP	NULL
	The last date the price list was modified.	
start_date	TIMESTAMP	NULL
	The date that this price list should become active.	
end_date	TIMESTAMP	NULL
	The date that this price list should become inactive.	
locale	INTEGER	NULL
	The locale for the price list. For example, en_US.	
base_price_list	VARCHAR(40)	NULL
	The ID of the base price list. (If a price is not found in the current price list, look in the base price list.)	
item_acl	LONG VARCHAR	NULL

Column	Data Type	Constraint
	The security for the price list.	

dcs_complex_price

This table contains information related to complex price list functionality.

Column	Data Type	Constraint
`complex_price_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the complex price.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`item_acl`	LONG VARCHAR	NULL
	The security for the complex price list.	

dcs_price

This table contains information related to the price of a product, a SKU, or a product/SKU pair. The price can be a list price, a bulk price, or a tiered price.

Column	Data Type	Constraint
`price_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the price.	
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`price_list`	VARCHAR(40)	NOT NULL
	The unique identifier associated with the price list. References `dcs_price_list(price_list_id)`	
`product_id`	VARCHAR(40)	NULL
	The ID of the product that this price refers to. (optional)	
`sku_id`	VARCHAR(40)	NULL

Column	Data Type	Constraint
	The ID of the SKU that this price refers to. (optional)	
parent_sku_id	VARCHAR(40)	NULL
	The ID of the configurable SKU that is a parent of this sku_id. (optional)	
pricing_scheme	INTEGER	NOT NULL
	The type of price. (LIST_PRICE, BULK_PRICE, TIERED_PRICE).	
list_price	DOUBLE PRECISION	NULL
	If pricing_scheme is LIST_PRICE, this is the price.	
complex_price	VARCHAR(40)	NULL
	If pricing_scheme is not LIST_PRICE, this is the ID of the complex price to use. References dcs_complex_price(complex_price_id).	
item_acl	LONG VARCHAR NULL	
	The security for the complex price list.	

dcs_price_levels

This table contains information related to which price levels are in each complex price.

Column	Data Type	Constraint
complex_price_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the complex price. References dcs_complex_price(complex_price_id).	
price_levels	VARCHAR(40)	NOT NULL
	The unique identifier associated with the price level.	
sequence_num	INTEGER	NOT NULL
(primary key)	Use to order the price levels.	

dcs_price_level

This table contains information related to the price level that is used to price a specific level or tier when using bulk pricing or tiered pricing.

Column	Data Type	Constraint
`version`	INTEGER	NOT NULL
	The integer incremented with each revision to prevent version conflict.	
`price_level_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the price.	
`quantity`	INTEGER	NOT NULL
	The quantity that must be purchased for this price level to take effect.	
`price`	DOUBLE PRECISION	NOT NULL
	The unit price of each quantity that is priced with this price level.	
`item_acl`	LONG VARCHAR	NULL
	The security for the price level.	

dcs_gen_fol_pl

This table contains information related to the folder structure used by the ACC.

Column	Data Type	Constraint
`folder_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier of the folder item.	
`type`	INTEGER	NOT NULL
	The type of folder.	
`name`	VARCHAR(40)	NOT NULL
	The name of the folder that gets display in the ACC.	
`parent`	VARCHAR(40)	NULL
	The parent folder of this folder.	
`description`	VARCHAR(254)	NULL
	A description of this folder.	
`item_acl`	LONG VARCHAR	NULL
	Security information for this folder.	

dcs_child_fol_pl

This table contains information related to the child folders of each folder.

Column	Data Type	Constraint
`folder_id`	VARCHAR(40)	NOT NULL
(primary key)	Unique identifier associated with the folder. References `dcs_gen_fol_pl(folder_id)`.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	This number is used to order the child folders.	
`child_folder_id`	VARCHAR(40)	NOT NULL
	The ID of the child folder.	

dcs_plfol_chld

This table contains information related to the price lists that are contained in each folder.

Column	Data Type	Constraint
`plfol_id`	VARCHAR(40)	NOT NULL
(primary key)	References `dcs_gen_fol_pl(folder_id)`.	
`sequence_num`	INTEGER	NOT NULL
(primary key)	Use to order the price lists.	
`price_list_id`	VARCHAR(40)	NOT NULL
	The ID of the price list contained in the folder.	

Abandoned Order Services Tables

Oracle ATG Web Commerce uses the following tables to store information about users' abandoned orders:

- dcspp_ord_abandon (page 651)
- dcs_user_abandoned (page 652)
- drpt_conv_order (page 652)
- drpt_session_ord (page 653)

See the *Using Abandoned Order Services* (page 453) chapter for information on the Abandoned Order Services module.

dcspp_ord_abandon

This table contains information about abandoned orders.

Column	Data Type	Constraint
`abandonment_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the `abandonmentInfo` item.	
`version`	INTEGER	NOT NULL
	The `abandonmentInfo` item's repository version number. This information is managed and used internally by the repository.	
`order_id`	VARCHAR(40)	NOT NULL
	The ID of the order for which the `abandonmentInfo` item holds abandonment information.	
`ord_last_updated`	DATE	NULL
	The date and time that the order was most recently modified. This property is used to detect activity on abandoned orders.	
`abandon_state`	VARCHAR(40)	NULL
	The abandonment state of the associated order.	
`abandonment_count`	INTEGER	NULL
	The number of times the associated order has been identified as ABANDONED.	
`abandonment_date`	DATE	NULL
	The date and time the associated order was most recently identified as ABANDONED.	
`reanimation_date`	DATE	NULL
	The date and time the associated order was most recently identified as REANIMATED.	
`convert_time`	DATE	NULL
	The date and time the associated order was identified as CONVERTED.	
`lost_date`	DATE	NULL

Column	Data Type	Constraint
	The date and time the associated order was most recently identified as LOST.	

dcs_user_abandoned

This table contains information about users' abandoned orders.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of this `abandoned-order` item.	
`order_id`	VARCHAR(40)	NOT NULL
	The ID of the order.	
`profile_id`	VARCHAR(40)	NOT NULL
	The profile ID of the user associated with the abandoned order.	

drpt_conv_order

This table contains information about users' converted orders, that is, previously abandoned, reanimated, or lost orders that subsequently have been checked out.

Column	Data Type	Constraint
`order_id`	VARCHAR(40)	NOT NULL
(primary key)	The ID of the converted order.	
`converted_date`	DATE	NOT NULL
	The date and time that the order was converted.	
`amount`	DOUBLE PRECISION	NOT NULL
	The total price of the converted order.	
`promo_count`	INTEGER	NULL
	The number of promotions that were applied to the converted order.	

Column	Data Type	Constraint
promo_value	DOUBLE PRECISION	NULL
	The total value of the promotions that were applied to the converted order.	

drpt_session_ord

This table tracks information about orders that have not been checked out at the end of a session.

Column	Data Type	Constraint
order_id	VARCHAR(40)	NOT NULL
(primary key)	The ID of the submitted order.	
dataset_id	VARCHAR(40)	NOT NULL
	The ID of the dataset.	
date_time	DATE timestamp	NOT NULL
	The date and time that the order was converted.	
amount	DOUBLE PRECISION numeric(19,7)	NOT NULL
	The total price of the submitted order.	
submitted	INTEGER	NULL
	The number of promotions that were applied to the converted order.	
session_id	VARCHAR(40)	NULL
	The total value of the promotions that were applied to the converted order.	
parent_session_id	VARCHAR(40)	NULL
	The total value of the promotions that were applied to the converted order.	
order_persistent	NUMERIC	NULL
	A value of 1 indicates that the order is persistent.	

Order Markers Tables

Oracle ATG Web Commerce uses the following tables to store information about order markers:

- dcs_order_marker (page 654)

- dcs_gwp_order_markers (page 654)
- dcspp_gwp_itemmarkers (page 656)

dcs_order_marker

This table holds information about markers assigned to orders when an order reaches a business process stage.

Column	Data Type	Constraint
`marker_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the marker.	
`order_id`	VARCHAR(40)	NOT NULL
	The ID of the order for that has an order marker.	
`marker_key`	VARCHAR(100)	NOT NULL
	The name of the business process associated with the marker.	
`marker_value`	VARCHAR(100)	NULL
	The name of the business process stage associated with the marker.	
`marker_data`	VARCHAR(100)	NULL
	This column is not currently in use.	
`creation_date`	timestamp	NULL
	The date the business process stage is reached and the order marker is assigned to the order.	
`version`	INTEGER	NOT NULL
	The order marker repository version number. This information is managed and used internally by the repository.	
`marker_type`	INTEGER	NULL
	This column is not currently in use.	

dcs_gwp_order_markers

This table holds information about markers assigned to gift with purchase promotion orders.

Column	Data Type	Constraint
`marker_id`	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	The unique identifier associated with the marker.	
gift_type	VARCHAR(100)	NOT NULL
	The type of gift, which can be sku, product, category, or contentGroup.	
gift_detail	VARCHAR(1024)	NOT NULL
	String identifying the gift, usually the repository id of the gift type.	
auto_remove	NUM	NOT NULL
	Flag to indicate whether free gifts should be auto removed if the promotion no longer qualifies. This is configured by the merchandiser when the promotion is created.	
quantity	INTEGER	NOT NULL
	The total quantity of sku for this gift selection. It is the quantity from the PMDL multiplied by the number of times the offer applied (if it applied more than once due to a 'for next' or multiple grants).	
targeted_quantity	INTEGER	NOT NULL
	The quantity of free sku for this gift selection that has already been targeted and made free by the calculator in the order.	
automatic_quantity	INTEGER	NOT NULL
	The quantity of free sku for this gift selection that has already been auto added to the order.	
selected_quantity	INTEGER	NULL
	The quantity of free sku for this gift selection that has already been selected by the Shopper and added to the order.	
removed_quantity	INTEGER	NULL
	The amount of free quantity that has since been manually removed by a shopper. Keeping track of removed quantities prevents them from being automatically re-added in future pricing operations. The assumption is that the customer does not want the free item.	
order_id	VARCHAR(40)	NOT NULL
	Identifier of the order that contains the gift item.	
marker_key	VARCHAR(100)	NOT NULL
	Key identifying the type of the marker.	
marker_value	VARC HAR(100)	NULL

Column	Data Type	Constraint
	Value of the marker.	
marker_data	VARCHAR(100)	NOT NULL
	Data for the marker.	
marker_type	INT	NULL
	The marker type.	
failed_quantity	INT	NOT NULL
	Quantity of gift with purchase items where an attempt was made to automatically add the item, but the attempt failed.	

dcspp_gwp_itemmarkers

This table holds information about markers assigned to gift with purchase promotion commerce items.

Column	Data Type	Constraint
marker_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the marker.	
targeted_quantity	INTEGER	NOT NULL
	The quantity of the free gift that this commerce item is making free and was targeted by pricing.	
automatic_quantity	INTEGER	NOT NULL
	The quantity of the free gift that this commerce item is making free and was auto added by pricing.	
selected_quantity	INTEGER	NULL
	The quantity of the free gift that this commerce item is making free and was selected by the shopper.	
commerce_item_id	VARCHAR(40)	NOT NULL
marker_key	VARCHAR(100)	NOT NULL
	Key identifying the type of the marker.	
marker_value	VARC HAR(100)	VARC HAR(100)
	Value of the marker.	

Column	Data Type	Constraint
marker_type	VARCHAR(100)	VARCHAR(100)
	Data for the marker.	
remaining_quantity	INT	NOT NULL

Organizational Tables

The following database tables contain information related to Oracle ATG Web Commerce organizational roles:

dbc_organization

The `dbc_organization` table contains information related to the basic model for organizational hierarchies. It layers properties onto the organization model that describe a business organization.

Column	Data Type	Constraint
id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the organization. References `dps_organization(org_id)`.	
type	INT	NULL
	An enumerated property of the type of organization this organization represents. Department, company, division etc.	
cust_type	INT	NULL

Column	Data Type	Constraint
	An enumerated property that indicates the status level of this company. Preferred, Enterprise or Standard.	
duns_number	VARCHAR(20)	NULL
	The company's Dun and Bradstreet number.	
dflt_shipping_addr	VARCHAR(40)	NULL
	A default shipping address for this organization. References dps_contact_info(id).	
dflt_billing_addr	VARCHAR(40)	NULL
	A default billing address for this organization. References dps_contact_info(id).	
dflt_payment_type	VARCHAR(40)	NULL
	A default credit card for this organization. References dps_credit_card(id).	
dflt_cost_center	VARCHAR(40)	NULL
	A default cost center for this organization.	
order_price_limit	NUMERIC(19, 7)	NULL
	The order limit for this organization. An order limit is used to trigger an approval condition.	
contract_id	VARCHAR(40)	NULL
	A contract that this organization has negotiated with the site owning organization. References dbc_contract(contract_id).	
approval_required	TINYINT	NULL
	Indicates whether approval is required for members of this organization.	

dbc_org_contact

The dbc_org_contact table contains information that associates an Organization with one or more contacts.

Column	Data Type	Constraint
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the organization. References dbc_organization(id).	

Column	Data Type	Constraint
contact_id	VARCHAR(40)	NOT NULL
	An individual that acts as the point contact for this organization. References dps_contact_info(id).	
Seq	INT	NOT NULL
(primary key)	Indicates the contact's sequence in the list.	

dbc_org_admin

The dbc_org_admin table contains information that associates an Organization with one or more administrators. Unlike contacts, administrators are required to be registered users of the site, since they can create, modify, and delete buyer accounts, organizational profile elements, etc.

Column	Data Type	Constraint
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the organization. References dbc_organization(id).	
User_id	VARCHAR(40)	NOT NULL
	The repository ID of the person is an administrator for the associated Organization. References dps_user(id).	
Seq	INT	NOT NULL
(primary key)	The sequence in the list where the admin is located.	

dbc_org_approver

The dbc_org_approver table contains information that associates an Organization with one or more order approvers. Similar to administrators, approvers are required to be registered users of the site so they can perform online approvals.

Column	Data Type	Constraint
Column	**Data Type**	**Constraint**
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the organization. References dbc_organization(id).	

Column	Data Type	Constraint
approver_id	VARCHAR(40)	NOT NULL
	The repository ID of an individual who acts as an approver for the associated organization. References dps_user(id).	
Seq	INT	NOT NULL
(primary key)	Indicates the approver's sequence in the list.	

dbc_org_costctr

The dbc_org_costctr table contains information that associates an Organization with one or more cost centers that are pre-approved for use by members of the organization.

Column	Data Type	Constraint
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the organization. References dbc_organization(id).	
Cost_center	VARCHAR(40)	NOT NULL
	The repository ID of a cost center that is associated with the organization.	
Seq	VARCHAR(40)	NOT NULL
(primary key)	Indicates the cost center's sequence in the list.	

dbc_org_payment

The dbc_org_payment table contains information that associates an Organization with one or more payment types that are pre-approved for use by members of the organization. For now, a payment type is always a credit card.

Column	Data Type	Constraint
Column	**Data Type**	**Constraint**
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of the organization. References dbc_organization(id).	
Tag	VARCHAR(42)	NOT NULL,

Column	Data Type	Constraint
(primary key)	A key by which the associated credit card will be known as. This value acts as a key into a map.	
`payment_id`	VARCHAR(40)	NOT NULL
	The repository ID of a credit card. References `dps_credit_card(id)`.	

dbc_org_shipping

The `dbc_org_shipping` table contains information that associates an Organization with one or more pre-approved shipping addresses

Column	Data Type	Constraint
`org_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of an organization. References `dbc_organization(id)`.	
`Tag`	VARCHAR(42)	NOT NULL
(primary key)	A key by which the associated shipping address will be known as. This value acts as a key into a map.	
`addr_id`	VARCHAR(40)	NOT NULL
	The repository ID of a contact info. This becomes a shipping address. References `dps_contact_info(id)`.	

dbc_org_billing

The `dbc_org_billing` table contains information that associates an Organization with one or more pre-approved billing addresses.

Column	Data Type	Constraint
`org_id`	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of an organization. References `dbc_organization(id)`.	
`Tag`	VARCHAR(42)	NOT NULL,
(primary key)	A key by which the associated billing address will be known as. This value acts as a key into a map.	

Column	Data Type	Constraint
addr_id	VARCHAR(40)	NOT NULL
	The repository ID of a contact info. This becomes a billing address. References dps_contact_info(id).	

dbc_org_prefvndr

The dbc_org_prefvndr table contains information that associates an Organization with one or more preferred vendors.

Column	Data Type	Constraint
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of an organization. References dbc_organization(id).	
Vendor	WVARCHAR(100)	NOT NULL
	A string name that identifies a vendor.	
Seq	INT	NOT NULL
(primary key)	Indicates the vendors sequence in the list.	

dbc_org_plist

The dbc_org_plist table contains information that associates an Organization with one or more standard purchase lists for buyers from that organization.

Column	Data Type	Constraint
org_id	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID of an organization.	
list_id	VARCHAR(40)	NOT NULL
	The repository ID of a purchase list.	
Tag	VARCHAR(40)	NOT NULL
(primary key)	A key that is used to identify the associated purchase list.	

User Profile Extensions

The following tables contain information about specific extensions for Oracle ATG Web Commerce user profiles:

dbc_cost_center

The `dbc_cost_center` table contains information related to a cost center. Cost centers are used by an organization for accounting purposes. These cost centers will be associated with either an organization or a user.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique repository ID that identifies a cost center.	
`identifier`	WVARCHAR(40)	NOT NULL
	A string identifier code that is used by a business for accounting purposes.	
`Description`	VARCHAR(254)	NULL
	A string description of the cost center. This description is for UI purposes to identify the cost center.	
`user_id`	VARCHAR(40)	NULL
	The ID of a user associated with the cost center.	

dbc_user

The `dbc_user` table provides additional user properties for commerce applications.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL

Column	Data Type	Constraint
(primary key)	References `dps_user(id)`. The unique repository ID that identifies a user.	
`price_list`	VARCHAR(40)	NULL
	The repository ID of a price list. This price list is then used to retrieve pricing information from that is specific to the user.	
`user_catalog`	VARCHAR(40)	NULL
	The repository ID of a catalog.	
`business_addr`	VARCHAR(40)	NULL
	The repository ID of a contact info item. This will become the users business address. References `dps_contact_info(id)`.	
`dflt_shipping_addr`	VARCHAR(40)	NULL
	The repository ID of a contact info item. This will becomes the users default shipping address. References `dps_contact_info(id)`.	
`dflt_billing_addr`	VARCHAR(40)	NULL
	The repository ID of a contact info item. This will become the users default billing address. References `dps_contact_info(id)`.	
`dflt_payment_type`	VARCHAR(40)	NULL
	The repository ID of a credit card item. This will become the users default credit card. References `dps_credit_card(id)`.	
`dflt_cost_center`	VARCHAR(40)	NULL
	The repository ID of a cost center. This will become the users default cost center. References `dbc_cost_center(id)`.	
`order_price_limit`	NUMERIC(19, 7)	NULL
	The greatest amount that the user is able to purchase before triggering an approval condition.	
`approval_required`	TINYINT	NULL
	A flag indicating whether or not the user should be checked to see if they triggered an approval condition.	

dbc_buyer_costctr

The `dbc_buyer_costctr` table is used to associate many cost centers to a user.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user.	
`Seq`	VARCHAR(42)	NOT NULL
(primary key)	The sequence in a list that contains the cost center.	
`cost_center_id`	VARCHAR(40)	NOT NULL
	The repository ID of a cost center item. References `dbc_cost_center(id)`.	

dbc_buyer_approver

The `dbc_buyer_approver` table contains information that associates a buyer with one or more order approvers. Approvers must be registered users so they can perform online approvals.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user. References `dps_user(id)`	
`approver_id`	VARCHAR(40)	NOT NULL
	The repository ID of a user who should act as an approver for the user identified by the `user_id` column. References `dps_user(id)`.	
`Seq`	INT	NOT NULL
(primary key)	Indicates the approver's sequence in the list.	

dbc_buyer_payment

The `dbc_buyer_payment` table contains information that associates a buyer with one or more pre-approved payment types. A payment type is a credit card.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user. References `dps_user(id)`.	
`Tag`	VARCHAR(42)	NOT NULL
(primary key)	A key that identifies the associated credit card.	

Column	Data Type	Constraint
payment_id	VARCHAR(40)	NOT NULL
	The repository ID of a credit card. References dps_credit_card(id).	

dbc_buyer_shipping

The dbc_buyer_shipping table contains information that associates a buyer with one or more pre-approved shipping addresses.

Column	Data Type	Constraint
Column	**Data Type**	**Constraint**
user_id	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user. References dps_user(id).	
Tag	VARCHAR(42)	NOT NULL
(primary key)	A key to identify the associated address.	
addr_id	VARCHAR(40)	NOT NULL
	A repository ID of a contact info item. This item becomes a shipping address for the user. References dps_contact_info(id).	

dbc_buyer_billing

The dbc_buyer_billing table contains information that associates a buyer with one or more pre-approved billing addresses.

Column	Data Type	Constraint
user_id	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user. References dps_user(id).	
Tag	VARCHAR(42)	NOT NULL
(primary key)	A key by which the associated address will be identified by	
addr_id	VARCHAR(40)	NOT NULL
	The repository ID of a contact info item. This will become a billing address. References dps_contact_info(id).	

dbc_buyer_prefvndr

The `dbc_buyer_prefvndr` table contains information that associates a buyer with one or more preferred vendors.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user References `dps_user(id)`.	
`Vendor`	WVARCHAR(100)	NOT NULL
	A text name for a vendor.	
`Seq`	INT	NOT NULL
(primary key)	The sequence in a list that contains the vendor.	

dbc_buyer_plist

The `dbc_buyer_plist` table contains information that associates a buyer with one or more standard purchase lists.

Column	Data Type	Constraint
`user_id`	VARCHAR(40)	NOT NULL
(primary key)	The repository ID of a user.	
`list_id`	VARCHAR(40)	NOT NULL
	The repository ID of a purchase list.	
`Tag`	VARCHAR(40)	NOT NULL
(primary key)	A key which is used to identify the associated purchase list.	

Invoice Tables

The following tables contain information related to Oracle ATG Web Commerce invoice functionality:

- dbc_inv_delivery (page 668)
- dbc_inv_pmt_terms (page 670)
- dbc_invoice (page 670)

dbc_inv_delivery

The following table contains information about the delivery information associated with an invoice. Each row represents one `deliveryInfo` repository item.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The repository id of this `deliveryInfo` item.	
`version`	INT	NOT NULL
	The version number of the data in this row. See the *SQL Repository Item Properties* chapter of the *ATG Repository Guide* for information on automatic version numbering.	
`type`	INT	NOT NULL
	The subtype code for the `deliveryInfo` repository item represented by this row. (See the GSA docs on item descriptor subtypes for more info.)	
`prefix`	WVARCHAR(40)	NULL
	An optional prefix (“Mr.”, “Ms.”, etc.) that appears as part of this invoice recipient’s name.	
`first_name`	WVARCHAR(40)	NULL
	The first name of the recipient of this invoice.	
`middle_name`	WVARCHAR(40)	NULL
	The middle name of the recipient of this invoice.	
`last_name`	WVARCHAR(40)	NULL
	The last name of the recipient of this invoice.	
`suffix`	WVARCHAR(40)	NULL
	An optional suffix (“Jr.”, etc) that appears as part of this invoice recipient’s name.	
`job_title`	WVARCHAR(80)	NULL
	The invoice recipient’s job title, which an application might choose to use when constructing a mailing address for this invoice.	
`company_name`	WVARCHAR(40)	NULL
	The invoice recipient’s company name, which an application might choose to use when constructing a mailing address for this invoice.	

Column	Data Type	Constraint
address1	WVARCHAR(80)	NULL
	The first line of the delivery address for this invoice.	
address2	WVARCHAR(80)	NULL
	The second line of the delivery address.	
address3	WVARCHAR(80)	NULL
	The third line of the delivery address.	
city	WVARCHAR(40)	NULL
	The city part of the delivery address.	
county	WVARCHAR(40)	NULL
	The county part of the delivery address.	
state	WVARCHAR(40)	NULL
	The state part of the delivery address.	
postal_code	WVARCHAR(10)	NULL
	The postal code of the delivery address.	
country	WVARCHAR(40)	NULL
	The country of the delivery address.	
phone_number	WVARCHAR(40)	NULL
	The invoice recipient's telephone number.	
fax_number	WVARCHAR(40)	NULL
	The invoice recipient's FAX number.	
email_addr	WVARCHAR(40)	NULL
	The invoice recipient's e-mail address.	
format	INT	NULL
	An enumerated type code indicating the format in which the recipient prefers to receive this invoice. This field is not used by default, but is provided as a placeholder for applications that want to present a list of available formats (text, HTML, EDI, XML, etc) and attempt to deliver the invoice in a preferred format.	
delivery_mode	INT	NULL

Column	Data Type	Constraint
	An enumerated type code indicating the means by which the recipient prefers to receive this invoice. This field is not used by default, but is provided as a placeholder for applications that want to present a list of available delivery methods (postal, fax, electronic, etc) and attempt to deliver the invoice using the preferred means.	

dbc_inv_pmt_terms

The following table contains information related to the payment terms for an invoice. Each row represents one `paymentTerms` repository item.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The repository id of this `paymentTerms` item.	
`version`	INT	NOT NULL
	The version number of the data in this row. (See the GSA docs on automatic version numbering for more info.)	
`type`	INT	NOT NULL
	The subtype code for the `paymentTerms` repository item represented by this row. (See the GSA docs on item descriptor subtypes for more info.)	
`disc_percent`	NUMERIC(19, 7)	NULL
	The `discountPercent` field of the payment terms item. This typically specifies a percentage discount on the price that is offered if the invoice is paid in full within a certain number of days (the `discountDays`"field).	
`disc_days`	INT	NULL
	The `discountDays` field of the payment terms item. Specifies the number of days within which the invoice must be paid to qualify for the discount percentage.	
`net_days`	INT	NULL
	The `netDays` field of the payment terms item. Specifies the number of days within which the invoice must be paid in full.	

dbc_invoice

The following table contains information related to invoices. Each row represents one invoice repository item.

Column	Data Type	Constraint
`id`	VARCHAR(40)	NOT NULL
(primary key)	The unique repository id of this invoice.	
`version`	INT	NOT NULL
	The version number of the data in this row. (See the GSA docs on automatic version numbering for more info.)	
`type`	INT	NOT NULL
	The subtype code for the invoice repository item represented by this row. (For more information on item descriptor subtypes, see the *Item Descriptor Inheritance* section of the *SQL Repository Data Models* chapter in the *ATG Repository Guide*.)	
`creation_date`	TIMESTAMP	NULL
	The date this invoice was created.	
`last_mod_date`	TIMESTAMP	NULL
	The date this invoice was last modified. This field is maintained automatically when you use the `InvoiceManager` API to create and modify invoices.	
`invoice_number`	VARCHAR(40)	NULL
	An application-generated invoice number that identifies this invoice for both the buyer and the seller. The invoice number is different from the repository id, and has no meaning other than as a way to identify a particular invoice.	
`po_number`	VARCHAR(40)	NULL
	The purchase order number the buyer specified when paying for all or part of an order by invoice.	
`req_number`	VARCHAR(40)	NULL
	The requisition number the buyer specified when paying for all or part of an order by invoice.	
`delivery_info`	VARCHAR(40)	NULL
	References `dbc_inv_delivery(id)`. Identifies the `deliveryInfo` object that describes where and how to deliver this invoice.	
`balance_due`	NUMERIC(19, 7)	NULL
	The balance due on this invoice.	
`pmt_due_date`	TIMESTAMP	NULL

Column	Data Type	Constraint
	The date on which payment for this invoice is due.	
pmt_terms	VARCHAR(40)	NULL
	References dbc_inv_pmt_terms(id). Identifies the paymentTerms object that describes the payment terms for this invoice.	
order_id	VARCHAR(40)	NULL
	The order ID of the order being paid for with this invoice.	
pmt_group_id	VARCHAR(40)	NULL
	The payment group ID of the specific payment group being paid for with this invoice.	

Contract Tables

The following tables contain information related to Oracle ATG Web Commerce contract functionality.

- dbc_contract (page 672)
- dbc_contract_term (page 673)

dbc_contract

The following table contains information related to contract functionality.

Column	Data Type	Constraint
contract_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the price list.	
display_name	WVARCHAR(254)	NULL
	The name of this contract.	
creation_date	TIMESTAMP	NULL
	The date that this contract was created.	
start_date	TIMESTAMP	NULL,
	The date that this contract becomes active.	
end_date	TIMESTAMP	NULL,
	The date that this contract is no longer active.	
creator_id	VARCHAR(40)	NULL

Column	Data Type	Constraint
	The user id of the creator of this contract.	
negotiator_info	WVARCHAR(40)	NULL
	The id of the contact_info for the negotiator of this contract.	
price_list_id	VARCHAR(40)	NULL
	The id of the price list that users of this contract will use.	
catalog_id	VARCHAR(40)	NULL
	The id of the catalog that users of this contract will use.	
term_id	VARCHAR(40)	NULL
	The id of the contract terms that apply.	
comments	WVARCHAR(254)	NULL
	A free form comment field.	

dbc_contract_term

The following table contains information related to contract terms.

Column	Data Type	Constraint
terms_id	VARCHAR(40)	NOT NULL
(primary key)	The unique identifier associated with the price list.	
terms	LONG VARCHAR	NULL
	A text field for describing any terms.	
disc_percent	NUMERIC(19, 7)	NULL
	The default discount percent for invoices.	
disc_days	INT	NULL
	The default discount days for invoices.	
net_days	INT	NULL
	The default net days for invoices.	

Appendix C. Messages

This appendix describes messages sent out as part of Oracle ATG Web Commerce. These messages are also described throughout the ATG Commerce Programming Guide (page 1) guide in the context of systems that use the messages.

Base Oracle ATG Web Commerce Messages

The messages described in this section are defined in the base Oracle ATG Web Commerce layer.

Fulfillment System Messages

Fulfillment messages include some messages that are sent by default to the scenario manager, and messages intended for use within the fulfillment system.

The scenario messages are all sent by the `OrderChangeHandler` component. This component listens for internal fulfillment messages (such as `ModifyOrderNotification`) and creates messages for consumption by scenarios.

The following messages are sent to the scenario manager during fulfillment.

- `atg.commerce.fulfillment.scenario.OrderModified`
- `atg.commerce.fulfillment.scenario.PaymentGroupModified`
- `atg.commerce.fulfillment.scenario.ShippingGroupModified`

The following messages are used within the fulfillment system.

- atg.commerce.fulfillment.UpdateInventoryImpl
- atg.commerce.fulfillment.FulfillOrderFragment
- atg.commerce.fulfillment.ModifyOrderNotification
- atg.commerce.fulfillment.ModifyOrder

atg.commerce.fulfillment.SubmitOrder

The SubmitOrder message is sent when an order has been submitted for fulfillment (checkout complete).

Base type: CommerceMessageImpl
JMS type: atg.commerce.fulfillment.SubmitOrder

Source

The ProcSendFulfillmentMessage processor, which is part of the processOrder pipeline chain.

Properties

- order
- orderId
- originalId
- originalSource
- parentsessionId
- profile
- sessionId
- siteId
- sourceId
- type
- userId

atg.commerce.fulfillment.scenario.OrderModified

The OrderModified message is sent when an order is modified.

JMS type: atg.commerce.fulfillment.scenario.OrderModified
Base type: ScenarioEvent

Four possible kinds of modifications are accessible in the subType property:

- Order is finished
- Order has unavailable items
- Order is pending merchant action
- Order was removed

Source

The `OrderChangeHandler` component.

Properties

- `order`
- `shipItemRels` (the list of unavailable items)
- `subType`
- `type`

atg.commerce.fulfillment.scenario.PaymentGroupModified

The `PaymentGroupModified` event is sent when a payment group is modified.

Base type: `ScenarioEvent`
JMS type: `atg.commerce.fulfillment.scenario.PaymentGroupModified`

Three possible kinds of modifications are accessible in the `subType` property:

- Payment group has been credited
- Payment group has been debited
- Payment group has failed to debit

Source

The `OrderChangeHandler` component.

Properties

- `order`
- `paymentGroups`
- `subType`
- `type`

atg.commerce.fulfillment.scenario.ShippingGroupModified

The `ShippingGroupModified` event is sent when a shipping group is modified.

Base type: `ScenarioEvent`
JMS type: `atg.commerce.fulfillment.scenario.ShippingGroupModified`

Six possible kinds of modifications are accessible in the `subType` property:

- Shipping group has been removed
- Shipping group shipped
- Shipping group split for unknown reasons
- Shipping group split to partially ship

- Shipping group split to use multiple fulfillers
- Shipping group is pending merchant action

Source

The `OrderChangeHandler` component.

Properties

- `order`
- `shippingGroup`
- `newShippingGroup` (if split)
- `subType`
- `type`

atg.commerce.fulfillment.UpdateInventoryImpl

The `UpdateInventoryImpl` message is used if there is new inventory available.

Base type: `CommerceMessageImpl`
JMS type: `atg.commerce.fulfillment.UpdateInventoryImpl`

Source

The `RepositoryInventoryManager` sends this whenever `inventoryWasUpdated` is called. This method is called by fulfillment if an order is cancelled. It may also be called by administrative tools. Oracle ATG Web Commerce includes an `InventoryFormHandler` that calls this method and is accessible through the Dynamo Server Admin pages.

Properties

- `itemIds`
- `type`

atg.commerce.fulfillment.FulfillOrderFragment

This message is used within fulfillment to notify a particular fulfiller of new items to fulfill. The `SubmitOrder` message includes the entire order. Fulfillment splits this message into one message per fulfiller. The `FulfillOrderFragment` message includes this per-fulfiller information.

Base type:`CommerceMessageImpl`
JMS type: `atg.commerce.fulfillment.FulfillOrderFragment`

Source

The source is `ProcSendFulfillOrderFragment`, which is one of the fulfillment pipeline processors

Properties

- `order`
- `shippingGroupIds`—Shipping groups for which the fulfiller is responsible.
- `type`

atg.commerce.fulfillment.ModifyOrderNotification

Base type: `CommerceMessageImpl`
JMS type: `atg.commerce.fulfillment.ModifyOrderNotification`

The `ModifyOrderNotification` message is used to announce that changes were made to an order during fulfillment. Each change is represented as a `Modification` object. Modifications can tell you things like which payment groups have changed, which shipping groups have changed, or when the state of an order object has changed.

Source

The fulfillment system can send this message when an order changes.

Properties

- `modifyOrderSource` (If this message was a response to a `ModifyOrder` message, this property references the source of that `ModifyOrder` message.)
- `modifications`
- `orderId`
- `originalSource`
- `originalId`
- `parentSessionId`
- `sessionId`
- `siteId`
- `sourceId`
- `type`
- `userId`

atg.commerce.fulfillment.ModifyOrder

The `ModifyOrder` message is used to request that a change be made to an order during fulfillment. Each change request is represented as a `Modification` object. Examples of modifications are to remove an order or change the state of this order object.

Base type: `CommerceMessageImpl`
JMS type: `atg.commerce.fulfillment.ModifyOrder`

Source

This message is sent by the external fulfillment system. Whenever a change needs to be made by another part of fulfillment, a `ModifyOrder` message is sent.

Properties

- `modifications`
- `orderId`
- `originalId`

- originalSource
- parentSessionId
- siteId
- sourceId
- sessionId
- type
- userId

Order and Pricing Messages

The messages described in this section are used during order-related changes.

atg.commerce.gifts.GiftPurchased

The GiftPurchased message is sent out when a gift is purchased off of a gift list.

Base type: CommerceMessageImpl
JMS type: atg.commerce.gifts.GiftPurchased

Source

The ProcSendGiftPurchasedMessage processor is part of the processOrder pipeline chain.

Properties

- item
- order
- originalId
- originalSource
- parentSessionId
- profile
- sessionId
- siteId
- sourceId
- type
- userId

atg.commerce.inventory.InventoryThresholdReached

The InventoryThresholdReached message is sent out when the inventory for a particular item dips below some threshold.

JMS type: atg.commerce.inventory.InventoryThresholdReached

Source

The `RepositoryInventoryManager` sends this message whenever an inventory level is decremented below its corresponding threshold.

Properties

- `type`
- `id`
- `inventoryId`
- `levelPropertyName`
- `thresholdPropertyName`
- `currentValue`
- `thresholdValue`

atg.commerce.order.ItemAddedToOrder

The `ItemAddedToOrder` message is sent out when an item was added to an order either before or after checkout.

Base type: `ScenarioEvent`
JMS type: `atg.commerce.order.ItemAddedToOrder`

Source

`CartModifiedFormHandler` and web services.

Properties

- `id`
- `order`
- `catalogRef`
- `product`
- `commerceItem`
- `quantity`
- `amount`
- `type`
- `siteId`

atg.commerce.order.ItemRemovedFromOrder

The `ItemRemovedFromOrder` message is sent out when an item is removed from an order either before or after checkout.

Base type: `ScenarioEvent`

JMS type: `atg.commerce.order.ItemRemovedFromOrder`

Source

`CartModifiedFormHandler` and web services.

Properties

- `order`
- `catalogRef`
- `product`
- `commerceItem`
- `quantity`
- `amount`
- `type`
- `siteId`

atg.commerce.order.OrdersMerged

An `OrdersMerged` message is sent whenever orders are merged.

Base type: `ScenarioEvent`
JMS type: `atg.commerce.order.OrdersMerged`

Source

`OrdersMerged` is sent from `OrderManager.mergeOrders`, which is called by `CommerceProfileTools.loadShoppingCarts` when a transient order in the session is merged with a saved order on the profile.

Properties

- `siteId`
- `sourceOrder`
- `destinationOrder`
- `sourceRemoved` (boolean)

atg.commerce.order.ItemQuantityChanged

The `ItemQuantityChanged` message is sent out when the quantity of an item changes in an order either before or after checkout.

Base type: `ScenarioEvent`
JMS type: `atg.commerce.order.ItemQuantityChanged`

Source

`CartModifierFormHandler`

Properties

- id
- order
- catalogRef
- product
- commerceItem
- oldQuantity
- newQuantity
- type
- siteId

atg.commerce.order.scheduled.ScheduledOrderMessage

The ScheduledOrderMessage is sent for any action on a scheduled order. You can tell what the action was by looking at the messages action property which is of type ScheduledOrderAction (Enum).

Base type: CommerceMessageImpl
JMS type: atg.commerce.order.scheduled.ScheduledOrderMessage

Actions

- Created
- Updated
- Deleted
- Submited *

  This action is spelled "submited."

Source

ScheduledOrderTools sends messages for all four actions.

Properties

- action
- originalId
- originalSource
- parentSessionId
- profile
- scheduledOrder
- sessionId
- siteId
- sourceId

- `type`
- `userId`

atg.commerce.order.SwitchOrder

The `ShoppingCart` can contain multiple saved orders; the `current` property indicates which order is the current order. The `SwitchOrder` message is sent when the user changes which order is the current order.

Base type: `CommerceMessageImpl`
JMS type: `atg.commerce.order.SwitchOrder`

Source

The shopping cart.

Properties

- `oldOrder`
- `order`
- `originalId`
- `originalSource`
- `parenSessionId`
- `sessionId`
- `sourceId`
- `siteId`
- `type`
- `userId`

atg.commerce.order.OrderSaved

The `OrderSaved` message is sent when order has been saved by the user and is moved to their list of saved orders (this is not the same as saving an order into the repository).

Base type: `CommerceMessageImpl`
JMS type: `atg.commerce.order.OrderSaved`

Source

`SaveOrderFormHandler`

Properties

- `order`
- `originalId`
- `originalSource`
- `parentSessionId`

- `sessionId`
- `siteId`
- `sourceId`
- `type`
- `userId`

atg.commerce.pricing.PriceChanged

The `PriceChanged` message is sent when the price of an order changes.

JMS type: `atg.commerce.pricing.PriceChanged`

Source

`PricingTools`, whenever it is used to price an order total or an order subtotal.

Properties

- `id`
- `type`
- `priceChangeType`
- `profile`
- `order`
- `repricedObject`
- `oldPrice`
- `newPrice`

Promotion Messages

The messages described in this section are used by the Oracle ATG Web Commerce promotions features. For additional information on how these messages are used, see the *ATG Commerce Guide to Setting Up a Store*.

atg.commerce.promotion.ScenarioAddedItemToOrder

The `ScenarioAddedItemToOrder` message is sent out when a scenario is used to add an item to the shopping cart.

Base type: `ItemAddedToOrder`
JMS type: `atg.commerce.promotion.ScenarioAddedItemToOrder`

Source

The `AddItemToOrder` scenario action.

Properties

- `id`

- order
- catalogRef
- product
- commerceItem
- quantity
- amount
- type
- siteId

atg.commerce.promotion.PromotionUsed

The PromotionUsed message is sent out when a promotion is used by an order that was submitted.

Base type: CommerceMessageImpl
JMS type: atg.commerce.promotion.PromotionUsed

Source

The ProcSendPromotionUsedMessage processor, which is part of the processOrder pipeline chain.

Properties

- order
- originalId
- originalSource
- parentSessionId
- profile
- promotion
- sessionId
- sourceId
- siteId
- type
- userId

atg.commerce.promotion.PromotionGrantedMessage

The PromotionGranted message is sent out when a promotion is granted to a user.

Base type: CommerceMessageImpl

JMS type: atg.commerce.promotion.PromotionGranted

Source

`PromotionTools.sendPromotionGrantedEvent` is called by `PromotionTools.addPromotion` which is called by anyone adding a promotion to the profile.

Properties

- `originaId`
- `originalSource`
- `parentSessionId`
- `profile`
- `promotionId`
- `sessionId`
- `sourceId`
- `siteId`
- `type`
- `userId`

atg.commerce.promotion.PromotionRevokedMessage

A `PromotionRevoked` message is sent out when a promotion is revoked from a user.

Base type: `CommerceMessageImpl`

JMS type: `atg.commerce.promotion.PromotionRevoked`

Source

The source for the `PromotionRevokedMessage` is `PromotionTools.sendPromotionRevokedEvent`, which is called by `PromotionTools.revokePromotion`, which is called by anyone removing a promotion from the profile.

Properties

- `originalId`
- `originalSource`
- `parentSessionId`
- `profile`
- `promotionId`
- `sessionId`
- `siteId`
- `sourceId`
- `type`
- `userId`

atg.commerce.pricing.PromotionClosenessMessage

A `PromotionClosenessMessage` is sent when an order meets a given promotion's `closenessQualification`, and when an order that previously met a `closenessQualification` now fails to do so.

Base type: `CommerceMessageImpl`

JMS type: varies

- `atg.commerce.promotion.PromotionClosenessQualificationEvent`
- `atg.commerce.promotion.PromotionClosenessDisqualificationEvent`

Source

`PricingTools`

Properties

- `closenessQualifier`
- `order`
- `originalId`
- `originaSource`
- `parentSessionId`
- `profile`
- `sessionId`
- `siteId`
- `sourceId`
- `type`
- `userId`

Abandoned Order Messages

The following message class is provided as part of the `DCS.AbandonedOrderServices` module. For detailed information on this module, see the *Using Abandoned Order Services* (page 453) chapter.

atg.commerce.order.abandoned.OrderAbandoned

An `OrderAbandoned` message is sent out when an order's abandonment state is changed.

Base type: `CommerceMessageImpl`

JMS type: varies

- `atg.commerce.order.abandoned.OrderAbandoned`
  Sent when an order is identified as abandoned.
- `atg.commerce.order.abandoned.OrderReanimated`
  Sent when an order is identified as reanimated.
- `atg.commerce.order.abandoned.OrderConverted`
  Sent when an order is identified as converted.
- `atg.commerce.order.abandoned.OrderLost`
  Sent when an order is identified as lost.

Source

This message is sent by the `/atg/commerce/order/abandoned/AbandonedOrderMessageFactory` component.

Properties

- `abandonmentState`
- `orderId`
- `originalId`
- `originalSource`
- `parentSessionId`
- `profileId`
- `sessionId`
- `siteId`
- `sourceId`
- `type`
- `userId`

atg.commerce.order.abandoned.TransientOrderEvent

A `TransientOrderEvent` message is sent when the `AbandonedOrderEventListener` checks the profile associated with a submitted order. See AbandonedOrderEventListener (page 472).

Base type: `CommerceMessageImpl`

Source

A `TransientOrderEvent` is sent by the `AbandonedOrderEventListener`.

Properties

- `amount`
- `currencyCode`
- `orderId`

- originalId
- originalSource
- parentSessionId
- profile
- sessionId
- siteId
- sourceId
- submitted
- userId

Approval Messages

These messages are used in the approvals feature.

- atg.commerce.approval.ApprovalRequiredMessage
- atg.commerce.approval.ApprovalUpdate

atg.commerce.approval.ApprovalRequiredMessage

The ApprovalRequired message is sent if there is an order that requires approval before it can be submitted.

Base type: CommerceMessageImpl
JMS type: atg.commerce.approval.ApprovalRequired

Source

- ProcSendApprovalRequiredMessage - Part of the approval pipeline which is executed during the process order pipeline.

Properties

- order
- originalId
- originalSource
- parentSessionId
- profile
- sessionId
- siteId
- sourceId

- `type`
- `userId`

atg.commerce.approval.ApprovalUpdate

The `ApprovalUpdate` message is sent with one of two JMS types.

Base type: `CommerceMessageImpl`
JMS type: varies

- `atg.commerce.approval.ApprovalUpdate`
  An approval update message will have one of two statuses: `approved` or `rejected`.
- `atg.commerce.approval.ApprovalComplete`
  An approval update message will have one of two statuses: `approval_passed` or `approval_failed`.

Source

`ProcSendApprovalCompleteMessage` or `ProcSendApprovalMessage`. Both of these are in pipeline chains that are executed by the `ApprovalFormHandler`.

Properties

- `approvalStatus`
- `order`
- `orderOwnerProfile`
- `originalId`
- `originalSource`
- `parentSessionId`
- `profile`
- `sessionId`
- `siteId`
- `sourceId`
- `type`
- `userId`

Invoice Messages

These message is used in the invoicing feature.

atg.commerce.invoice.messaging.InvoiceMessage

JMS type: varies

- `atg.commerce.invoice.scenario.CreateInvoice`
  A new invoice repository item was created.
- `atg.commerce.invoice.scenario.UpdateInvoice`
  An existing invoice repository item was modified.
- `atg.commerce.invoice.scenario.RemoveInvoice`
  An existing invoice repository item was removed.

Source

- Each message is sent by a different processor in the invoice pipeline (used by the `InvoiceManager`).

Properties

- `invoiceRepositoryId`
- `orderId`
- `paymentGroupId`
- `profile`
- `invoiceNumber`
- `PONumber`
- `requisitionNumber`
- `billingAddress`
- `preferredFormat`
- `preferredDeliveryMode`
- `balanceDue`
- `paymentDueDate`
- `paymentNetDays`
- `paymentDiscountDays`
- `paymentDiscountPercent`
- `siteId`

Appendix D. Scenario Recorders

The following scenario recorders are provided in Oracle ATG Web Commerce:

dcs (page 693)

dcs-analytics (page 694)

shoppingprocess (page 696)

See the reference entries that follow for recorder descriptions and supporting elements. You can access the datasets, mappers, and data collection objects, respectively, in the CONFIGPATH at:

`/atg/registry/data/datasets/`

`/atg/registry/data/mappers/`

`/atg/reporting/dataset/`

For general information on scenario recorders, including information on creating custom recorders, see the *ATG Personalization Programming Guide*.

dcs

This scenario recorder records events that affect an order's items.

Recorded Event	Supporting Elements

Item added to order	Dataset:
	Item added to order
	(`itemaddedtoorder.xml`)
	
	Mapper:
	DCS Cart Event SQL Mapper (`cartsqlmapper.xml`)
	
	Data collection object:
	`DCSCartSQLLogger`
	
	Database table:
	`dcs_cart_event`

Recorded Event	Supporting Elements

Item removed from order	Dataset:
	Item removed from order
	(`itemremovedfromorder.xml`)
	
	Mapper:
	DCS Cart Event SQL Mapper (`cartsqlmapper.xml`)
	
	Data collection object:
	`DCSCartSQLLogger`
	
	Database table:
	`dcs_cart_event`

A scenario added an item to an order	Dataset:
	Scenario added item to order
	(`scenarioaddeditemtoorder.xml`)
	
	Mapper:
	DCS Cart Event SQL Mapper (`cartsqlmapper.xml`)
	
	Data collection object:
	`DCSCartSQLLogger`
	
	Database table:
	`dcs_cart_event`

dcs-analytics

This scenario recorder records various order events.

Recorded Event	Supporting Elements

Order submitted	Dataset:
	Order Submitted
	(`submitorderevent.xml`)
	
	Mapper:
	Submit Order Event SQL Mapper
	(`submitordermapper.xml`)
	
	Data collection object:
	`DCSSubmitOrderSQLLogger`
	
	Database Table:
	`dcs_submt_ord_evt`

Recorded Event	Supporting Elements

Uses promotion	Dataset:
	Promotion Used
	(`promotionusedevent.xml`)
	
	Mapper:
	Promotion Used Event SQL Mapper
	(`promotionusedmapper.xml`)
	
	Data collection object:
	`DCSPromotionUsedSQLLogger`
	
	Database table:
	`dcs_prom_used_evt`

Orders merged	Dataset:
	Orders Merged
	(`ordersmergedevent.xml`)
	
	Mapper:
	Orders Merged Event SQL Mapper
	(`ordersmergedmapper.xml`)
	
	Data collection object:
	`DCSOrdersMergedSQLLogger`
	
	Database table:
	`dcs_ord_merge_evt`

Item quantity changed in order	Dataset:
	Item quantity changed in order
	(`itemquantitychanged.xml`)
	
	Mapper:
	DCS Item Quantity Changed Event SQL Mapper
	(`itemquantitychangedmapper.xml`)
	
	Data collection object:
	`DCSItemQuantityChangedSQLLogger`
	
	Database table:
	`dcs_cart_event`

Recorded Event	Supporting Elements

Promotion offered	Dataset:
	DCS Promotion Granted
	(`promotiongranted.xml`)
	
	Mapper:
	Promotion Granted SQL Mapper
	(`promotiongrantedmapper.xml`)
	
	Data collection object:
	`PromotionGrantedLoggerQueue`
	
	Database table:
	`dcs_promo_grntd`

Promotion revoked	Dataset:
	DCS Promotion Revoked
	(`promotionrevoked.xml`)
	
	Mapper:
	Promotion Revoked SQL Mapper
	(`promotionrevokedmapper.xml`)
	
	Data collection object:
	`PromotionRevokedLoggerQueue`
	
	Database table:
	`dcs_promo_rvkd`

shoppingprocess

This scenario recorder records events generated when an order reaches a new stage in the shopping process.

Recorded Event	Supporting Elements

Shopping Process Stage Reached	Dataset:
	Shopping Process Stage Reached
	(`shoppingprocess.xml`)
	
	Mapper:
	Business Process Stage Reached SQL Mapper
	(`bpstage_reached_mapper.xml`)
	
	Data collection object:
	`BusinessProcessStageReachedSQLLoggerQueue`
	
	Database table:
	`drpt_stage_reached`

Appendix E. Session Backup

By default, Oracle ATG Web Commerce adds several property values to the list of values that are written to the session backup server after every request. The property values that are added preserve the following for the user:

- current and saved orders
- active promotions
- products currently being compared
- event name for the current gift list

The properties that store this information are specified by layering on the following configuration file for the central configuration component, `/atg/dynamo/Configuration`.

```
# Add orders, promotions, and product comparisons to the list of
# items that are restored on session failover.
sessionBackupServerPropertyList+=\
     /atg/commerce/ShoppingCart.restorableOrders,\
     /atg/userprofiling/ProfileFailService.activePromotions,\
     /atg/commerce/catalog/comparison/ProductList.items,\
     /atg/commerce/gifts/GiftlistFormHandler.eventName
```

The configuration file is located at `<ATG10dir>/DCS/config/`.

For more information on backing up sessions, see the *ATG Installation and Configuration Guide*.

For information on session failover and migration, see the documentation for your application server.

Appendix F. Pipeline Chains

This appendix describes the pipeline chains included with Oracle ATG Web Commerce and their component links. For general information on pipelines, see the *Processor Chains and the Pipeline Manager* (page 363) chapter. This appendix contains the following sections:

Core Commerce Pipelines

This section describes the pipelines that make up core Commerce functionality.

updateOrder Pipeline Chain

The `updateOrder` pipeline saves the `Order` supplied to it. The `updateOrder` pipeline chain is executed by the `updateOrder()` method in the `OrderManager`. The `updateOrder()` method adds the given Order and the `OrderManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`updateOrderObject`

Saves the properties in the Order object.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveOrderObject`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveOrderObject`

Transitions: return value of 1 executes `updateCommerceItemObjects`.

`updateCommerceItemObjects`

Saves the properties in the `CommerceItem` objects in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveCommerceItemObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveCommerceItemObjects`

Transitions: return value of 1 executes `updateShippingGroupObjects`.

`updateShippingGroupObjects`

Saves the properties in the `ShippingGroup` objects in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveShippingGroupObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveShippingGroupObjects`

Transitions: return value of 1 executes `updateHandlingInstructionObjects`.

`updateHandlingInstructionObjects`

Saves the properties in the `HandlingInstruction` objects in all the `ShippingGroups` in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveHandlingInstructionObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveHandlingInstructionObjects`

Transitions: return value of 1 executes `updatePaymentGroupObjects`.

`updatePaymentGroupObjects`

Saves the properties in the `PaymentGroup` objects in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SavePaymentGroupObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSavePaymentGroupObjects`

Transitions: return value of 1 executes `updatePaymentStatusObjects`.

`updatePaymentStatusObjects`

Saves the properties in the `PaymentStatus` objects in all the `PaymentGroups` in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SavePaymentStatusObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSavePaymentStatusObjects`

Transitions: return value of 1 executes `updateRelationshipObjects`.

`updateRelationshipObjects`

Saves the properties in the Relationship objects in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveRelationshipObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveRelationshipObjects`

Transitions: return value of 1 executes `updatePriceInfoObjects`.

`updatePriceInfoObjects`

This processor saves the properties in the `OrderPriceInfo` and `TaxPriceInfo` objects in the `Order`, the `ShippingPriceInfo` object in the `ShippingGroups`, and the `ItemPriceInfo` object in the `CommerceItems`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SavePriceInfoObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSavePriceInfoObjects`

Transitions: return value of 1 will execute `updateCostCenterObjects` next

`ProcSavePriceInfoObjects` includes an `orderStateSaveModes` property, which you can use to map order states (see Order States (page 258) in the *Working With Purchase Process Objects* (page 223) chapter) to save modes,which determine which types of PriceInfo object are saved. The valid save modes are:

- ALL—Saves all `PriceInfo` types.
- ALL_NO_AUDIT—Saves all `PriceInfo` types, but does not save audit trail information (pricing adjustments and detailed price info objects)
- ORDER—Saves only the `OrderPriceInfo` object (not shipping, item, tax)
- ORDER_NO_AUDIT—Saves only the `OrderPriceInfo` object, with no audit information
- NONE—Saves no pricing information

For example:

```
orderStateSaveModes=INCOMPLETE=ALL_NO_AUDIT
```

`ProcSavePriceInfoObjects` also includes a `defaultSaveMode` to use if the current order state does not have an entry in the `orderStateSaveModes` map.

updateCostCenterObjects

Transactional Mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveCostCenterObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveCostCenterOjbects`

Transitions: return value of 1 execute `saveManualAdjustments`.

`saveManualAdjustments`

Updates the order for any manual adjustments made by agents in Oracle ATG Web Commerce Service Center. If you are not using Commerce Service Center, this step does not apply. See Warning below.

Transactional Mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SaveManualAdjustments`

PipelineProcessor object: `atg.commerce.order.processor.ProcSaveManualAdjustments`

Transitions: return value of 1 execute `setLastModifiedTime`.

Warning: Manual adjustments are changes to an order made by an agent using Oracle ATG Web Commerce Service Center. Manual adjustments are applied unconditionally by the `OrderAdjustmentCalculator` (see the *ATG Commerce Service Center User Guide* for information on this class). Once an adjustment has been added, it affects the order's price regardless of the order's contents. If an adjustment is applied to an incomplete order, changes at checkout time do not affect the adjustment; for example, if a $20 credit is manually applied to an incomplete order, and the customer removes items, the order could end up with a $0 total. By default, the processor is configured not to save manual adjustments for incomplete orders, in which case the adjustment is discarded if the agent does not check out the order. However, this behavior is configurable. If you want to apply manual adjustments to incomplete orders, set the `saveIncompleteOrderAdjustments` property to true in the `/atg/commerce/order/processor/SaveManualAdjustments` component. To save manual adjustments for orders in additional states, adjust the configuration of the `incompleteStates` property.

```
# The processor will save the manual adjustments to the repository for orders
# in these states, depending on the value of saveIncompleteOrderAdjustments.

incompleteStates^=/atg/commerce/order/OrderLookupService.incompleteStates

# The processor will save the manual adjustments to the repository for orders
# in the configured incomplete states if this property is true. Otherwise, the
# manual adjustments are not saved for incomplete orders.

saveIncompleteOrderAdjustments=false
```

`setLastModifiedTime`

Sets the `lastModifiedTime` property of an Order to the current time if any changes were made to an Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SetLastModifiedTime`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetLastModifiedTime`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

loadOrder Pipeline Chain

The `loadOrder` pipeline chain loads the Order from the repository whose ID is supplied as a parameter. The `loadOrder` pipeline chain is executed by the `loadOrder()` method in the `OrderManager`. The `loadOrder()` method adds the given `OrderId`, `OrderRepository`, `CatalogTools` Nucleus component, and the `OrderManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`loadOrderObject`

This processor constructs a new Order object and loads the properties from the repository into it.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadOrderObject`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadOrderObject`

Transitions: return value of 1 will execute `loadPriceInfoObjectsForOrder` next.

`loadPriceInfoObjectsForOrder`

This processor constructs a new `OrderPriceInfo` and `TaxPriceInfo` object for the Order it loads and loads the properties from the repository into it. It then sets the `PriceInfo` to the corresponding object in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadPriceInfoObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadPriceInfoObjects`

Transitions: None. This is the last link in the chain and will cause the `PipelineManager` to return to the caller.

refreshOrder Pipeline Chain

The `refreshOrder` pipeline chain reloads an Order from the repository. The Order object is supplied as a parameter. The `refreshOrder` pipeline chain is not executed explicitly, but rather by the Oracle ATG Web Commerce components. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`loadOrderObjectForRefresh`

This processor takes an existing Order object and reloads the properties from the repository into it. It also loads all the supporting objects, such as `CommerceItem`, `ShippingGroup`, etc.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadOrderObject`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadOrderObject`

Transitions: return value of 1 will execute `loadCommerceItemObjects` next

`loadCommerceItemObjects`

This processor constructs a new `CommerceItem` object for each item it loads and loads the properties from the repository into it. It then adds the object to the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadCommerceItemObjects`

PipelineProcessor object**:** `atg.commerce.order.processor.ProcLoadCommerceItemObjects`

Transitions: return value of 1 will execute `loadShippingGroupObjects` next

`loadShippingGroupObjects`

This processor constructs a new `ShippingGroup` object for each item it loads and loads the properties from the repository into it. It then adds the object to the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadShippingGroupObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadShippingGroupObjects`

Transitions: return value of 1 will execute `loadHandlingInstructionsObjects` next

`loadHandlingInstructionsObjects`

This processor constructs a new `HandlingInstruction` object for each `ShippingGroup` that was loaded in the previous processor and loads the properties from the repository into it. It then adds the object to the `ShippingGroup` to which it belongs.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadHandlingInstructionObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadHandlingInstructionObjects`

Transitions: return value of 1 will execute `loadPaymentGroupObjects` next

`loadPaymentGroupObjects`

This processor constructs a new `PaymentGroup` object for each item it loads and loads the properties from the repository into it. It then adds the object to the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadPaymentGroupObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadPaymentGroupObjects`

Transitions: return value of 1 will execute `loadCostCenterObjects` next

`loadCostCenterObjects`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadCOstCenterObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcCostCenterObjects`

Transitions: return value of 1 will execute `loadPaymentStatusObjects` next

`loadPaymentStatusObjects`

This processor constructs a new `PaymentStatus` object for each `PaymentGroup` that was loaded in the previous processor and loads the properties from the repository into it. It then adds the object to the `PaymentGroup` to which it belongs.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadPaymentStatusObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadPaymentStatusObjects`

Transitions: return value of 1 will execute `loadRelationshipObjects` next

`loadRelationshipObjects`

This processor constructs a new Relationship object for each item it loads and loads the properties from the repository into it. It then adds the object to the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadRelationshipObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadRelationshipObjects`

Transitions: return value of 1 will execute `loadPriceInfoObjects` next

`loadPriceInfoObjects`

This processor constructs a new `OrderPriceInfo`, `TaxPriceInfo`, `ShippingPriceInfo`, or `ItemPriceInfo` object for each item it loads and loads the properties from the repository into it. It then sets the `PriceInfo` to the corresponding object in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/LoadPriceInfoObjects`

PipelineProcessor object: `atg.commerce.order.processor.ProcLoadPriceInfoObjects`

Transitions: return value of 1 will execute `setCatalogRefs` next

`setCatalogRefs`

This processor sets the `catalogRef` property in the `auxiliaryData` object in the `CommerceItem`. It does this by loading the `RepositoryItem` object using the `catalogRefId` in the `auxiliaryData` object. Additionally, if `SetCatalogRefs.substituteRemovedSku` is true, this processor replaces all deleted SKUs in the `Order` with the "dummy" SKU defined by `SetCatalogRefs.substituteDeletedSkuId`. For more information, see Refreshing Orders (page 265) in the *Configuring Purchase Process Services* chapter.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SetCatalogRefs`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetCatalogRefs`

Transitions: Return value of 1 executes `setProductRefs` next.

`setProductRefs`

This processor sets the `productRef` property in the `auxiliaryData` object in the `CommerceItem`. It does this by loading the `RepositoryItem` object using the `productId` in the `auxiliaryData` object. Additionally, if `SetProductRefs.substituteRemovedProduct` is true, this processor replaces all deleted products in the `Order` with the "dummy" product defined by `SetProductRefs.substituteDeletedProductId`. For more information, see Refreshing Orders (page 265) in the *Configuring Purchase Process Services* chapter.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SetProductRefs`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetProductRefs`

Transitions: Return value of 1 executes `removeExpiredCommerceItems` next.

`removeExpiredCommerceItems`

Used in conjunction with `SetCatalogRefs` and `SetProductRefs`. If the state of the `Order` is one that is defined in `RemoveExpiredCommerceItems.openOrderStates`, this processor removes from the `Order` any `CommerceItem` that contains a "dummy" SKU or product that was substituted by `SetCatalogRefs` or `SetProductRefs`. A "dummy" SKU is automatically removed. A "dummy" product is removed only if `RemoveExpiredCommerceItems.removeItemsWithDeletedProducts` is set to true; the default is true. For more information, see Refreshing Orders (page 265) in the *Configuring Purchase Process Services* chapter.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/RemoveExpiredCommerceItems`

PipelineProcessor object: `atg.commerce.order.processor.ProcRemoveExpiredCommerceItems`

Transitions: None, this is the last link in the chain, and will cause the `PipelineManager` to return to the caller.

repriceOrderForInvalidation Pipeline Chain

This chain reprices an order when it is invalidated. It includes a single link. The following section describes the processor in the pipeline chain.

`executeRepriceOrderChain`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/pricing/processor/ExecuteRepriceOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: None, this is the last link in the chain, and will cause the `PipelineManager` to return to the caller.

processOrderWithReprice Pipeline Chain

This chain processes an unpriced Order. The following sections describe each processor in the pipeline chain.

executeRepriceOrderChainForProcessOrder

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/pricing/processor/ExecuteRepriceOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes `executeProcessOrderAfterReprice` next.

executeProcessOrderAfterReprice

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ExecuteProcessOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: None, this is the last link in the chain, and will cause the `PipelineManager` to return to the caller.

processOrder Pipeline Chain

The `processOrder` pipeline chain submits the given Order for checkout. The `processOrder` pipeline chain is executed by the `processOrder()` method in the `OrderManager`. The `processOrder()` method adds the given Order, Profile, Request, Locale, and `OrderManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`executeValidateForCheckoutChain`

This processor causes the `validateForCheckout` chain to be executed. If the execution of this chain causes any errors, then execution will be returned to the caller.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ExecuteValidateForCheckoutChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes `setStimulusMarkers` next.

setStimulusMarkers

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SetStimulusMarkers`

PipelineProcessor object: `atg.commerce.order.processor.SetStimulusMarkers`

Transitions: Return value of 1 executes `setSalesChannel` processor.

setSalesChannel

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SetSalesChannel`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetSalesChannel`

Transitions: Return value of 1 executes `setSubmittedSite` processor.

setSubmittedSite

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SetSubmittedSite`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetSubmitedSite`

Transitions: Return value of 1 executes `executeApproveOrderChain` next.

`executeApproveOrderChain`

This processor executes the `approveOrder` pipeline chain, which begins the order approval process.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/ExecuteApproveOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes `stopChainIfOrderRequiresApproval` next. Return value of -1 (STOP_CHAIN_EXECUTION_AND_ROLLBACK) stops the execution of the `processOrder` chain; this means that an error occurred.

`stopChainIfOrderRequiresApproval`

This processor checks whether the order has been determined to require approval. Specifically, it checks whether the state of the order is PENDING_APPROVAL. If it isn't, the order moves to the next processor in `processOrder`. If it is, execution of the `processOrder` chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/StopChainIfOrderRequiresApproval`

PipelineProcessor object: `atg.commerce.order.processor.ProcCheckOrderState`

Transitions: Return value of 1 executes `executeValidatePostApprovalChain` next. Return value of 2 executes `executeValidateNoApprovalChain`.

`executeValidatePostApprovalChain`

If the order requires approval and has been approved, this processor revalidates order information in case the approver changed anything.

Transactional mode: TX_MANDATORY

Nucleus component: `atg/commerce/order/processor/ExecuteValidatePostApprovalChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes `checkForExpiredPromotions` next.

`executeValidateNoApprovalChain`

If the order does not require approval, finish validation.

Transactional mode: TX_MANDATORY

Nucleus component: `atg/commerce/order/processor/ExecuteValidateNoApprovalChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes `checkForExpiredPromotions`.

`checkForExpiredPromotions`

This processor walks through all the promotions that are being applied to the Order and verifies that none of them have expired.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/CheckForExpiredPromotions`

PipelineProcessor object: `atg.commerce.order.processor.ProcCheckForExpiredPromotions`

Transitions: return value of 1 will execute `removeEmptyShippingGroups` next

`removeEmptyShippingGroups`

This processor checks to see if there are any empty `ShippingGroups` in the Order. It removes any empty groups it finds. An empty `ShippingGroup` contains no Relationships. If the Order contains only one `ShippingGroup` then it will not be removed if it is empty.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/RemoveEmptyShippingGroups`

PipelineProcessor object: `atg.commerce.order.processor.ProcRemoveEmptyShippingGroups`

Transitions: return value of 1 will execute `removeEmptyPaymentGroups` next

`removeEmptyPaymentGroups`

This processor checks to see if there are any empty `PaymentGroups` in the Order. If so then it will remove them. An empty `PaymentGroup` contains no Relationships. If the Order contains only one `PaymentGroup` then it will not be removed if it is empty.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/RemoveEmptyPaymentGroups`

PipelineProcessor object: `atg.commerce.order.processor.ProcRemoveEmptyPaymentGroups`

Transitions: return value of 1 will execute `createImplicitRelationships` next

`createImplicitRelationships`

This processor adds Relationships to the Order if there is only one `ShippingGroup` or one `PaymentGroup`. If either one of these or both have no Relationships, then relationships will automatically be created. For the `ShippingGroup`, Relationships will be created between it and each `CommerceItem`. For the `PaymentGroup`, a Relationship will be created between itself and the Order with type `OrderAmountRemaining`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/CreateImplicitRelationships`

PipelineProcessor object: `atg.commerce.order.processor.ProcCreateImplicitRelationships`

Transitions: return value of 1 will execute `setPaymentGroupAmount` next

`setPaymentGroupAmount`

This processor sets the amount property of each `PaymentGroup` in the Order based on the Relationships in each `PaymentGroup`. This amount is the amount that will ultimately be debited by the `PaymentManager`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SetPaymentGroupAmount`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetPaymentGroupAmount`

Transitions: return value of 1 will execute `moveUsedPromotions` next

moveUsedPromotions

This processor updates the promotion use information in the Profile repository for the user.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/MoveUsedPromotions`

PipelineProcessor object: `atg.commerce.order.processor.ProcMoveUsedPromotions`

Transitions: return value of 1 will execute `removeUnusedPromotions` next

removeUnusedPromotions

This processor removes unused promotions if configured as explained in (xref).

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/RemoveUnusedPromotions`

PipelineProcessor object: `atg.commerce.order.processor.ProcRemoveUnusedPromotions`

Transitions: return value of 1 will execute `authorizePayment` next

authorizePayment

This processor authorizes all the payment information in the `PaymentGroups`. It essentially calls the `authorize()` method in the `PaymentManager` for each `PaymentGroup`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/AuthorizePayment`

PipelineProcessor object: `atg.commerce.order.processor.ProcAuthorizePayment`

Transitions: return value of 1 will execute `updateGiftRepository` next

updateGiftRepository

This processor updates the gift list repository information for the user.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/UpdateGiftRepository`

PipelineProcessor object: `atg.commerce.order.processor.ProcUpdateGiftRepository`

Transitions: return value of 1 will execute `sendGiftPurchasedMessage` next

sendGiftPurchasedMessage

This processor sends a gift purchased message to the messaging system.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SendGiftPurchasedMessage`

PipelineProcessor object: `atg.commerce.order.processor.ProcSendGiftPurchasedMessage`

Transitions: return value of 1 will execute `setSubmittedSiteId` next

`setSubmittedSiteId`

If you are using multisite, this processor sets the ID of the site on which the user created the order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processing/SetSumittedSite`

PipelineProcessor object: `atg.commerce.order.processor.SetSubmittedSite`

Transitions: return value of 1 executes `addOrderToRepository` next.

`addOrderToRepository`

This processor saves the Order to the Order Repository and if the user is not a registered user, adds the Order to the repository and then saves it.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/AddOrderToRepository`

PipelineProcessor object: `atg.commerce.order.processor.ProcAddOrderToRepository`

Transitions: return value of 1 will execute `sendPromotionUsedMessage` next

`sendPromotionUsedMessage`

This processor sends a message to the Scenario Server for each promotion that was used in the Order signifying that the given promotion was used by the user.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SendPromotionUsedMessage`

PipelineProcessor object: `atg.commerce.order.processor.ProcSendPromotionUsedMessage`

Transitions: return value of 1 will execute `sendFulfillmentMessage` next

`sendFulfillmentMessage`

This processor sends a message to the fulfillment engine signifying that it should begin processing the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SendFulfillmentMessage`

PipelineProcessor object: `atg.commerce.order.processor.ProcSendFulfillmentMessage`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateForCheckout Pipeline Chain

The `validateForCheckout` pipeline chain verifies that the Order is ready for checkout. The `validateForCheckout` pipeline chain is executed by the `validateOrder()` method in the `OrderManager` and the `processOrder` pipeline chain. The `validateOrder()` method adds the given `Order`, `Locale`, and `OrderManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

validateOrderForCheckout

This processor validates that there is at least one ShippingGroup, one PaymentGroup, and one CommerceItem in the Order before checking out.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/ValidateOrderForCheckout

PipelineProcessor object: atg.commerce.order.processor.ProcValidateOrderForCheckout

Transitions: return value of 1 will execute verifyOrderAddresses next

verifyOrderAddresses

This processor verifies the given addresses in the HardgoodShippingGroup and CreditCard objects. It does this by calling the verifyAddress() method in the AddressVerificationProcessor, which is configured in the verifyOrderAddresses processor.

Note: The AddressVerificationProcessor is not a pipeline processor.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/VerifyOrderAddresses

PipelineProcessor object: atg.commerce.order.processor.ProcVerifyOrderAddresses

Transitions: return value of 1 will execute validateShippingGroupsForCheckout next

validateShippingGroupsForCheckout

This processor validates ShippingGroups before checking an Order out. It checks that all CommerceItems in the Order are assigned to ShippingGroups and that all the required fields in all the ShippingGroups, regardless of type, are not null or empty String.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/ValidateShippingGroupsForCheckout

PipelineProcessor:object: atg.commerce.order.processor.ProcValidateShippingGroupsForCheckout

Transitions: return value of 1 will execute creditCardModCheck next

creditCardModCheck

This processor does a mod check on credit card numbers to see if they are valid. The verifyCreditCard method of the atg.payment.creditcard.ExtendableCreditCardTools class is called on each credit card number in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/CreditCardModCheck

PipelineProcessor object: atg.commerce.order.processor.ProcCreditCardModCheck

Transitions: return value of 1 will execute validatePaymentGroupsForCheckout next

validatePaymentGroupsForCheckout

This processor validates PaymentGroups before checking an Order out. It checks that all CommerceItems, shipping costs, and tax in the Order are assigned to PaymentGroups. It also checks that all the required fields in all the PaymentGroups, regardless of type, are not null or an empty String.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/ValidatePaymentGroupsForCheckout

PipelineProcessor object: atg.commerce.order.processor.ProcValidatePaymentGroupsForCheckout

Transitions: return value of 1 will execute validateCostCentersForCheckout next

validateCostCentersForCheckout

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/ValidateCostCentersForCheckout

PipelineProcessor object: atg.commerce.order.processor.ProcCostCentersForCheckout

Transitions: return value of 1 will execute validateShippingCostsForCheckout next

validateShippingCostsForCheckout

This processor validates that all shipping costs are accounted for by a PaymentGroup. Shipping costs are accounted for if there is only one PaymentGroup and it has no Relationships, if the ShippingGroup has been assigned to a PaymentGroup, or if an order level Relationship exists in the Order that covers the entire amount of the Order.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/ValidateShippingCostsForCheckout

PipelineProcessor object: atg.commerce.order.processor.ProcValidateShippingCostsForCheckout

Transitions: return value of 1 will execute validateOrderCostsForCheckout next.

validateOrderCostsForCheckout

This processor validates that all order costs are accounted for by a PaymentGroup. Order costs are accounted for if there is only one PaymentGroup and it has no Relationships or if order level Relationships exist in the Order that cover the entire amount of the Order.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/order/processor/ValidateOrderCostsForCheckout

PipelineProcessor object: atg.commerce.order.processor.ProcValidateOrderCostsForCheckout

Transitions: return value of 1 will execute validateHandlingInstructionsForCheckout next

validateHandlingInstructionsForCheckout

This processor validates that the total quantities in the HandlingInstructions do not exceed the amount assigned to the ShippingGroup. It does this by iterating over all the HandlingInstructions in the ShippingGroups and validating that the sum of the quantities in the HandlingInstructions do not exceed

that which is assigned to the `ShippingGroup`. It will also catch errors if `HandlingInstructions` contain errors such as invalid `ShippingGroup` and `CommerceItem` IDs or `CommerceItems` that are not assigned to the `ShippingGroup` that contains the `HandlingInstruction`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateHandlingInstructionsForCheckout`

PipelineProcessor object:
`atg.commerce.order.processor.ProcValidateHandlingInstructionsForCheckout`

Transitions: return value of 1 will execute `validateCurrencyCodes` next

`validateCurrencyCodes`

Verifies that all the `PriceInfo` objects in the Order have been priced using the same currency code. The currency code in the `OrderPriceInfo` object is the one that must be matched. The code checks the `TaxPriceInfo` object's currency code in the Order and all the `ShippingPriceInfo` and `ItemPriceInfo` currency codes in all the `ShippingGroups` and `CommerceItems`, respectively.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateCurrencyCodes`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateCurrencyCodes`

Transitions: Return value of 1 executes `checkForDiscontinuedProducts` next.

`checkForDiscontinuedProducts`

Ensures that the order does not contain any products that are no longer available.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/CheckForDiscontinuedProducts`

PipelineProcessor object: `atg.commerce.order.processor.ProcCheckForDiscontinuedProducts`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validatePostApproval Pipeline Chain

The `validatePostApproval` pipeline chain revalidates an order after an order requiring approval has been approved, or after the system determines that an order does not require approval. It only needs to check information that the approver is expected to specify, or that the approver might change as part of the order approval process.

By default it revalidates all payment-related information, as well as checking that all order and shipping costs are accounted for. It assumes that other information about the order is unchanged, and does not revalidate shipping addresses, etc. If the application's approval process allows approvers to change other order information, that information should also be revalidated here.

The following sections describe each processor in the pipeline chain.

`valid`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidatePaymentGroupsPostApproval`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidatePaymentGroupsForCheckout`

Transitions: Return value of 1 executes the `validatePaymentGroupsPostApproval` pipeline chain.

`validateCostCentersPostApproval`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateCostCentersForCheckout`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateCostCentersForCheckout`

Transitions: Return value of 1 executes the next link in the pipeline chain.

`validateShippingCostsPostApproval`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateShippingCostsForCheckout`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateShippingCostsForCheckout`

Transitions: Return value of 1 executes the next link in the pipeline chain.

`validateOrderCostsPostApproval`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateOrderCostsForCheckout`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateOrderCostsForCheckout`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validatePaymentGroupsPostApproval Pipeline Chain

This chain is called by `validatePaymentGroupsPostApproval` to validate each payment group in turn. This configuration is in part from the base Oracle ATG Web Commerce configuration for the `validatePaymentGroup` chain, but also includes validation for the `invoiceRequest` payment method. If you add new payment methods to the `validatePaymentGroup`, add them here as well so they are revalidated after order approval.

The following section describes the processor in the pipeline chain.

`validatePaymentGroupsPostApproval`

This processor uses `dispatchOnPGTypePostApproval` to determine which payment group types are included in the Order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidatePaymentGroupByType`

PipelineProcessor object: `atg.commerce.order.processor.ProcDispatchOnProperty`

Transitions: A return value of 4000 results in execution of `validateCreditCardPGPostApproval`. A return value of 4001 results in execution of `validateGiftCertificatePGPostApproval`. A return value of 4002 results in execution of `validateStoreCreditPGPostApproval`. A return value of 5000 results in execution of `validateInvoiceRequestPGPostApproval`.

validateNoApproval Pipeline Chain

This chain validates orders that do not require approval. By default, Oracle ATG Web Commerce defers validating the `invoiceRequest` payment method until it is known whether the order requires approval, so it is validated in this chain.

The following section describes the processor in the pipeline chain.

`validatePaymentGroupsNoApproval`

This processor validates payment groups not validated by the `validatePaymentGroupsForCheckout` chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidatePaymentGroupsNoApproval`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidatePaymentGroupsForCheckout`

Transitions: None, this is the only link in the chain and will cause the `PipelineManager` to return to the caller.

validatePaymentGroupNoApproval Pipeline Chain

This chain is called by ValidatePaymentGroupsNoApproval to validate payment groups that were skipped in the validatePaymentGroupsForCheckout chain.

dispatchOnPGTypeNoApproval

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidatePaymentGroupsNoApproval`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidatePaymentGroupsForCheckout`

Transitions: A return value of 4000 results in execution of validateCreditCardNoApproval. A return value of 4001 results in execution of validateGiftCertificateNoApproval. A return value of 4002 results in execution of validateStoreCreditNoApproval. A return value of 5000 results in execution of validateInvoiceRequestNoApproval.

validatePaymentGroup Pipeline Chain

This chain validates one payment group. The following sections describe each processor in the pipeline chain.

dispatchOnPGType

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidatePaymentGroupByType`

PipelineProcessor object: `atg.commerce.order.processor.ProcDispatchOnProperty`

Transitions: If the processor returns a value of 4000, execute the `validateCreditCardPG` pipeline processor. If the processor returns 4001, execute the `validateGiftCertificatePG` pipeline processor.If the processor returns 4002, execute the `validateStoreCreditPG` processor. If the processor returns 5000, execute the `validateInvoiceRequestPG` processor.

validateCreditCardPG

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateCreditCard`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateCreditCard`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateGiftCertificatePG

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateGiftCertificate`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateGiftCertificate`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateStoreCreditPG

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateStoreCredit`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateStoreCredit`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateInvoiceRequestPG

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/EmptyProcessor`

PipelineProcessor object: `atg.commerce.order.processor.ValidateInvoiceRequest`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

recalcPaymentGroupAmounts Pipeline Chain

The `recalcPaymentGroupAmounts` pipeline chain regenerates the amount that must be assigned to each `PaymentGroup` in the Order. The `recalcPaymentGroupAmounts` pipeline chain is executed by the `recalculatePaymentGroupAmounts()` method in the `OrderManager`. The `recalculatePaymentGroupAmounts()` method adds the given Order and `OrderManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following section describes the processor in the pipeline chain.

setPaymentGroupAmount2

This processor sets the `amount` property of each `PaymentGroup` in the Order based on the Relationships in each `PaymentGroup`. This amount is the amount that will ultimately be debited by the `PaymentManager`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SetPaymentGroupAmount`

PipelineProcessor object: `atg.commerce.order.processor.ProcSetPaymentGroupAmount`

Transitions: None, this is the only link in the chain and will cause the `PipelineManager` to return to the caller.

repriceOrder Pipeline Chain

The `repriceOrder` pipeline chain prices the Order. The `repriceOrder` pipeline chain is executed by the `handleRepriceOrder()` method in the `CartModifierFormHandler` and the `ExpressCheckoutFormHandler`. The pipeline chain's transaction mode is TX_REQUIRED.

The following section describes the processor in the pipeline chain.

priceOrderTotal

This processor causes the Order to be re-priced using the pricing engine.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/pricing/processor/PriceOrderTotal`

PipelineProcessor object: `atg.commerce.pricing.processor.PriceOrderTotal`

Transitions: None, this is the only link in the chain and will cause the `PipelineManager` to return to the caller.

repriceAndUpdateOrder Pipeline Chain

The following sections describe each processor in the pipeline chain.

executeRepriceOrderChainForUpdate

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/pricing/processor/ExecuteRepriceOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: If returns one, executes the updateOrderAfterReprice processor.

updateOrderAfterReprice

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/UpdateOrder`

PipelineProcessor object: `atg.commerce.order.processor.ProcUpdateOrder`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

moveToConfirmation Pipeline Chain

The `moveToConfirmation` pipeline chain prices the Order and validates it. The `moveToConfirmation` pipeline chain is executed by the `handleMoveToConfirmation()` method in the `PaymentGroupFormHandler`. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`executeRepriceOrderChain`

This processor causes the `repriceOrder` pipeline chain to be executed.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/pricing/processor/ExecuteRepriceOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: return value of 1 will execute `executeValidateForCheckoutChain2` next.

`executeValidateForCheckoutChain2`

This processor causes the `validateForCheckout` chain to be executed.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ExecuteValidateForCheckoutChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validatePaymentGroupPreConfirmation Pipeline Chain

The following sections describe each processor in the pipeline chain.

dispatchOnPGTypePreConfirmation

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidatePaymentGroupByType`

PipelineProcessor object: `atg.commerce.order.processor.ProcDispatchOnProperty`

Transitions: If the processor returns a value of 4000, execute the validateCreditCardPGPreConfirmation pipeline processor. If the processor returns 4001, execute the validateGiftCertificatePGPreConfirmation pipeline processor.If the processor returns 4002, execute the validateStoreCreditPGPreConfirmation processor.

validateCreditCardPGPreConfirmation

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateCreditCard`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateCreditCard`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateGiftCertificatePGPreConfirmation

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateGiftCertificate`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateGiftCertificate`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateStoreCreditPGPreConfirmation

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateStoreCredit`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateStoreCredit`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

moveToPurchaseInfo Pipeline Chain

The `moveToPurchaseInfo` pipeline chain validates the Order. The `moveToPurchaseInfo` pipeline chain is executed by the `handleMoveToPurchaseInfo()` method in the `CartModifierFormHandler`. The pipeline chain's transaction mode is TX_REQUIRED.

The following section describes the processor in the pipeline chain.

validateOrderForCheckout2

This processor causes the `validateForCheckout` chain to be executed.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateOrderForCheckout`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateOrderForCheckout`

Transitions: None, this is the only link in the chain and will cause the `PipelineManager` to return to the caller.

validateShippingInfo Pipeline Chain

The `validateShippingInfo` pipeline chain validates the `ShippingGroups` in the Order. The `validateShippingInfo` pipeline chain is executed by the `validateShippingGroupsChainId()` method in the `ShippingGroupFormHandler`. The pipeline chain's transaction mode is TX_REQUIRED.

The following section describes the processor in the pipeline chain.

validateShippingGroupsInfo

This processor validates `ShippingGroups` before checking an Order out. This processor checks that all `CommerceItems` in the Order are assigned to `ShippingGroups` and that all the required fields in all the `ShippingGroups`, regardless of type, are not null or empty String.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateShippingGroupsForCheckout`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateShippingGroupsFor Checkout`

Transitions: None, this is the only link in the chain and will cause the `PipelineManager` to return to the caller.

validateShippingGroup Pipeline Chain

The following sections describe each processor in the pipeline chain.

dispatchOnSGType

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateShippingGroupByType`

PipelineProcessor object: `atg.commerce.order.processor.ProcDispatchOnProperty`

Transitions: If the processor returns 4000, call the validateHardGoodSG processor. If it returns 4001, call the ValidateElectronicSG processor.

validateHardgoodSG

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateHardgoodShippingGroup`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateHardgoodShippingGroup`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

validateElectronicSG

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/ValidateElectronicShippingGroup`

PipelineProcessor object: `atg.commerce.order.processor.ProcValidateElectronicShippingGroup`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

sendScenarioEvent Pipeline Chain

The `sendScenarioEvent` pipeline chain sends a message to the Dynamo Message System. The `sendScenarioEvent` pipeline chain is used in various areas of Oracle ATG Web Commerce. The pipeline chain's transaction mode is TX_REQUIRED.

The following section describes the processor in the pipeline chain.

`sendScenarioEvent`

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SendScenarioEvent`

PipelineProcessor object: `atg.commerce.order.processor.ProcSendScenarioEvent`

Transitions: None, this is the only link in the chain and will cause the `PipelineManager` to return to the caller.

Notes: This processor sends scenario action events to the scenario server.

processScheduledOrder Pipeline Chain

This chain places a scheduled order and then sends an event. The following sections describe each processor in the pipeline chain.

runProcessOrderPipeline

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/RunProcessOrderChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: If the processor returns 1, call the sendMessageScheduledOrderMessage processor.

sendMessageScheduledOrderMessage

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/SendScheduledOrderMessage`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScheduledOrderMessage`

Transitions: None, this is the last link in the chain and will cause the `PipelineManager` to return to the caller.

Fulfillment Pipelines

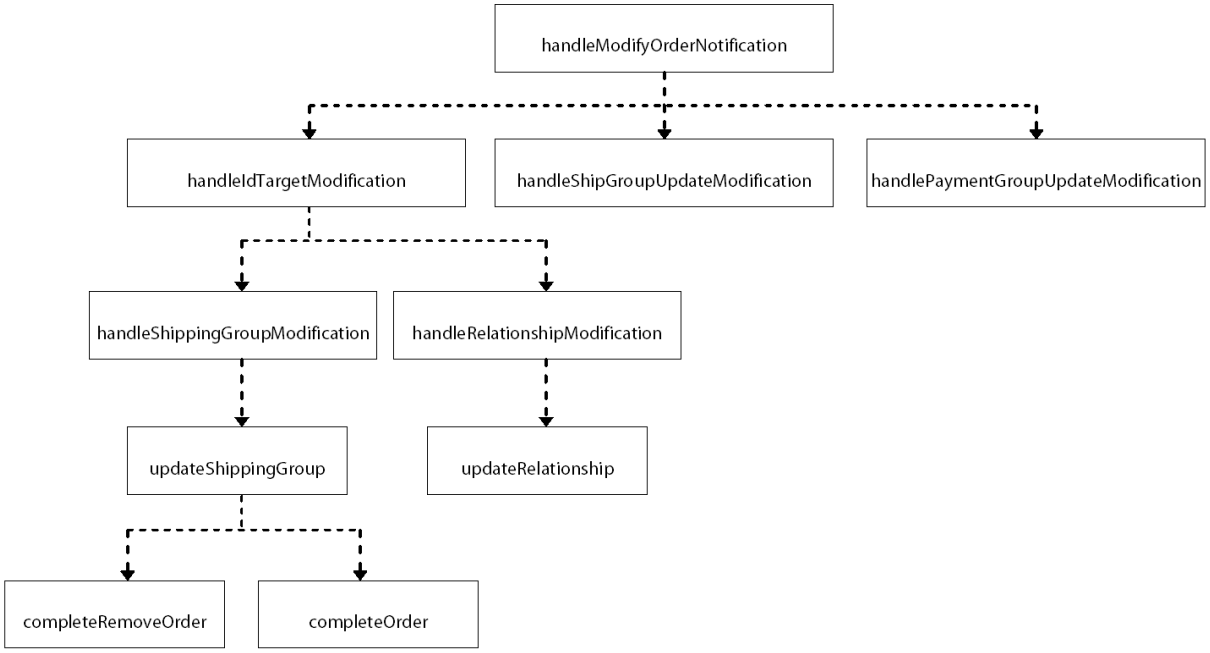
The following section includes diagrams that describe how processor chains work together in the fulfillment system, as specified in the `<ATG10dir>/Fulfillment/atg/commerce/fulfillment/fulfillmentpipeline.xml` configuration file.

The `handleSubmitOrder` chain is triggered when the `OrderFulfiller` receives a `SubmitOrder` message. It branches to either `splitShippingGroupsFulfillment` or `executeFulfillOrderFragment`.

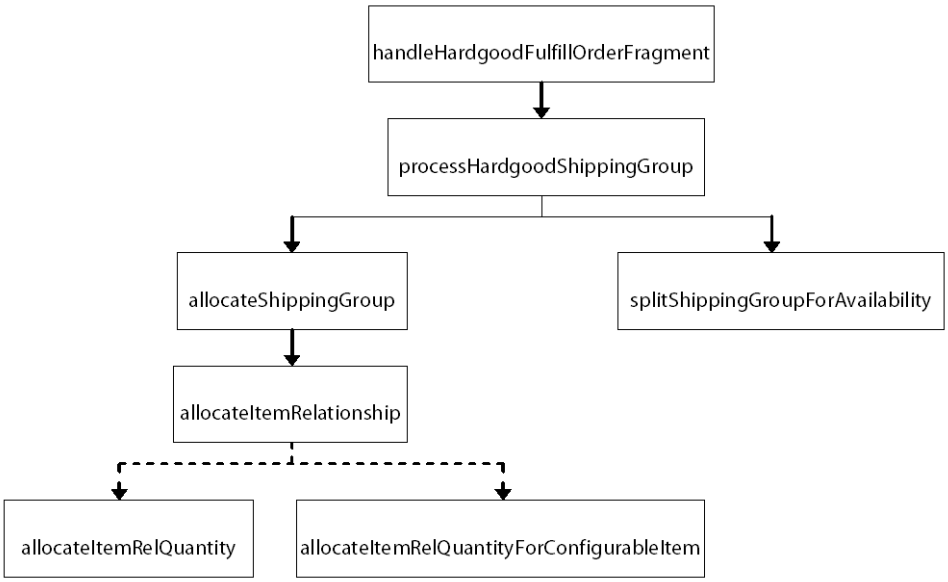
This series of chains is triggered when the `OrderFulfiller` receives a `ModifyOrder` message:

1. `handleModifyOrder`
2. `performIdTargetModification`
3. `performOrderModification`
4. `removeOrder`

The following series of chains is triggered by `OrderFulfiller` receiving a `ModifyOrderNotification` message.



The following series of chains is triggered when the `HardgoodFulfiller` receives a `FulfillOrderFragment` message.

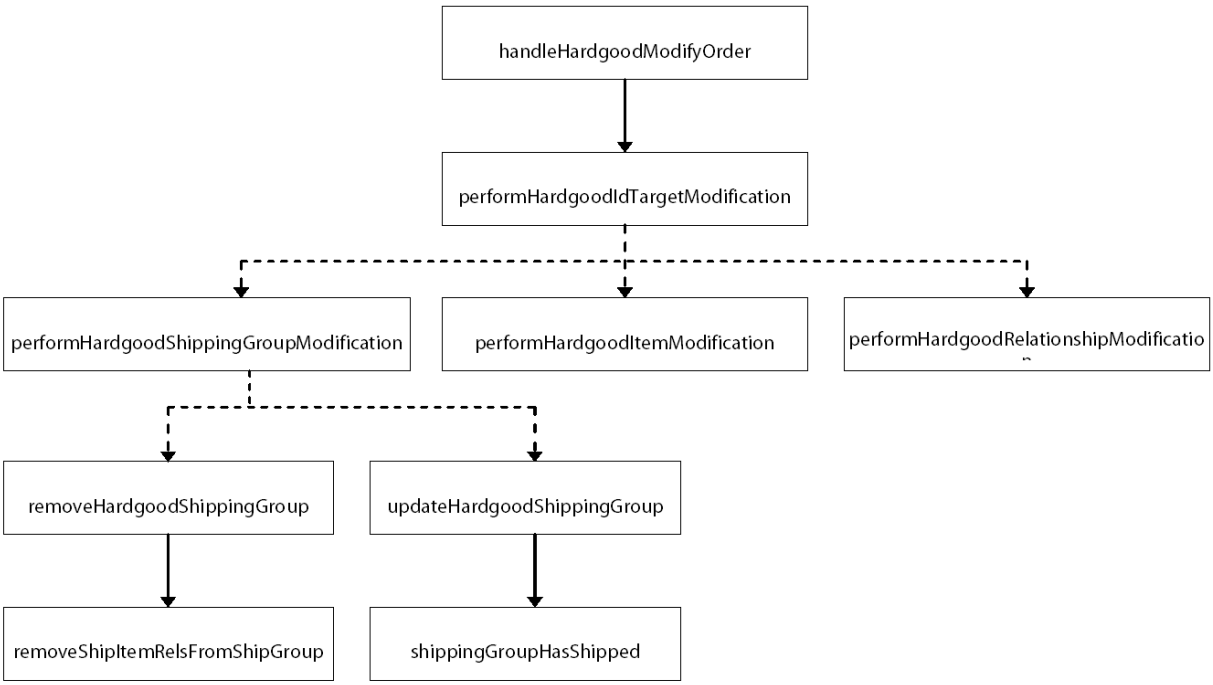


This series of chains is triggered when `HardgoodFulfiller` receives an `UpdateInventory` message:

1. `handleHardgoodUpdateInventory`
2. `handleOrderWaitingShipMap`

3. `processHardgoodShippingGroup`

The following series of chains is triggered when `HardgoodFulfiller` receives a `ModifyOrder` message.



This series of chains is triggered when `HardgoodFulfiller` receives a `ModifyOrderNotification` message:

1. `handleHardgoodModifyOrderNotification`
2. `handleHardgoodShipGroupUpdateModification`
3. `processHardgoodShippingGroup`

By default, the following series of chains is not triggered by the fulfillment system. This series of chains is provided as a resource for users extending the fulfillment system:

1. `shipPendingShippingGroups`
2. `shipShippingGroup`
3. `shippingGroupHasShipped`

The following series of chains is triggered when an `ElectronicFulfiller` receives a `FulfillOrderFragment` message:

1. `handleElectronicFulfillOrderFragment`
2. `prcoessElectronicShippingGroup`
3. `allocateElectronicGood`

By default, the following series of chains is not triggered by the fulfillment system. This series of chains is provided as a resource for users extending the fulfillment system:

1. `processElectronicShippingGroups`

2. `processElectronicShippingGroup`

3. `allocateElectronicGood`

handleSubmitOrder Pipeline Chain

The `handleSubmitOrder` chain is triggered when `OrderFulfiller` receives a `SubmitOrder` message. The purpose of this chain is to load the order, verify that the order should be fulfilled, divide it up among appropriate fulfillers, and deliver the necessary information to each fulfiller. This chain is triggered when `OrderFulfiller` receives a `SubmitOrder` message.

The following sections describe each processor in the pipeline chain.

`extractOrderId`

Attempts to extract the ID of the order from the `OrderId` property of the `SubmitOrder` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `handleRetrieveOrder` processor.

`handleRetrieveOrder`

Determines the method by which the Order should be loaded. If the order ID was successfully extracted in the `extractOrderId` processor, then moves to the `loadOrder` processor.

If the order ID was not extracted successfully, then check the `LookUpOrderIdFromOrder` property of the `OrderFulfiller`. If this property is true, the chain moves to the `loadSaveOrder` processor. If this property is false, then the processor throws an `InvalidParameterException` and chain execution stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleRetrieveOrder`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleRetrievingOrder`

Transitions: Return value of 1 executes the `loadSaveOrder` processor. Return value of 2 executes the `loadOrder` processor.

`loadSaveOrder`

Checks to see if the Order exists in the order repository, using the `OrderExists` method of `OrderManager` and the ID of the serialized order within the `SubmitOrder` message as the parameter. If the order exists, the processor loads the order. If the order does not exist, then fulfillment is using a different repository than the order placement system. The processor then saves the order from the message into the repository. In either case, the chain then moves to the `verifyOrderForFulfillment` processor.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadSaveOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadSaveOrderRepository`

Transitions: Return value of 1 executes the `verifyOrderForFulfllment` processor.

`loadOrder`

Loads the order from the order repository. Control then passes to `verifyOrderForFulfillement`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `verifyOrderForFulfllment` processor.

`verifyOrderForFulfillment`

Calls the `verifyOrderFulfillment` method of `OrderFulfillmentTools`, which checks to make sure the order is in a valid state for fulfillment: not INCOMPLETE, PENDING_REMOVE, REMOVED, or NO_PENDING_ACTION.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/VerifyOrderForFulfillment`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcVerifyOrderForFulfillment`

Transitions: Return value of 1 executes the splitShippingGroupsFulfillmentChain processor.

`splitShippingGroupsFulfillmentChain`

Runs `splitShippingGroupsFulfilllment` chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SplitShippingGroupsFulfillmentChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes the `executeFulfillOrderFragmentChain` processor.

`executeFulfillOrderFragmentChain`

Iterates through the shipping groups, and runs `executeFulfillOrderFragment` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExecuteFulfillOrderFragmentChain`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExecuteFulfillOrderFragment`

Transitions: Return value of 1 executes the `updateOrderRepository` processor.

`updateOrderRepository`

Updates the order in the repository with any changes that may have been made during the execution of this chain (splitting of shipping groups, update of states, etc.).

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification` processor.

`sendModifyOrderNotification`

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcSendModifyOrderNotification`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

splitShippingGroupsFulfillment Pipeline Chain

The `splitShippingGroupsFulfillment` chain splits shipping groups according to the fulfillment systems that handle each of the particular items within each shipping group.

The following sections describe each processor in the pipeline chain.

`retrieveShippingGroupsToBeSplit`

Iterates through the shipping groups contained within the order and determines if the entire shipping group can be fulfilled by one fulfiller. It does this using the `isShippingGroupSingleFulfiller` method of the `OrderFulfillmentTool`. Those shipping groups that cannot be fulfilled by one fulfiller are placed as an `ArrayList` in the pipeline's `pParam` map parameter, with the key being the pipeline's SHIPPINGGROUPIDS constant.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/RetrieveShippingGroupsToBeSplit`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcRetrieveShippingGroupsToBeSplit`

Transitions: Return value of 1 executes the splitShippingGroupsForFulfillment processor.

`splitShippingGroupsForFulfillment`

Sends the `ArrayList` generated in the previous processor to `splitShippingGroupsByFulfiller` method of `OrderFulfillmentTools`, which does the actual splitting of the shipping groups, keeping track of the changes through a Modification list.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SplitShippingGroupsForFulfillment`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcSplitShippingGroupsForFulfillment`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

executeFulfillOrderFragment Pipeline Chain

The `executeFulfillOrderFragment` chain verifies that each shipping group is in a state that is ready for fulfillment, and sends `FulfillOrderFragment` messages out to the appropriate fulfillers.

The following sections describe each processor in the pipeline chain.

`verifyShippingGroupsForFulfillers`

Attempts to verify that the shipping groups can be fulfilled by the default fulfiller using the `verifyShippingGroupsForFulfiller` method of `OrderFulfillmentTools`. The state of the shipping group is set to PENDING_MERCHANT_ACTION if either of the following is true:

- The fulfiller does not appear in the `FulfillerShippingGroupHashMap` of `OrderFufillmentTools`.
- The shipping group is not of the appropriate class for that fulfiller.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/VerifyShippingGroupsForFulfillers`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcVerifyShippingGroupsForFulfillers`

Transitions: Return value of 1 executes the sendFulfillOrderFragment processor.

`sendFulfillOrderFragment`

Sets the order's and all the shipping groups' states to PROCESSING, then builds a `FulfillOrderFragment` message and sends it using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendFulfillOrderFragment`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcSendFulfillOrderFragment`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleModifyOrder Pipeline Chain

The `handleModifyOrder` chain is triggered when a fulfiller receives a `ModifyOrder` message. Determines whether the modification is valid, performs it if it is valid, and sends out a `ModifyOrderNotification` message to inform other systems of the changes that were made, or that the changes requested were invalid.

The following sections describe each processor in the pipeline chain.

extractOrderId1

This processor attempts to extract the ID of the order from the `OrderId` property of the `ModifyOrder` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `loadOrder1processor`.

`loadOrder1`

This processor loads the order from the order repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `handleModificationClassType` processor.

`handleModificationClassType`

Determines if the modifications listed in the `ModifyOrder` message are valid. If so, it calls the appropriate processor chains, and upon conclusion, passes control to the updateOrderRepository1 processor. The only chain that this processor will trigger is `performIdTargetModification`. If a modification listed is not valid, then the chain moves on to the `modificationNotSupported` processor.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationClassType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationClassType`

Transitions: Return value of 1 executes the `updateOrderRepository1` processor. Return value of 2 executes the `modificationNotSupported` processor.

`modificationNotSupported`

Sets the status of the particular modification to STATUS_FAILED and adds the modification to the list to be sent back in a `ModifyOrderNotification` message. Control then passes to updateOrderRepository1.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: Return value of 1 executes the updateOrderRepository1processor.

updateOrderRepository1

Updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification1`processor.

sendModifyOrderNotification1

Sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcSendModifyOrderNotification`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

performIdTargetModification Pipeline Chain

The `performIdTargetModification` chain is triggered by a `ModifyOrder` message that includes a modification of type `IdTargetModification`.

The following sections describe each processor in the pipeline chain.

`handleModificationTargetType`

This processor determines which processor to pass control to by looking at the `TargetType` property of the `IdTargetModification`. If the `TargetType` is TARGET_ORDER, then control passes to `performOrderModificationChain`. If the `TargetType` is TARGET_SHIPPING_GROUP, control passes to `performShippingGroupModification`. If the `TargetType` is TARGET_ITEM, control passes to `performItemModification`. If `TargetType` is TARGET_RELATIONSHIP, control passes to `performRelationshipModification`. If `TargetType` is none of the above types, control passes to `modificationNotSupported1`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationTargetType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the performOrderModificationChain processor. Return value of 2 executees the performShippingGroupModification processor. Return value of 3 executes the performItemModification. Return value of 4 executes the performRelationshipModification processor. Return value of 5 executes the `modificationNotSupported1` processor.

`performOrderModificationChain`

This processor executes the `performOrderModification` chain. After execution, the execution of this chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/PerformOrderModificationChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`performShippingGroupModification`

Determines the appropriate fulfiller for the shipping group of the modification, and sends a `ModifyOrder` message to that fulfiller. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/PerformShippingGroupModification`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcPerformShippingGroupModification`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`performItemModification`

Item modifications are not currently supported, so this processor sets the status of the particular modification to STATUS_FAILED and adds the modification to the list to be sent back in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`performRelationshipModification`

Determines the appropriate fulfiller for the shipping group involved in the `ShippingGroupCommerceItem` relationship the modification is requested for, and sends a `ModifyOrder` message to that fulfiller. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/PerformRelationshipModification`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcPerformRelationshipModification`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`modificationNotSupported1`

Sets the status of the particular modification to STATUS_FAILED and adds the modification to the list to be sent back in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

performOrderModification Pipeline Chain

The `performOrderModification` chain is triggered when called by the `performOrderModifcationChain` processor of the `performIdTargetModification` chain.

The following sections describe each processor in the pipeline chain.

`handleModificationType`

This processor determines the type of modification requested by looking at the `ModificationType` property of the modification. If the `ModificationType` is ADD_MODIFICATION, control passes to `addOrder`.

If the `ModificationType` is REMOVE_MODIFICATION, control passes to `removeOrderChain`. If the `ModificationType` is neither of these, control passes to `updateOrder`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationType`

Transitions: Return value of 1 executes the `addOrder` processor. Return value of 2 executes the `removeOrderChain` processor. Return value of 3 executes the `updateOrder` processor.

`addOrder`

Modifications that add orders are currently not supported, so this processor sets the status of the particular modification to STATUS_FAILED and adds the modification to the list to be sent back in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`removeOrderChain`

Executes the `removeOrder` pipeline. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/RemoveOrderChain`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcRemoveOrder`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`updateOrder`

Modifications that update orders are currently not supported, so this processor sets the status of the particular modification to STATUS_FAILED and adds the modification to the list to be sent back in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

removeOrder Pipeline Chain

The `removeOrder` chain is triggered by the `removeOrderChain` processor of the `performOrderModification` chain.

The following sections describe each processor in the pipeline chain.

`verifyOrderForRemoval`

Verifies that the order is in an appropriate state for removal, that none of the shipping groups have been shipped, and that either none of the shipping groups are in a state of PENDING_SHIPMENT, or that the fulfiller's `AllowRemoveOrderWithPendingShipment` property is true. If any of those conditions are not met, the chain stops execution. If all those conditions are met, the chain continues.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/VerifyOrderForRemoval`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcVerifyOrderForRemoval`

Transitions: Return value of 1 executes the `sendModifyOrderForRemoval` processor.

`sendModifyOrderForRemoval`

This processor iterates through each fulfiller, and then each shipping group within each fulfiller. It sets the state of the shipping group to REMOVED, and adds the modification to a modification list. After all the fulfillers have been processed, it sends out a `ModifyOrderNotification` message containing the modifications.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderForRemoval`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcSendModifyOrderForRemoval`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleModifyOrderNotification Pipeline Chain

The `handleModifyOrderNotification` chain is triggered by `OrderFulfiller` receiving a `ModifyOrderNotification` message.

The following sections describe each processor in the pipeline chain.

`extractOrderId2`

This processor attempts to extract the ID of the order from the `OrderId` property of the `ModifyOrderNotification` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `loadOrder2` processor.

`loadOrder2`

This processor loads the order from the order repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `handleModificationClassType1` processor.

`handleModificationClassType1`

Determines if the modifications listed in the `ModifyOrderNotification` message are valid. If so, it calls the appropriate processor chains, and upon conclusion, passes control to the `updateOrderRepository2` processor. Possible chains that `ModifyOrderNotification` modifications could trigger are `handleIdTargetModification`, `handleShipGroupUpdateModification`, and `handlePayGroupUpdateModification`.

If a modification listed is not valid, then the chain moves on to the `modificationNotSupported3` processor.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationClassType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationClassType`

Transitions: Return value of 1 executes the `updateOrderRepository2` processor. Return value of 2 executes the `modificationNotSupported3` processor.

`modificationNotSupported3`

This processor currently does nothing. Control then passes to `updateOrderRepository1`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: Return value of 1 executes the `updateOrderRepository2` processor.

`updateOrderRepository2`

Updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification2` processor.

`sendModifyOrderNotification2`

If any modifications were made during the execution of this chain, this processor sends a `ModifyOrderNotification` message with the list of modifications using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcSendModifyOrderNotification`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleIdTargetModification Pipeline Chain

The `handleIdTargetModification` chain is executed when called by the `handleModifyOrderNotification` chain.

The following sections describe each processor in the pipeline chain.

`handleModificationTargetType1`

This processor determines which processor to pass control to by looking at the `TargetType` property of the `IdTargetModification`. If the `TargetType` is TARGET_ORDER, then control passes to `handleOrderModificationChain`. If the `TargetType` is TARGET_SHIPPING_GROUP, control passes to `handleShippingGroupModificationChain`. If the `TargetType` is TARGET_ITEM, control passes to `handleItemModification`. If `TargetType` is TARGET_RELATIONSHIP, control passes to `handleRelationshipModificationChain`. If `TargetType` is none of the above types, control passes to `modificationNotSupported4`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationTargetType`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcHandleModificationTargetType`

Transitions: Return value of 1 executes the `handleOrderModification` processor. Return value of 2 executes the `handleShippingGroupModificationChain` processor. Return value of 3 executes the `handleItemModification` processor. Return value of 4 executes the `handleRelationshipModificationChain` processor. Return value of 5 executes the `modificationNotSupported4` processor.

`handleOrderModification`

This modification type is currently not supported. This processor simply logs an error. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/EmptyProcessor`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`handleShippingGroupModificationChain`

This processor executes the `handleShippingGroupModification` chain. After execution, the execution of this chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleShippingGroupModificationChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`handleItemModification`

This modification type is currently not supported. This processor simply logs an error. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/EmptyProcessor`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`handleRelationshipModificationChain`

This processor executes the `handleRelationshipModification` chain. After execution, the execution of this chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleRelationshipModificationChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

modificationNotSupported4

This processor logs an error. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleShipGroupUpdateModification Pipeline Chain

The `handleShipGroupUpdateModification` chain is executed when called by the `handleModifyOrderNotification` chain.

The following section describes the processor in the pipeline chain.

`shipGroupUpdateModification`

This processor iterates through each shipping group contained within the modification. Makes sure the state of the shipping group is INITIAL, then sets it to PROCESSING, resets the submitted date on the shipping group to the current time, and sends out a `ModifyOrderNotification` message detailing the changes made.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleShipGroupUpdateModification`

PipelineProcessor object:
`atg.commerce.fulfillment.processor.ProcHandleShippingGroupUpdateModification`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

handlePaymentGroupUpdateModification Pipeline Chain

The `handlePaymentGroupUpdateModification` chain is executed when called by the `handleModifyOrderNotification` chain.

The following section describes the processor in the pipeline chain.

`paymentGroupUpdateModification`

This type of modification is currently not supported. This processor simply logs an error.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/EmptyProcessor`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

handleShippingGroupModification Pipeline Chain

The `handleShippingGroupModification` chain is executed when called by the `handleIdTargetModification` chain.

The following sections describe each processor in the pipeline chain.

handleModificationType1

This processor determines the type of modification requested by looking at the `ModificationType` property of the modification. If the `ModificationType` is ADD_MODIFICATION, control passes to `addShippingGroup`. If the `ModificationType` is REMOVE_MODIFICATION, control passes to `cancelRemoveOrder`. If the `ModificationType` is neither of these, control passes to `updateShippingGroupChain`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationType`

Transitions: Return value of 1 executes the addShippingGroup processor. Return value of 2 executes the cancelRemoveOrder processor. Return value of 3 executes the updateShippingGroupChain processor.

`addShippingGroup`

This type of modification is currently not supported. This processor logs an error. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/EmptyProcessor`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`cancelRemoveOrder`

Cancels the remove order modification because a component of the order could not be removed. Sets the state of the order to PENDING_MERCHANT_ACTION. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/CancelRemoveOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`updateShippingGroupChain`

This processor executes the `updateShippingGroup` chain. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/UpdateShippingGroupChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

updateShippingGroup Pipeline Chain

The `updateShippingGroup` chain is executed when called by the `handleShippingGroupModification` chain.

The following sections describe each processor in the pipeline chain.

`handleShippingGroupState`

Checks the `newValue` property of the modification to determine what state the modification is requesting that the shipping group be set to. If the value is REMOVED, control passes to `completeRemoveOrderChain`. If the value is NO_PENDING_ACTION, control passes to `completeOrderChain`. If the value is PENDING_MERCHANT_ACTION, control passes to `failOrder`. If the value is anything else then execution of this chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleShippingGroupState

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the completeRemoveOrderChain processor. Return value of 2 executes the completeOrderChain processor. Return value of 3 executes the failOrder processor.

completeRemoveOrderChain

This processor executes the `completeRemoveOrder` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/CompleteRemoveOrderChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

completeOrderChain

This processor executes the `completeOrder` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/CompleteOrderChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

failOrder

Sets the state of the order to PENDING_MERCHANT_ACTION, and adds this modification to the list of modifications.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/FailOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

completeRemoveOrder Pipeline Chain

The `completeRemoveOrder` chain is executed when called by the `updateShippingGroup` chain.

The following sections describe each processor in the pipeline chain.

creditOrder

Iterates through the payment groups in the order, and checks to see if each state is SETTLED. If it is, it calls the credit method of the `PaymentGroupManager` with that payment group, then sets the status of the payment group to INITIAL. If the payment group is not SETTLED, it checks to see if the payment group represents a gift certificate, and if so, calls the `expireGiftCertificateAuthorization` method of the `PaymentGroupManager` to credit the gift certificate.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/CreditOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the finishRemoveOrder processor.

`finishRemoveOrder`

Iterates through all commerce items and payment groups contained in the order, and sets their states to REMOVED. Also sets the order's state to REMOVED.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/FinishRemoveOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

completeOrder Pipeline Chain

The `completeOrder` chain is executed when called by the `updateShippingGroup` chain.

The following sections describe each processor in the pipeline chain.

`settleOrder`

Checks to see if all shipping groups have shipped, or if one shipping group has shipped and the `SettleOnFirstShipment` property of the `OrderFulfiller` is true. If not, then chain execution stops. Otherwise, it iterates through the Order's payment groups and calls the `debit` method of the `PaymentGroupManager` on all of them. If the debit fails for a payment group, the payment group's state is set to SETTLE_FAILED, and the order's state is set to PENDING_MERCHANT_ACTION. If the debit succeeds, the payment group's state is set to SETTLED.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/SettleOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the finishOrder processor.

`finishOrder`

Sets the order's state to NO_PENDING_ACTION and adds the modification to the modification list.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/FinishOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleRelationshipModification Pipeline Chain

The `handleRelationshipModification` chain is executed when called by the `HandleIdTargetModification` chain.

The following sections describe each processor in the pipeline chain.

handleModificationType2

This processor determines the type of modification requested by looking at the `ModificationType` property of the modification. If the `ModificationType` is ADD_MODIFICATION, control passes to `addRelationship`. If the `ModificationType` is REMOVE_MODIFICATION, control passes to `removeRelationship`. If the `ModificationType` is neither of these, control passes to `updateRelationshipChain`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationType`

Transitions: Return value of 1 executes the `addRelationship` processor. Return value of 2 executes the `removeRelationship` processor. Return value of 3 executes the `updateRelationshipChain` processor.

`addRelationship`

This type of modification is currently not supported. This processor logs an error. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/EmptyProcessor

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`removeRelationship`

This type of modification is currently not supported. This processor logs an error. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/EmptyProcessor

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`updateRelationshipChain`

This processor executes the `updateRelationship` chain.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/UpdateRelationshipChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

updateRelationship Pipeline Chain

The `updateRelationship` chain is executed when called by the `handleRelationshipModification` chain.

The following sections describe each processor in the pipeline chain.

handleRelationshipState

Checks to make sure the relationship exists, is a `ShippingGroupCommerceItem` relationship, that the shipping group's state was set to REMOVED, and that the modification was a success. If all these conditions are met, the chain moves to the next processor. Otherwise, chain execution stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleRelationshipState

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the removeShipItemRelationshipFromItem processor.

removeShipItemRelationshipFromItem

Deducts the quantity that was to ship in the given shipping group from the commerce item.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/RemoveShipItemRelationshipFromItem

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleHardgoodFulfillOrderFragment Pipeline Chain

The `handleHardgoodFulfillOrderFragment` chain is executed when `HardgoodFulfiller` receives a `FulfillOrderFragment` message.

The following sections describe each processor in the pipeline chain.

extractOrderId3

This processor attempts to extract the ID of the order from the `OrderId` property of the `FulfillOrderFragment` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `handleRetrieveOrder1` processor.

handleRetrieveOrder1

Determines the method by which the Order should be loaded. If the order ID was successfully extracted in the `extractOrderId` processor, then we move to the loadOrder3 processor. If not, then it checks the Boolean `LookUpOrderIdFromOrder` property of the `OrderFulfiller`. If true, we move to the loadSaveOrder1 processor. If false, then it throws an `InvalidParameterException`, and chain execution stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleRetrieveOrder`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleRetrievingOrder`

Transitions: Return value of 1 executes the `loadSaveOrder1` processor. Return value of 2 executes the `loadOrder3` processor.

`loadSaveOrder1`

Checks to see if the Order exists in the order repository, using the `OrderExists` method of `OrderManager`, and using the ID of the serialized order within the `FulfillOrderFragment` message as the parameter. If the order exists, the processor loads the order. If it does not, then fulfillment is using a different repository than the order placement system. The processor then saves the order from the message into the repository. In either case, the chain then moves to the `processHardgoodShippingGroupsChain` processor.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadSaveOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadSaveOrderRepository`

Transitions: Return value of 1 executes the `processHardgoodShippingGroupsChain` processor.

`loadOrder3`

This processor loads the order from the order repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `processHardgoodShippingGroupsChain` processor.

`processHardgoodShippingGroupsChain`

Iterates through the shipping groups contained in the `FulfillOrderFragment` message, and runs the `processHardgoodShippingGroup` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ProcessHardgoodShippingGroupsChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the `updateOrderRepository3` processor.

`updateOrderRepository3`

Updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification3` processor.

sendModifyOrderNotification3

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

processHardgoodShippingGroup Pipeline Chain

The `processHardgoodShippingGroup` chain is executed when called by the `handleHardgoodFulfillOrderFragment` chain or the `handleHardgoodUpdateShipGroupModification` chain.

The following sections describe each processor in the pipeline chain.

verifyShippingGroupForFulfillment

This processor checks to make sure the shipping group's state is PROCESSING. If not, it throws an exception and execution of the chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/VerifyShippingGroupForFulfillment

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the allocateShippingGroupChain processor.

allocateShippingGroupChain

This processor executes the `allocateShippingGroup` chain.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/AllocateShippingGroupChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the splitShippingGroupForAvailabilityChain processor.

splitShippingGroupForAvailabilityChain

This processor executes the `splitShippingGroupForAvailability` chain.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/SplitShippingGroupForAvailabilityChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

allocateShippingGroup Pipeline Chain

The `allocateShippingGroup` chain is executed when called by the `processHardgoodShippingGroup` chain.

The following section describes the processor in the pipeline chain.

`allocateItemRelationshipChain`

Iterates through the `ShippingGroupCommerceItem` relationships contained in the shipping group, and executes the `allocateItemRelationship` chain for that relationship.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/AllocateItemRelationshipChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

allocateItemRelationship Pipeline Chain

The `allocateItemRelationship` chain is executed when called by the `allocateShippingGroup` chain.

The following sections describe each processor in the pipeline chain.

`retrieveItemRelQuantity`

This processor gets the quantity of the commerce item in the relationship (or the remaining quantity if the relationship type is SHIPPINGQUANTITYREMAINING), and places it into the pipeline's parameter map.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ RetrieveItemRelQuantity

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the switchOnCommerceItemType processor.

`switchOnCommerceItemType`

This processor checks for the type of Commerce Item in the relationship. If it is a `ConfigurableCommerceItem`, control passes to `allocateItemRelQuantityForConfigurableItemChain`. Otherwise, control passes to `allocateItemRelQuantityChain`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/SwitchOnCommerceItemType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the allocateItemRelQuantityChain processor. Return value of 2 executes the allocateItemRelQuantityForConfigurableItemChain processor.

`allocateItemRelQuantityChain`

This processor executes the `allocateItemRelQuantity` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/AllocateItemRelQuantityChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`allocateItemRelQuantityForConfigurableItemChain`

This processor executes the `allocateItemRelQuantityForConfigurableItem` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/AllocateItemRelQuantityForConfigurableItemChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

allocateItemRelQuantity Pipeline Chain

The `allocateItemRelQuantity` chain is executed when called by the `allocateItemRelationship` chain.

The following sections describe each processor in the pipeline chain.

`handleItemRelationshipState`

This processor checks the current state of the `ShippingGroupCommerceItem` relationship. If it is BACK_ORDERED, control passes to `purchaseItemOffBackOrder`. If it is PRE_ORDERED, control passes to `purchaseItemOffPreOrder`. Otherwise, control passes to `purchaseItem`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleItemRelationshipState

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the purchaseItem processor. Return value of 2 executes the purchaseItemOffPreOrder processor. Return value of 3 executes the purchaseItemOffBackOrder processor.

`purchaseItem`

This processor calls the purchase method of the `InventoryManager`. Depending on the result of the purchase, the state of the relationship is set accordingly. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PurchaseItem

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`purchaseItemOffPreOrder`

This processor calls the `purchaseOffPreorder` method of the `InventoryManager`. Depending on the result of the purchase, the state of the relationship is set accordingly. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PurchaseItemOffPreOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`purchaseItemOffBackOrder`

This processor calls the `purchaseOffBackorder` method of the `InventoryManager`. Depending on the result of the purchase, the state of the relationship is set accordingly. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PurchaseItemOffBackOrder

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

allocateItemRelQuantityForConfigurableItem Pipeline Chain

The `allocateItemRelQuantityForConfigurableItem` chain is executed when called by the `allocateItemRelationship` chain.

The following section describes the processor in the pipeline chain.

`purchaseConfigurableItem`

Attempts to allocate the configurable commerce item and sub items from the inventory system. Depending upon the result of the allocation, the state of the relationship is set accordingly.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PurchaseConfigurableItem

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

splitShippingGroupForAvailability Pipeline Chain

Executed when called by the `processHardgoodShippingGroup` chain.

The following sections describe each processor in the pipeline chain.

`shipAsItemsAreAvailable`

This processor checks to make sure that the fulfiller is configured to allow partial shipments. If it is not, chain execution stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ShipAsItemsAreAvailable

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the splitShippingGroupForAvailableItems processor.

`splitShippingGroupForAvailableItems`

This processor splits the shipping group into two shipping groups – one that contains all items in state PENDING_DELIVERY, and one that contains all items in states that indicate they are not ready for shipment.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/SplitShippingGroupForAvailableItems

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleHardgoodUpdateInventory

The `handleHardgoodUpdateInventory` chain is executed when `HardgoodFulfiller` receives an `UpdateInventory` message.

The following sections describe each processor in the pipeline chain.

`retrieveOrderWaitingShipMap`

This processor compiles a `HashMap`, where the keys are Order Ids and the values are sets of shipping group Ids whose quantities could not previously be allocated from inventory. This `HashMap` is placed in the pipeline's parameter map.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/RetrieveOrderWaitingShipMap

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the handleOrderWaitingShipMapChain processor.

`handleOrderWaitingShipMapChain`

This processor iterates through the `HashMap` compiled in the previous processor, and executes the `handleOrderWaitingShipMap` chain for each item.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleOrderWaitingShipMapChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleOrderWaitingShipMap Pipeline Chain

The `handleOrderWaitingShipMap` chain is executed when called by the `handleHardgoodUpdateInventory` chain.

The following sections describe each processor in the pipeline chain.

`lockMessage`

This processor uses the `ClientLockManager` to guarantee that only one thread dealing with a message for a given key is running through the system at any moment in time. The key used to acquire the lock is returned by the method `getKeyForMessage()`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/LockMessage

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the loadOrder4 processor.

`loadOrder4`

This processor loads the given order from the repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `processHardgoodShippingGroupsChain1` processor.

`processHardgoodShippingGroupsChain1`

This processor iterates through the shipping groups contained in the order, and runs the `processHardgoodShippingGroup` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ProcessHardgoodShippingGroupsChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the `updateOrderRepository4` processor.

`updateOrderRepository4`

This processor updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification4` processor.

`sendModifyOrderNotification4`

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleHardgoodModifyOrder Pipeline Chain

The `handleHardgoodModifyOrder` chain is executed when `HardgoodFulfiller` receives a `ModifyOrder` message

The following sections describe each processor in the pipeline chain.

`extractOrderId4`

This processor attempts to extract the ID of the order from the `OrderId` property of the `ModifyOrder` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `loadOrder5` processor.

`loadOrder5`

This processor loads the order from the repository

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `handleModificationClassType2` processor.

`handleModificationClassType2`

This processor determines if the modifications listed in the `ModifyOrder` message are valid. If so, it calls the appropriate processor chains. The only chain that will be called from this processor is `performHardgoodTargetIdModification`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationClassType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the `updateOrderRepository5` processor.

`updateOrderRepository5`

This processor updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification5`processor.

`sendModifyOrderNotification5`

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

performHardgoodIdTargetModification Pipeline Chain

The `performHardgoodIdTargetModification` chain is executed when called by the `handleHardgoodModifyOrder` chain.

The following sections describe each processor in the pipeline chain.

`handleModificationTargetType2`

This processor determines which processor to pass control to by looking at the `TargetType` property of the `IdTargetModification`. If the `TargetType` is TARGET_SHIPPING_GROUP, control passes to `performHardgoodShippingGroupModificationChain`. If the `TargetType` is TARGET_ITEM, control passes to `performHardgoodItemModificationChain`. If `TargetType` is TARGET_RELATIONSHIP, control passes to `performHardgoodRelationshipModificationChain`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationTargetType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 2 executes the performHardgoodShippingGroupModificationChain processor. Return value of 3 executes the performHardgoodItemModificationChain processor. Return value of 4 executes the performHardgoodRelationshipModificationChain processor.

`performHardgoodShippingGroupModificationChain`

This processor executes the `performHardgoodShippingGroupModification` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PerformHardgoodShippingGroupModificationChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

performHardgoodItemModificationChain

This processor executes the performHardgoodItemModification chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PerformHardgoodItemModificationChain

PipelineProcessor object: atg.commerce.pricing.processor.ProcSendScenarioEvent

Transitions: None. This is the last link in the chain and causes the PipelineManager to return to the caller.

performHardgoodRelationshipModificationChain

This processor executes the performHardgoodRelationshipModification chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/PerformHardgoodRelationshipModificationChain

PipelineProcessor object: atg.commerce.pricing.processor.ProcSendScenarioEvent

Transitions: None. This is the last link in the chain and causes the PipelineManager to return to the caller.

performHardgoodShippingGroupModification Pipeline Chain

The performHardgoodShippingGroupModification chain is executed when called by the performHardgoodIdTargetModification chain.

The following sections describe each processor in the pipeline chain.

handleModificationType3

This processor determines the type of modification requested by looking at the ModificationType property of the modification. If the ModificationType is ADD_MODIFICATION, control passes to addHardgoodShippingGroup. If the ModificationType is REMOVE_MODIFICATION, control passes to removeHardgoodShippingGroupChain. If the ModificationType is neither of these, control passes to updateHardgoodShippingGroupChain.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationType

PipelineProcessor object: atg.commerce.fulfillment.processor.ProcHandleModificationType

Transitions: Return value of 1 executes the addHardgoodShippingGroup processor. Return value of 2 executes the removeHardgoodShippingGroupChain processor. Return value of 3 executes the updateHardgoodShippingGroupChain processor.

addHardgoodShippingGroup

This type of modification is currently not supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be returned in a ModifyOrderNotification message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`removeHardgoodShippingGroupChain`

This processor executes the `removeHardgoodShippingGroup` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/RemoveHardgoodShippingGroupChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`updateHardgoodShippingGroupChain`

This processor executes the `updateHardgoodShippingGroup` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/UpdateHardgoodShippingGroupChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

removeHardgoodShippingGroup Pipeline Chain

The `removeHardgoodShippingGroup` chain is executed when called by the `performHardgoodShippingGroupModification` chain.

The following sections describe each processor in the pipeline chain.

`verifyShippingGroupForRemoval`

This processor verifies that the shipping group exists and is in a proper state for removal. If it is not, then execution of this chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/VerifyShippingGroupForRemoval

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the removeShipItemRelsFromShipGroupChain processor.

`removeShipItemRelsFromShipGroupChain`

This processor iterates through the `ShippingGroupCommerceItem` relationships contained within the shipping group, and calls the `removeShipItemRelsFromShipGroup` chain for each relationship.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/RemoveShipItemRelsFromShipGroupChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

removeShipItemRelsFromShipGroup Pipeline Chain

The `removeShipItemRelsFromShipGroup` chain is executed when called by the `removeHardgoodShippingGroup` chain.

The following sections describe each processor in the pipeline chain.

`verifyShipItemRelationshipForRemoval`

This processor verifies that the relationship is in a proper state for removal. If the state is REMOVED or PENDING_REMOVE, then chain execution stops. If the state is DELIVERED or PENDING_RETURN, then the state of the modification is set to FAILED, an error is logged, and chain execution stops. Otherwise, chain execution continues.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/VerifyShipItemRelationshipForRemoval

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the removeShipItemRelationship processor.

`removeShipItemRelationship`

This processor subtracts the quantity of the commerce item contained in the relationship from the commerce item contained in the order. Sets the state of the shipping group to REMOVED.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/RemoveShipItemRelationship

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

updateHardgoodShippingGroup Pipeline Chain

The `updateHardgoodShippingGroup` chain is executed when called by the `performHardgoodShippingGroupModification` chain.

The following sections describe each processor in the pipeline chain.

`handleShippingGroupState1`

This processor checks the `NewValue` property of the modification to determine what state the modification is requesting that the shipping group be set to. If the value is SHIP_SHIPPING_GROUP,

control passes to `shippingGroupHasShippedChain`. If the value is anything else then control passes to `modificationNotSupported5`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleShippingGroupState

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 4 executes the shippingGroupHasShippedChain processor. Return value of 5 executes the `modificationNotSupported5` processor.

`shippingGroupHasShippedChain`

This processor executes the `shippingGroupHasShipped` chain. After execution, execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ShippingGroupHasShippedChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`modificationNotSupported5`

Sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

shippingGroupHasShipped Pipeline Chain

Executed when called by the `updateHardgoodShippingGroup` chain, or the `shipShippingGroups` chain.

The following sections describe each processor in the pipeline chain.

`verifyShippingGroupForCompletion`

This processor verifies that the shipping group's state is PENDING_SHIPMENT. If it is, control is passed to the next processor. If the state is NO_PENDING_ACTION, execution of the chain stops. If the state is anything else, the state of the modification is set to STATUS_FAILED, and execution of the chain stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/VerifyShippingGroupForCompletion

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the finishShippingGroup processor.

finishShippingGroup

This processor sets the state of each `ShippingGroupCommerceItem` relationship in the shipping group to DELIVERED, sets the state of the shipping group to NO_PENDING_ACTION, and sets the shipped date in the shipping group.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/FinishShippingGroup

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

performHardgoodItemModification Pipeline Chain

The `performHardgoodItemModification` chain is executed when called by the `performHardgoodIdTargetModification` chain.

The following sections describe each processor in the pipeline chain.

handleModificationType4

This processor determines the type of modification requested by looking at the `ModificationType` property of the modification. If the `ModificationType` is ADD_MODIFICATION, control passes to `addHardgoodItem`. If the `ModificationType` is REMOVE_MODIFICATION, control passes to `removeHardgoodItem`. If the `ModificationType` is neither of these, control passes to `updateHardgoodItem`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationType`

Transitions: Return value of 1 executes the `addHardgoodItem` processor. Return value of 2 executes the `removeHardGoodItem` processor. Return value of 3 executes the `udpateHardGoodItem` processor.

addHardgoodItem

This type of modification is not currently supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

removeHardgoodItem

This type of modification is not currently supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`updateHardgoodItem`

This type of modification is not currently supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

performHardgoodRelationshipModification Pipeline Chain

The `performHardgoodRelationshipModification` chain is executed when called by the `performHardgoodIdTargetModification` chain.

The following sections describe each processor in the pipeline chain.

`handleModificationType5`

This processor determines the type of modification requested by looking at the `ModificationType` property of the modification. If the `ModificationType` is ADD_MODIFICATION, control passes to `addHardgoodRelationship`. If the `ModificationType` is REMOVE_MODIFICATION, control passes to `removeHardgoodRelationship`. If the `ModificationType` is neither of these, control passes to `updateHardgoodRelationship`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleModificationType`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleModificationType`

Transitions: Return value of 1 executes the `addHardgoodRelationship` processor. Return value of 2 executes the `removeHardGoodRelationship` processor. Return value of 3 executes the `updateHardGoodRelationship`.

`addHardgoodRelationship`

This type of modification is not currently supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`removeHardgoodRelationship`

This type of modification is not currently supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`updateHardgoodRelationship`

This type of modification is not currently supported. This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleHardgoodModifyOrderNotification Pipeline Chain

The `handleHardgoodModifyOrderNotification` chain is executed when `HardgoodFulfiller` receives a `ModifyOrderNotification` message.

The following sections describe each processor in the pipeline chain.

`extractOrderId5`

This processor attempts to extract the ID of the order from the `OrderId` property of the `ModifyOrderNotification` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `loadOrder6` processor.

`loadOrder6`

This processor loads the given order from the order repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `handleModificationClassType3` processor.

`handleModificationClassType3`

This processor determines if the modifications listed in the `ModifyOrderNotification` message are valid. If the modifications are valid, it calls the appropriate processor chains, and upon conclusion, passes control to the `updateOrderRepository6` processor. The only chain that this processor could trigger is `handleHardgoodShipGroupUpdateModification`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationClassType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the updateOrderRepository6 processor.

`updateOrderRepository6`

This processor updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification6` processor.

`sendModifyOrderNotification6`

If any changes were made during the execution of this chain, this processor sends a `ModifyOrderNotification` message with the list of modifications using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleHardgoodShipGroupUpdateModification Pipeline Chain

The `handleHardgoodShipGroupUpdateModification` chain is executed when called by the `handleHardgoodModifyOrderNotification` chain.

The following sections describe each processor in the pipeline chain.

`extractShippingGroupIds`

This processor extracts the shipping group IDs from the `ModifyOrderNotification` message and places them in the pipeline's parameter map.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ExtractShippingGroupIds

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the processHardgoodShippingGroupsChain2 processor.

`processHardgoodShippingGroupsChain2`

This processor iterates through the list of shipping groups and executes the `processHardgoodShippingGroup` chain for each group.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ProcessHardgoodShippingGroupsChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

shipPendingShippingGroups Pipeline Chain

The following sections describe each processor in the pipeline chain.

`retrieveOrderPendingShipMap`

This processor compiles a `HashMap` from the Order repository where the keys are the ID of the orders that have shipping groups that are PENDING_SHIPMENT, and the values are sets of shipping group Ids whose states are PENDING_SHIPMENT. This `HashMap` is then placed in the pipeline's parameter map.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/RetrieveOrderPendingShipMap

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the handleOrderPendingShipMapChain processor.

`handleOrderPendingShipMapChain`

This processor iterates through the `HashMap` compiled in the previous processor, and then iterates through each shipping group within each value, and runs the `shipShippingGroup` chain for each shipping group.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleOrderPendingShipMapChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

shipShippingGroup Pipeline Chain

The `shipShippingGroup` chain is executed when called by the `shipPendingShippingGroups` chain.

The following sections describe each processor in the pipeline chain.

lockMessage1

This processor uses the `ClientLockManager` to guarantee that only one thread dealing with a message for a given key is running through the system at any moment in time. The key used to acquire the lock is returned by the method `getKeyForMessage()`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/LockMessage

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the loadOrder7 processor.

loadOrder7

This processor loads the given order from the repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `shippingGroupHasShippedChain1` processor.

shippingGroupHasShippedChain1

This processor executes the `shippingGroupHasShipped` chain.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ShippingGroupHasShippedChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the `updateOrderRepository7` processor.

updateOrderRepository7

This processor updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification7` processor.

sendModifyOrderNotification7

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleElectronicFulfillOrderFragment Pipeline Chain

The `handleElectronicsFulfillOrderFragment` chain is executed when a `ElectronicFulfiller` receives a `FulfillOrderFragment` message.

The following sections describe each processor in the pipeline chain.

`extractOrderId6`

This processor attempts to extract the ID of the order from the `OrderId` property of the `FulfillOrderFragment` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `handleRetrieveOrder2` processor.

`handleRetrieveOrder2`

This processor determines the method by which the Order should be loaded. If the order ID was successfully extracted in the `extractOrderId` processor, then we move to the loadOrder8 processor. If not, then it checks the Boolean `LookUpOrderIdFromOrder` property of the `OrderFulfiller`. If true, we move to the loadSaveOrder2 processor. If false, then it throws an `InvalidParameterException`, and chain execution stops.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/HandleRetrieveOrder`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcHandleRetrievingOrder`

Transitions: Return value of 1 executes the `loadSaveOrder2` processor. Return value of 2 executes the `loadOrder8` processor.

`loadSaveOrder2`

This processor checks to see if the Order exists in the order repository, using the `OrderExists` method of `OrderManager`, and using the ID of the serialized order within the `FulfillOrderFragment` message as the parameter. If the order exists, the processor loads the order. If it does not, then fulfillment is using a different repository than the order placement system. The processor then saves the order from the message into the repository. In either case, the chain then moves to the `processElectronicShippingGroupsChain` processor.

`loadOrder8`

This processor loads the order from the repository. Control then passes to `processElectronicShippingGroupsChain`.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `processElectronicShippingGroupsChain` processor.

`processElectronicShippingGroupsChain`

This processor iterates through the shipping groups contained in the `FulfillOrderFragment` message, and runs the `processElectronicShippingGroup` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/ProcessElectronicShippingGroupsChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the `updateOrderRepository8` processor.

`updateOrderRepository8`

This processor updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification8` processor.

`sendModifyOrderNotification8`

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

processElectronicShippingGroup Pipeline Chain

The `processElectronicShippingGroup` chain is executed when called by the `handleElectronicFulfillOrderFragment` chain, or the `handleElectronicShipGroupUpdateModification`.

The following section describes the processor in the pipeline chain.

`allocateElectronicGoodChain`

This processor iterates through all of the `ShippingGroupCommerceItem` relationships within the given shipping group, obtains the quantity in that relationship, and for each distinct item to be sent, it executes the `allocateElectronicGood` chain.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/AllocateElectronicGoodChain

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

allocateElectronicGood Pipeline Chain

The `allocateElectronicGood` chain is executed when called by the `processElectronicShippingGroup` chain.

The following sections describe each processor in the pipeline chain.

`createElectronicGood`

This processor creates a gift certificate by using the `createClaimableGiftCertificate` method of `ClaimableManager`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/CreateElectronicGood

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the handleElectronicSenderType processor.

`handleElectronicSenderType`

This processor determines how the electronic good is to be delivered by checking `useTemplateEmailSender` property of `ElectronicFulfiller`. If it is true, control passes to `deliverElectronicGoodByTemplate`. Otherwise, control passes to `deliverElectronicGoodByListener`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleElectronicSenderType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the deliverElectronicGoodByTemplate processor. Return value of 2 executes the deliverElectronicGoodByListener processor

`deliverElectronicGoodByTemplate`

This processor sends the electronic good out via e-mail using the `GiftCertificateEmailTemplate` of the `OrderFulfiller`. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/DeliverElectronicGoodByTemplate

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

`deliverElectronicGoodByListener`

This processor delivers the electronic good out via e-mail using `EmailListener` property of the `OrderFulfiller`. Execution of this chain then stops.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/DeliverElectronicGoodByListener

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleElectronicModifyOrder Pipeline Chain

The `handleElectronicModifyOrder` chain is executed when a `ElectronicFulfiller` receives a `ModifyOrder` message.

The following sections describe each processor in the pipeline chain.

`extractOrderId7`

This processor attempts to extract the ID of the order from the `OrderId` property of the `ModifyOrder` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `loadOrder9` processor.

`loadOrder9`

This processor loads the given order from the repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `handleModificationClassType4` processor.

`handleModificationClassType4`

This processor determines if the modifications listed in the `ModifyOrder` message are valid. If so, it calls the appropriate processor chains. Currently, `ElectronicFulfiller` does not support handling of `ModifyOrder` messages, so this processor will always pass control to `modificationNotSupported6`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationClassType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 2 executes the `modificationNotSupported6` processor.

`modificationNotSupported6`

This processor sets the status of the modification to STATUS_FAILED, and adds the modification to the list to be sent out in a `ModifyOrderNotification` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ModificationNotSupported`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcModificationUnsupported`

Transitions: Return value of 1 executes the `sendModifyOrderNotification9` processor.

`sendModifyOrderNotification9`

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleElectronicModifyOrderNotification Pipeline Chain

The `handleElectronicModifyOrderNotification` chain is executed when a `ElectronicFulfiller` receives a `ModifyOrderNofication` message.

The following sections describe each processor in the pipeline chain.

`extractOrderId8`

This processor attempts to extract the ID of the order from the `OrderId` property of the `ModifyOrderNotification` message.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractOrderId`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractOrderId`

Transitions: Return value of 1 executes the `loadOrder10` processor.

`loadOrder10`

This processor loads the order from the repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/LoadOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcLoadOrderRepository`

Transitions: Return value of 1 executes the `handleModificationClassType5` processor.

`handleModificationClassType5`

This processor determines if the modifications listed in the `ModifyOrder` message are valid. If so, it calls the appropriate processor chains. The only chain that this processor can trigger is `handleElectronicShipGroupUpdateModification`.

Transactional mode: TX_MANDATORY

Nucleus component: /atg/commerce/fulfillment/processor/HandleModificationClassType

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: Return value of 1 executes the `updateOrderRepository9` processor.

`updateOrderRepository9`

This processor updates the order in the repository with any changes that may have been made during the execution of this chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/UpdateOrderRepository`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcUpdateOrderRepository`

Transitions: Return value of 1 executes the `sendModifyOrderNotification10` processor.

`sendModifyOrderNotification10`

This processor sends a `ModifyOrderNotification` message with the list of modifications performed during the execution of this chain using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendModifyOrderNotification`

PipelineProcessor object: `atg.commerce.pricing.processor.ProcSendScenarioEvent`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

handleElectronicShipGroupUpdateModification Pipeline Chain

Executed when called by the `handleElectronicModifyOrderNotification` chain.

The following sections describe each processor in the pipeline chain.

`extractShippingGroupIds1`

This processor extracts the shipping group IDs from the `ModifyOrderNotification` message and places them in the pipeline's parameter map.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ExtractShippingGroupIds`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcExtractShippingGroupIds`

Transitions: Return value of 1 executes the processElectronicShippingGroupsChain1 processor.

`processElectronicShippingGroupsChain1`

This processor iterates through the shipping groups contained in the `ModifyOrderNotification` message, and runs the `processElectronicShippingGroup` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ProcessElectronicShippingGroupsChain`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcProcessShippingGroups`

Transitions: None. This is the last link in the chain and causes the `PipelineManager` to return to the caller.

sendOrderToFulfiller Pipeline Chain

The following sections describes the processor in the pipeline chain.

`sendFulfillOrderFragment1`

This processor sets the order's and all the shipping groups' states to PROCESSING, then builds a `FulfillOrderFragment` message and sends it using JMS.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/SendFulfillOrderFragment`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcSendFulfillOrderFragment`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

processHardgoodShippingGroups Pipeline Chain

The following section describes the processor in the pipeline chain.

`processHardgoodShippingGroupsChain3`

This processor iterates through the shipping groups, and runs the `processHardgoodShippingGroup` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ProcessHardgoodShippingGroupsChain`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcProcessShippingGroups`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

retrieveWaitingShipMap Pipeline Chain

The following section describes the processor in the pipeline chain.

`retrieveOrderWaitingShipMap1`

This processor compiles a `HashMap`, where the keys are Order IDs and the values are sets of shipping group IDs whose quantities could not previously be allocated from inventory. This `HashMap` is placed in the pipeline's parameter map.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/RetrieveOrderWaitingShipMap`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcRetrieveOrderWaitingShipMap`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

processElectronicShippingGroups Pipeline Chain

The following section describes the processor in the pipeline chain.

`processElectronicShippingGroupsChain2`

This processor iterates through the shipping groups contained in the `FulfillOrderFragment` message, and runs the `processElectronicShippingGroup` chain for each.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/fulfillment/processor/ProcessElectronicShippingGroupsChain`

PipelineProcessor object: `atg.commerce.fulfillment.processor.ProcProcessShippingGroups`

Transitions: None. This is the only link in the chain and causes the `PipelineManager` to return to the caller.

Order Approval Pipelines

Several pipeline chains manage the different phases of the approval process.

The `.xml` configuration file for these pipeline chains is located in a `.jar` file at `<ATG10dir>/DCS/config/config.jar`. The Nucleus location for their processors is `/atg/commerce/approval/processor/`.

This section describes each pipeline chain and processor used in the order approval process.

Note: By default, both the `approveOrder` and `checkRequiresApproval` pipeline chains are configured to run in the context of the same transaction as the calling chain, `processOrder`. This prevents the situation where a processor in either of these pipelines throws an exception that causes them to roll back but does not cause the `processOrder` pipeline to roll back as well. In this problematic situation, the `processOrder` pipeline would finish executing without notifying the user of the error condition that exists.

approveOrder Pipeline Chain

The `approveOrder` pipeline determines whether the given order already is approved. If the order isn't already approved, it determines whether an approval for the order is required.

The `approveOrder` pipeline chain is executed by the `executeApproveOrderChain` processor in the `processOrder` pipeline chain. The `approveOrder()` method adds the given Order and the `ApprovalPipelineManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`verifyApproval`

This processor checks whether the given order has already been approved.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/VerifyApproval`

PipelineProcessor object: `atg.commerce.approval.processor.ProcVerifyApproval`

Transitions: Returns a value of 0 (STOP_CHAIN_EXECUTION_AND_COMMIT) if the order has already been approved; this stops execution of the `approveOrder` chain and resumes the `processOrder` chain to complete checkout. Returns a value of 1 if the order has not already been approved; this executes the next processor, `runCheckRequiresApprovalChain`.

`runCheckRequiresApprovalChain`

This processor executes the `checkRequiresApproval` pipeline chain. The properties file for the `/atg/commerce/approval/processor/RunCheckRequiresApproval` component specifies `checkRequiresApproval` in the `chainToRun` property.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/RunCheckRequiresApprovalChain`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Returns a value of 0 (STOP_CHAIN_EXECUTION_AND_COMMIT) if the order does not require approval; this stops execution of `approveOrder` so the order can proceed through checkout. Returns a value of 1 if the order requires approval; this executes the next processor, `addApproverIdsToOrder`.

`addApproverIdsToOrder`

This processor adds to the order the list of profile IDs for the users who can approve the customer's order. This list is obtained from the customer's `approvers` profile property and is added to the order's `authorizedApproverIds` property.

If the customer's `approvers` profile property is unset and the `AddApproverIdsToOrder.allowCheckoutIfApproversNotDefined` property is set to false (which it is by default), then an `ApprovalException` is thrown.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddApproverIdsToOrder`

PipelineProcessor object: `atg.commerce.approval.processor.ProcAddApproverIdsToOrder`

Transitions: Return value of 1 executes `changeOrderToPendingApproval` next. However, if the customer's `approvers` profile property is unset and the `AddApproverIdsToOrder.allowCheckoutIfApproversNotDefined` property is set to true, the processor returns a value of 0 (STOP_CHAIN_EXECUTION_AND_COMMIT); this stops execution of `approveOrder` so the order can proceed through checkout.

`changeOrderToPendingApproval`

This processor sets the order's state to PENDING_APPROVAL.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/ChangeOrderToPendingApproval`

PipelineProcessor object: `atg.commerce.order.processor.ProcChangeOrderState`

Transitions: Return value of 1 executes `addApprovalSystemMessagesToOrder` next.

`addApprovalSystemMessagesToOrder`

This processor adds to the order the list of system messages that correspond to the conditions that triggered an approval being required. An example might be "order limit exceeded." This list is added to the order's `approvalSystemMessages` property. The system messages are defined by the processors in the `checkRequiresApproval` pipeline chain.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddApprovalSystemMessagesToOrder`

PipelineProcessor object:
`atg.commerce.approval.processor.ProcAddApprovalSystemMessagesToOrder`

Transitions: Return value of 1 executes `saveOrder` next.

`saveOrder`

This processor saves the order in its present state to the Order Repository.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/order/processor/UpdateOrder`

PipelineProcessor object: `atg.commerce.order.processor.ProcUpdateOrder`

Transitions: Return value of 1 executes `sendApprovalRequiredMessage` next.

`sendApprovalRequiredMessage`

This processor sends a message to the `/Approval/Scenarios` JMS message topic; the message includes the order requiring approval and the profile repository item for the customer associated with the order. The message can then be used to execute scenarios.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SendApprovalRequiredMessage`

PipelineProcessor object: `atg.commerce.approval.processor.ProcSendApprovalRequiredMessage`

Transitions: None. This is the last processor in the pipeline, which causes the `ApprovalPipelineManager` to return to the caller.

checkRequiresApproval Pipeline Chain

The `checkRequiresApproval` pipeline chain is the chain that actually checks whether an approval is required for a customer's order. The default implementation of this chain checks the `approvalRequired` property in the customer's profile. If the `approvalRequired` property is true, then approval is required for the customer. An error is then added to the `PipelineResult` object, which tells the system that an approval is required, and the reason that approval is required is stored in the `errorMessages` property of the Order. This reason for approval is later added to the order's `approvalSystemMessages` property by the `approveOrder` chain's `addApprovalSystemMessagesToOrder` processor. If the `approvalRequired` property is false, then approval isn't required for the customer.

The `checkRequiresApproval` pipeline chain is executed by the `runCheckRequiresApprovalChain` processor in the `approveOrder` pipeline chain. The `checkRequiresApproval()` method adds the given `Order` and the `ApprovalPipelineManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

Note: You can edit this chain to create specific requirements for whether an approval is required for a given customer. For example, you might want to include a processor that checks the total amount of the customer's order against an order limit in the customer's profile. If the order amount exceeds the specified limit, then approval for the customer's order would be required. Similarly, you might want to include a processor that checks the manufacturers of the items in the customer's order against a list of preferred suppliers in the customer's profile. If a manufacturer isn't in the list of preferred suppliers, then approval for the customer's order would be required.

The following section describes the processor in the pipeline chain.

`checkProfileApprovalRequirements`

This processor checks the `approvalRequired` property in the customer's profile. If the property is true, then approval is required for the customer. If the property is false, then approval isn't required for the customer.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/CheckProfileApprovalRequirements`

PipelineProcessor object: `atg.commerce.order.processor.ProcPropertyRestriction`

Transitions: None. This is the only processor in the pipeline, which causes the `ApprovalPipelineManager` to return to the caller.

orderApproved Pipeline Chain

The `orderApproved` pipeline chain processes an approval of a given order. When an approver submits her approval of an order via a form using the `ApprovalFormHandler`, the form handler's `handleApproveOrder()` method executes the `orderApproved` pipeline chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`addApproverIdToOrderForApproval`

This processor adds the profile ID for the approver who approved the order to the order's `approverIds` property. The `approverIds` property contains a list of approvers who have approved or rejected the order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddApproverIdToOrder`

PipelineProcessor object: `atg.commerce.approval.processor.ProcAddApproverIdToOrder`

Transitions: Return value of 1 executes `addApproverMessagesToOrderForApproval` next.

`addApproverMessagesToOrderForApproval`

This processor adds the message that the approver attaches to the order to the list of messages in the order's `approverMessages` property. The message typically indicates the reason for approval. It is passed to the `orderApproved` chain by the `ApprovalFormHandler` form handler, which is used to process the approval.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddApproverMessagesToOrder`

PipelineProcessor object: `atg.commerce.approval.processor.ProcAddApproverMessagesToOrder`

Transitions: Return value of 1 executes `sendApprovalUpdateMessageForApproval` next.

`sendApprovalUpdateMessageForApproval`

This processor sends an `ApprovalUpdate` message that includes the order requiring approval and the profile repository item for the customer associated with the order to both the `/Approval/ApprovalUpdate` JMS message queue and the `/Approval/Scenarios` JMS message topic. The `approvalStatus` property of the message is set to "approved". The `ApprovalCompleteService` listens for the message sent to the `/Approval/ApprovalUpdate JMS message queue`. The message sent to `/Approval/Scenarios` can be used to execute scenarios.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SendApprovalUpdateMessage`

PipelineProcessor object: `atg.commerce.approval.processor.ProcSendApprovalMessage`

Transitions: None. This is the last processor in the pipeline, which causes the `ApprovalPipelineManager` to return to the caller.

orderRejected Pipeline Chain

The `orderRejected` pipeline chain processes a rejection of a given order. When an approver submits her rejection of an order via a form using the `ApprovalFormHandler`, the form handler's `handleRejectOrder()` method executes the `orderRejected` pipeline chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`addApproverIdToOderForRejection`

This processor adds the profile ID for the approver who rejected the order to the order's `approverIds` property. The `approverIds` property contains a list of approvers who have approved or rejected the order.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddApproverIdToOrder`

PipelineProcessor object: `atg.commerce.approval.processor.ProcAddApproverIdToOrder`

Transitions: Return value of 1 executes `addApproverMessagesToOrderForRejection` next.

`addApproverMessagesToOrderForRejection`

This processor adds the message that the approver attaches to the order to the list of messages in the order's `approverMessages` property. The message typically indicates the reason for rejection. It is passed to the `orderRejected` chain by the `ApprovalFormHandler` form handler, which is used to process the rejection.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddApproverMessagesToOrder`

PipelineProcessor object: `atg.commerce.approval.processor.ProcAddApproverMessagesToOrder`

Transitions: Return value of 1 executes `sendApprovalUpdateMessageForRejection` next.

`sendApprovalUpdateMessageForRejection`

This processor sends an `ApprovalUpdate` message that includes the order requiring approval and the profile repository item for the customer associated with the order to both the `/Approval/ApprovalUpdate` JMS message queue and the `/Approval/Scenarios` JMS message topic. The `approvalStatus` property of the message is set to "rejected". The `ApprovalCompleteService` listens for the message sent to the `/Approval/ApprovalUpdate JMS message queue`. The message sent to `/Approval/Scenarios` can be used to execute scenarios.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SendApprovalUpdateMessage`

PipelineProcessor object: `atg.commerce.approval.processor.ProcSendApprovalMessage`

Transitions: None. This is the last processor in the pipeline, which causes the `ApprovalPipelineManager` to return to the caller.

checkApprovalComplete Pipeline Chain

The `checkApprovalComplete` pipeline determines whether the approval process for the given order is complete. The `checkApprovalComplete` pipeline chain is executed by `ApprovalCompleteService` when the service receives an `ApprovalUpdate` message from the `/Approval/ApprovalUpdate` JMS message queue. The `checkApprovalComplete()` method adds the given Order and the `ApprovalPipelineManager` to its parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRED.

The following sections describe each processor in the pipeline chain.

`getApprovalCompleteParams`

This processor takes properties from the `ApprovalUpdate` message and adds them to the Map object that is passed to the `ApprovalPipelineManager` for execution of the `checkApprovalComplete` chain. You define the properties to take from the `ApprovalUpdate` message in the `.properties` file of this processor.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/GetApprovalCompleteParams`

PipelineProcessor object: `atg.commerce.approval.processor.ProcPopulatePipelineParams`

Transitions: Return value of 1 executes `approvalCompleteAnalyzer` next.

`approvalCompleteAnalyzer`

This processor determines whether the approval process for the given order is complete. By default, `approvalCompleteAnalyzer` checks whether at least one person has approved or rejected the order. If so, then the approval process for the order is considered to be complete.

Note: You can change the implementation of `approvalCompleteAnalyzer` in order to change the requirements for completion of the approval process.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/ApprovalCompleteAnalyzer`

PipelineProcessor object: `atg.commerce.approval.processor.ProcApprovalCompleteAnalyzer`

Transitions: Returns a value of 0 if the approval process for the order isn't complete (that is, the order requires further approvals); this stops execution of the chain, and the transaction commits. Returns a value of 1 if the order has been approved; this executes `changeOrderToApproved` next. Returns a value of 2 if the order has been rejected; this executes `changeOrderToFailedApproval` next.

`changeOrderToFailedApproval`

This processor sets the order's state to FAILED_APPROVAL, as specified in the `newOrderState` property of the `/atg/commerce/approval/processor/ChangeOrderToFailedApproval` component.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/ChangeOrderToFailedApproval`

PipelineProcessor object: `atg.commerce.order.processor.ProcChangeOrderState`

Transitions: Return value of 1 executes `sendApprovalCompleteMessage` next.

`changeOrderToApproved`

This processor sets the order's state to APPROVED, as specified in the `newOrderState` property of the `/atg/commerce/approval/processor/ChangeOrderToApproved` component.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/ChangeOrderToApproved`

PipelineProcessor object: `atg.commerce.order.processor.ProcChangeOrderState`

Transitions: Return value of 1 executes `completeProcessingOrder` next.

`completeProcessingOrder`

This processor executes the `processOrder` chain, passing the given order to `processOrder` as one of its parameters. The properties file for the `/atg/commerce/approval/processor/CompleteProcessingOrder` component specifies `processOrder` in the `chainToRun` property.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/CompleteProcessingOrder`

PipelineProcessor object: `atg.commerce.order.processor.ProcExecuteChain`

Transitions: Return value of 1 executes `sendApprovalCompleteMessage` next.

`sendApprovalCompleteMessage`

This processor sends a message to the `/Approval/Scenarios` JMS message topic that includes the order requiring approval and the profile repository item for the customer associated with the order. The `approvalStatus` property of the message is set to either `approval_passed` or `approval_failed`, depending on the state of the order. The message can then be used to execute scenarios.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/SendApprovalCompleteMessage`

PipelineProcessor object: `atg.commerce.approval.processor.ProcSendApprovalCompleteMessage`

Transitions: None. This is the last processor in the pipeline, which causes the `ApprovalPipelineManager` to return to the caller.

checkApprovalCompleteError Pipeline Chain

If an error occurs while `ApprovalCompleteService` is processing an `ApprovalComplete` message, the `checkApprovalCompleteError` chain is executed. This chain is a recovery chain that executes logic when an error occurs.

Note: The default implementation of this chain adds the error message to the order's `approvalSystemMessages` property and sets the order's state to FAILED. You can edit the chain to perform different logic to meet your application's needs.

The `checkApprovalCompleteError` pipeline chain is executed by the `checkApprovalCompleteError()` method in the `ApprovalPipelineManager`. The `checkApprovalCompleteError()` method adds the parameters that were passed to the chain in which the error occurred to the `ApprovalPipelineManager`'s parameter list, which is supplied to the executing chain. The pipeline chain's transaction mode is TX_REQUIRES_NEW.

The following sections describe each processor in the pipeline chain.

`addMessageMapperErrorToOrder`

This processor adds the error message to the order's `approvalSystemMessages` property.

Transactional mode: TX_MANDATORY

Nucleus component: `/atg/commerce/approval/processor/AddMessageMapperErrorToOrder`

PipelineProcessor object: `atg.commerce.approval.processor.ProcAddMessageMapperErrorToOrder`

Transitions: Return value of 1 executes `changeOrderToFailed` next.

`changeOrderToFailed`

This processor sets the order's state to FAILED.

Transactional mode: TX_MANDATORY

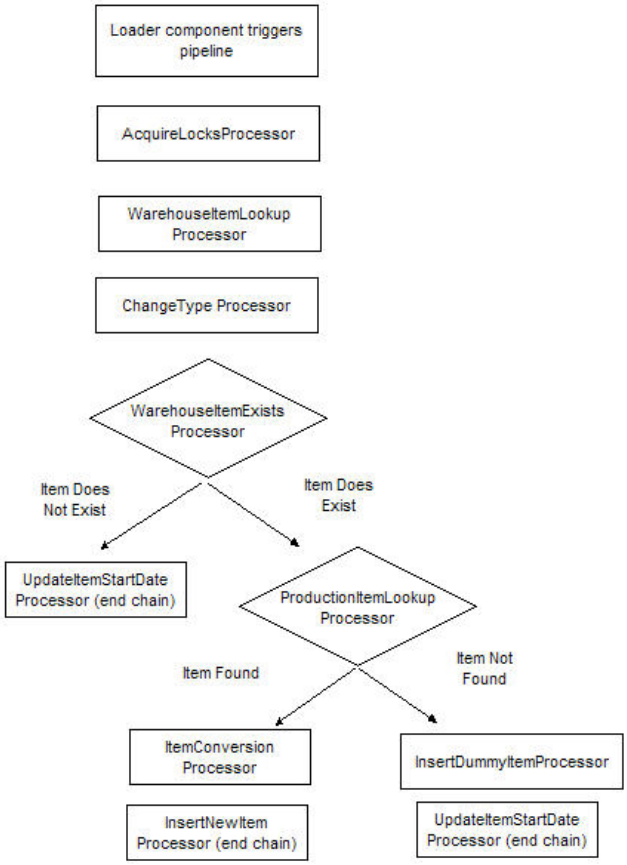
Nucleus component: `/atg/commerce/approval/processor/ChangeOrderToFailed`

PipelineProcessor object: `atg.commerce.order.processor.ProcChangeOrderState`

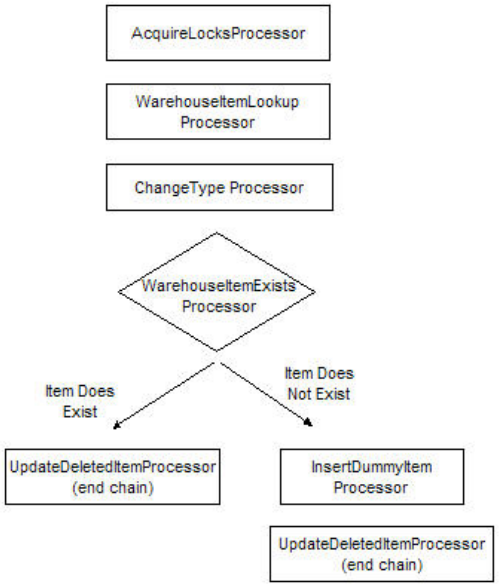
Transitions: None. This is the last processor in the pipeline, which causes the `ApprovalPipelineManager` to return to the caller.

Reporting Pipelines

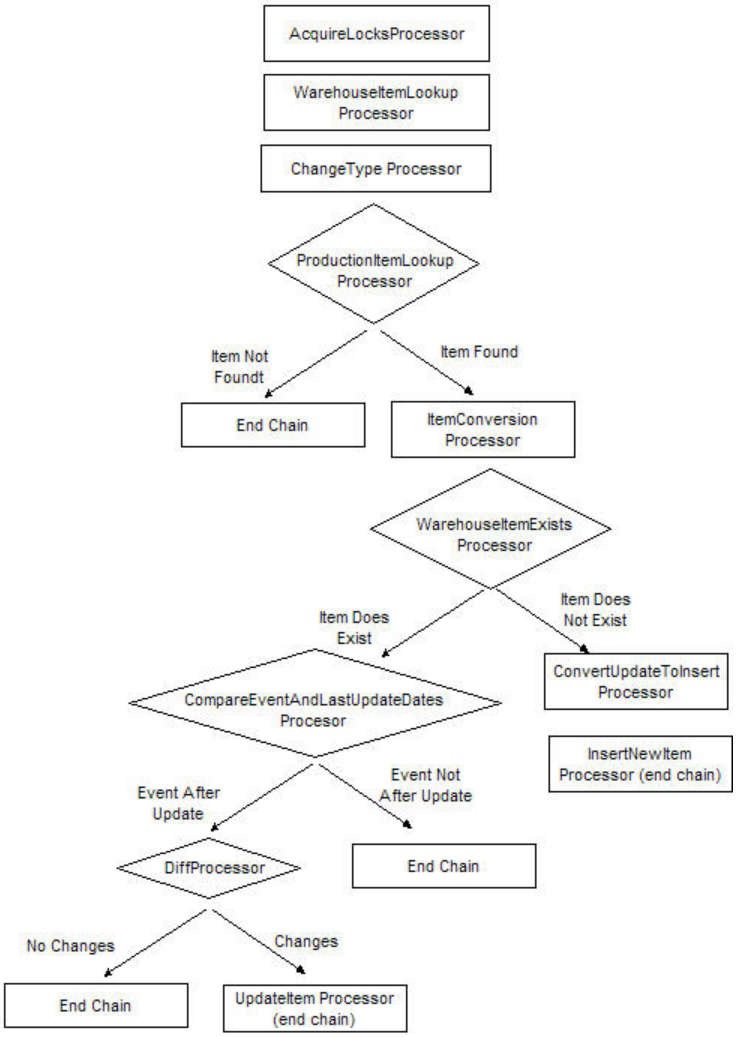
The `dimensionUpdate` pipeline chain is triggered by the `ProductCatalogLoader` and `UserUpdateLoader` components. The following diagrams show the processors in the chain for insert, update, and delete events.



Pipeline Processor Chain for an Insert Event



Pipeline Processor Chain for Delete Events



Pipeline Processor Chain for Update Events

To ensure that locks are acquired correctly for all processed items, this pipeline makes two passes. The first pass identifies which items require locks. The second pass acquires the locks and then makes all inserts and updates to the repository. When the second pipeline pass is finished, all locks are released.

The pipeline uses a map to pass data through the pipeline to each processor. Each processor can get items it needs from the map, add items it is responsible for creating to the map, and update items in the map if needed.

The processors in this pipeline are:

- `AcquireLocksProcessor`—Runs only on the second pipeline pass, when it makes a call on the lock manager to get the locks identified as necessary during the first pass of the pipeline.
- `WarehouseItemLookupProcessor`—In both pipeline passes, looks up the most recent item in the Data Warehouse, using the `itemId` from the map.
- `ChangeTypeProcessor`—In both pipeline passes, determines the type of record change indicated by the log record. If the change type is insert or delete, the next processor that executes is the `WarehouseItemExistsProcessor`. If the change type is update, the next processor is `ProductionItemLookupProcessor`.

- `WarehouseItemExistsProcessor`—In both pipeline passes, determines whether the item being processed already exists in the warehouse. For delete actions, if the item exists, the next processor that executes is the `UpdateDeletedItemProcessor`; if not, it is `InsertDummyItemProcessor`. For insert actions, if the item does not exist, the next processor is the `ProductionItemLookupProcessor`.
- `InsertDummyItemProcessor`—In the first pass, requests a lock for the item to be inserted. In the second pass, creates a dummy item and adds it to the warehouse repository.
- `UpdateDeletedItemProcessor`—In the first pass, requests a lock for the item to be updated. In the second pass, for delete actions, this processor sets the `endDate` and `lastUpdateDate` to the event date, and `deleted` to true. This is the end of the processor pipeline for delete events.
- `ProductionItemLookupProcessor`—In the first processor pass for updates, looks up the item in the production repository. If the item exists, the next processor called is the `ItemConversionProcessor`. If it does not, the chain ends. In the second pass, this processor only does a lookup if the first lookup was unsuccessful.
- `ItemConversionProcessor`—Converts production items into warehouse items. If the conversion finds a reference to a non-existent warehouse item, this processor creates that item by calling the pipeline chain recursively.
- `InsertNewItemProcessor`—In the first pass requests a lock for the item to be updated. In the second pass, creates the new warehouse item.
- `ConvertUpdateToInsertProcessor`—Takes an update and makes changes necessary to allow an insert into the Data Warehouse. Sets the `eventDate` to null, since it is not known when the item was actually inserted into the product catalog. The change type is left as update. This processor performs the conversion only on the second pass, since the item may be added by another loader between the first and second passes.
- `CompareEventAndLastUpdatedDatesProcessor`—In both pipeline passes, this processor looks at the time of an update and the last update time of the Data Warehouse item and compares them, to see if any changes that caused the event have already been put into the Data Warehouse record.
- `DiffProcessor`—Compares the values in the production item to the values in the Data Warehouse. This processor only does its work in the second pass.
- `UpdateItemProcessor`—In the first pass this processor requests a lock for the item to be updated. On the second, it updates the Data Warehouse with changes.

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W