atg®

Version 10.0

# ATG Commerce Guide to Setting Up a Store

**ATG**
**One Main Street**
**Cambridge, MA 02142**
**USA**
**www.atg.com**

# ATG Commerce Guide to Setting Up a Store

## Document Version

Doc10.0 COMMSTOREv1 12/21/10

Contact: ATG • One Main Street, Cambridge, MA 02142, USA • phone 617.386.1000 • fax 617.386.1111 • www.atg.com

# Contents

1 Introduction 1

Commerce Overview 1

Product Catalog 1

Purchasing and Fulfillment Services 2

Targeted Promotions 3

Commerce Services 3

Portal Gears 4

Reporting 4

Multisite Integration 4

Reference Applications 5

Finding What You Need 6

2 ATG Commerce Catalog Administration 9

Organizing Your Product Catalog 9

Commerce Catalog Item Types 11

Viewing Catalogs 11

Viewing Catalogs as a Hierarchy 12

Viewing Catalogs as Lists 15

Creating Catalog Items 15

Creating Catalog Folders 16

Creating Catalogs 16

Creating Root Categories 16

Creating Child Categories 17

Creating Products 17

Creating SKUs 18

Creating Configurable SKUs 18

Creating SKU Bundles 19

Adding Subcatalogs to Catalogs 20

Adding Catalogs to Categories 20

Editing Catalog Items 20

Moving Items 21

Duplicating Items 21

Deleting Items 22

Adding Templates and Images to the Catalog 22

Creating Image and Template Folders 22

Adding Images 23

Adding Templates 24

Associating Images and Templates with Catalog Items 24
Searching for Items in the ACC 25
Preventing Version Conflict 26

3 Inventory and Fulfillment Administration 27
ATG Commerce Inventory Administration 27
Accessing the Inventory Administration Page 27
Viewing the Inventory Display 28
Updating the Inventory 28
Sending Inventory Update Notifications 28
ATG Commerce Fulfillment Administration 29
Accessing the Fulfillment Administration Page 29
Notifying Fulfillment of Order Shipment 29
Reprocessing Shipping Groups 30
Printing an Order 30

4 Managing Price Lists 31
Viewing Existing Price Lists 31
Creating a New Price List Folder 33
Creating a New Price List 33
Changing Prices in an Existing Price List 34
Copying Prices Between Price Lists 35
Setting Bulk and Tiered Pricing 35
Viewing Volume Pricing 36
Setting Volume Pricing 36
Deleting a Price List 37
Assigning Price Lists to Users 37

5 Creating and Maintaining Promotions 39
How Promotions Work 40
Creating Promotions 40
Adding a New Promotion 41
Creating a Discount Rule 49
Specifying the People Who Receive the Promotion 52
Creating Closeness Qualifiers 55
Updating the Promotion 57
Detecting a Closeness Qualifier 57
Adding an Item to a Slot When Users Qualifies for a Closeness Qualifier 58
Sending an Email When Users Qualify for A Closeness Qualifier 59
Removing Closeness Qualifiers From a Slot When They No Longer Apply 60
Setting up Upselling Incentives 61
Sample Upsell Incentives 61
Working with Upsell Actions 63
Disabling Promotions 65
Displaying Promotion Media 65
Setting Up Coupon Promotions 66

Two Types of Coupons 67
Adding a Coupon 67
Delivering Promotions via a URL 68

6 Managing Cost Centers 69
Viewing Existing Cost Centers 69
Adding New Cost Centers 70
Assigning a Default Cost Center to a User 70
Adding, Modifying, and Deleting Cost Centers in a Profile 71
Adding Cost Centers to an Order 73
Adding Items to a Cost Center 74
Tracking Orders by Cost Center 74
Cost Center Classes 74
Using the CostCenterFormHandler Framework 76

7 Using Commerce Elements in Scenarios 85
Using Commerce Event Elements in Scenarios 85
Approval Complete Event 86
Approval Required Event 86
Approval Update Event 87
FulfillOrderFragment 87
Gift Purchased 87
Inventory Threshold Reached 87
Invoice Is Created 88
Invoice Is Removed 88
Invoice Is Updated 88
Item Added to Order 88
Item Quantity Changed in Order 89
Item Removed from Order 89
Modify Order 89
Modify Order Notification 89
Order Changes 89
Order Submitted 90
Orders Merged 90
Payment Group Changes 90
Price Changed 91
Promotion Closeness Disqualification 91
Promotion Closeness Qualification 91
Promotion Offered 91
Promotion Revoked 92
Scenario Added an Item to an Order 92
Scheduled Order Event 92
Shipping Group Changes 92
Update Inventory 93
Uses Promotion 93
Using Commerce Condition Elements in Scenarios 93

Item Where 94
Order Where 94
Using Commerce Action Elements in Scenarios 94
Add Item to Order 95
Fill Related Items to Slot 95
Give Promotion 95
Revoke Promotion 95
Using Scenarios to Cross-Sell and Up-Sell Products 95

8 Managing Abandoned Orders 99
Understanding Order Abandonment 99
Responding to Order Abandonment Activity 102
Creating Scenarios that Respond to Abandonment Activity 103
Testing Scenarios that Respond to Abandonment Activity 105
Scenario Event Elements 106
Scenario Action Elements 107

9 Catalog Navigation and Searching 109
Using the parentCategory Property 109
Displaying Catalog Items 110
Looking Up Items in the Catalog 110
ForEachItemInCatalog Servlet Bean 111
Sending Messages When Items are Viewed 111
Catalog Navigation 112
Displaying Root Categories 112
Displaying Child Categories and Products 113
Historical Navigation 114
Catalog Searching 118
Overview of Catalog Searching 118
Preconfigured Catalog Search Components 119
Configuring the Search Form Handler 120
Configuring Catalog Search Types 121
Combining Catalog Search Types 124
Processing Searches 125
Displaying Search Results 127
Searching Catalogs in Preview Mode 131
Using Search Form Handlers with Internationalized Catalogs 132

10 Implementing Product Comparison 133
Understanding the ProductList Component 133
Querying the Product Comparison List 136
Managing Product Comparison Lists 137
Examples of Product Comparison Pages 139
Displaying a Product Comparison Table 139
Adding or Removing a Product from a Product Comparison List 141
Adding Multiple Products to a Product Comparison List 142

Removing Specific Entries from a Product Comparison List 143
Using Product Comparison Lists in a Multisite Environment 144

11 Implementing Shopping Carts 145
Understanding the ShoppingCart Component 145
Managing Shopping Carts 147
Creating and Retrieving Shopping Carts 147
Adding Items to Shopping Carts 149
Adding Shipping Information to Shopping Carts 154
Adding Payment Information to Shopping Carts 167
Repricing Shopping Carts 182
Saving Shopping Carts 183

12 Implementing an Order Approval Process 189
Displaying Orders Requiring Approval 189
Processing Approvals and Rejections 190
Displaying a History of Approved and Rejected Orders 191

13 Filtering Commerce Item Collections 193
How Product Collection Filtering Works 193
Using ATG Collection Filtering Components 194
Using InventoryFilter 195
Using ExcludeItemsInCartFilter 195
Using ProductFilter 196
Using CartSharingFilter 196
Filtering Multisite Gift and Wish Lists 196

14 Using ATG Commerce Portal Gears 201
Order Status Gear 201
Setting Up the Order Status Gear 201
Using the Order Status Gear 202
Configuring the Order Status Gear 204
Order Status Gear Implementation 206
Order Approval Gear 209
Setting Up the Order Approval Gear 209
Using the Order Approval Gear 210
Configuring the Order Approval Gear 213
Order Approval Gear Implementation 216

Appendix: ATG Commerce Servlet Beans 221
AddItemToCartServlet 226
AddBusinessProcessStage 228
ApprovalRequiredDroplet 228
ApprovedDroplet 231
AvailableShippingMethodsDroplet 233
AvailableStoreCredits 235

| | |
|---|---|
| B2BOrderLookup | 236 |
| CatalogItemLookupDroplet | 240 |
| CatalogPossibleValues | 243 |
| ClosenessQualifierDroplet | 244 |
| CollectionFilter | 246 |
| ComplexPriceDroplet | 249 |
| ConvertAbandonedOrderDroplet | 250 |
| CostCenterDroplet | 251 |
| CouponDroplet | 254 |
| CurrencyCodeDroplet | 255 |
| DisplaySkuProperties | 256 |
| ExcludeItemsInCartFilterDroplet | 257 |
| ForEachItemInCatalog | 259 |
| GetApplicablePromotions | 261 |
| GiftCertificateAmountAvailable | 263 |
| GiftitemDroplet | 264 |
| GiftlistDroplet | 265 |
| GiftShippingGroupDroplet | 266 |
| GiftShippingGroupsDroplet | 267 |
| HasBusinessProcessStage | 268 |
| InventoryDroplet | 269 |
| IsHardGoodsDroplet | 271 |
| ItemLookupDroplet | 272 |
| ItemPricingDroplet | 273 |
| MostRecentBusinessProcessStage | 274 |
| NavHistoryCollector | 275 |
| OrderLookup | 277 |
| PaymentGroupDroplet | 281 |
| PossibleValues | 285 |
| PriceDroplet | 285 |
| PriceEachItemDroplet | 286 |
| PriceItemDroplet | 288 |
| PriceRangeDroplet | 290 |
| ProductListContains | 291 |
| PromotionDroplet | 293 |
| ReanimateAbandonedOrderDroplet | 294 |
| RemoveBusinessProcessStage | 295 |
| RepriceOrder | 296 |
| SetLastUpdatedDroplet | 298 |
| ShipItemRelPrice | 299 |
| ShippableGroupsDroplet | 300 |
| ShippingDroplet | 302 |
| ShippingGroupDroplet | 303 |
| SiteIdForCatalogItem | 306 |
| UnitPriceDetailDroplet | 306 |
| ViewItemEventSender | 308 |

Index 310

# 1 Introduction

Welcome to the *ATG Commerce Guide to Setting Up a Store*. The ATG Commerce application 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 comprehensive guide covers ATG Commerce concepts for store administrators, business users, and page developers.

ATG Commerce is available in two versions. ATG Consumer Commerce is used for developing standard business-to-consumer (B2C) online stores. ATG Business Commerce is used for sites oriented toward business-to-business (B2B) uses. You will occasionally see the text "ATG Business Commerce only" or "ATG Consumer Commerce only" in this manual.

This chapter includes the following sections:

**Commerce Overview**

**Finding What You Need**

## Commerce Overview

This section introduces you to the major features of ATG Commerce:

- Product Catalog
- Purchasing and Fulfillment Services
- Targeted Promotions
- Commerce Services
- Portal Gears
- Reporting
- Multisite Integration
- Reference Applications

### Product Catalog

The product catalog is a collection of repository items (categories, products, media, etc.) that provides the organizational framework for your commerce site. ATG Commerce includes a catalog implementation based on the ATG SQL Repository, that 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. You can display different versions of your product catalog for different viewers; for example, if a corporate customer only wants its employees to order certain items from your store, you can confine them to viewing and ordering only those products. You can also configure commerce items to include variable components, such as a computer that can be purchased with different hard drive capacities.

## Purchasing and Fulfillment Services

ATG 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. You can integrate with third-party authorization and settlement tools such as Payflow Pro, CyberSource, and TAXWARE.

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.

ATG 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.

The fulfillment framework includes the following features:

- Cost Centers (ATG Business Commerce only). Cost Centers allow customers to track internal costs by designating parts of their organization as cost centers, enabling them to track costs by department and run related reports.
- Export an Order Via XML. Classes in ATG Commerce allow you to export customer orders in XML for easy integration with your other systems.
- Scheduled Orders. Your customers can create template orders from a new or existing order, 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, then stop so the company can review its needs and perhaps change the standard order.
- Order Approvals (ATG Business Commerce only). B2B applications often require that customers' orders be reviewed by authorized persons who can approve or reject them. The approval process in ATG Business Commerce 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.

- Invoicing (ATG Business Commerce only). This feature gives your customers the option of being invoiced for orders they place.
- Requisitions (ATG Business Commerce only). 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.

## Targeted Promotions

Business managers can use ATG 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 through a simple interface in the ATG Control Center as described in this guide, or through ATG Merchandising (see the *ATG Merchandising Administration Guide*).

## Commerce Services

ATG Commerce provides services for implementing a variety of merchandising 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 them 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.

You can use the ATG Control Center to manage gift list, coupon, and gift certificate repository items.

## Portal Gears

If you are an ATG Commerce *and* ATG Portal user, you can use ATG Commerce gears in the portal pages of your Commerce site to provide customers with a personalized gateway to their order information.

The Order Status gear, which is included with both ATG Consumer Commerce and ATG Business Commerce, provides customers with access to their current and historical order information. The Order Approval gear, which is included with ATG Business Commerce, displays to approvers those orders that require their approval and provides a mechanism for approving or rejecting them. For more information on these portal gears or "portlets," see the Using ATG Commerce Portal Gears chapter.

## Reporting

ATG Commerce is fully integrated with ATG Customer Intelligence, and includes a default set of reports that can provide essential information on store performance. See the *Guide to ATG Commerce Reports* for detailed information on these reports.

## Multisite Integration

ATG'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 an ATG 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 lists 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.
- 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.
- Searching—Search form handlers are site-aware and can be constrained by site.
- Reports—All ATG Commerce reports include site information. See the *Guide to ATG Commerce Reports*.

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

**Note:** If you are using B2B, some multisite features are turned off by default. Standard B2B processing assigns catalogs and price lists to individual shoppers or organizations, and does not rely on site information even in a multisite-enabled configuration. See the *ATG Commerce Programming Guide* for details.

## Reference Applications

The ATG platform includes reference applications that demonstrate how a Web site could use ATG Commerce features:

- ATG Commerce Reference Store, described in the *ATG Commerce Reference Store Overview* and related documents
- ATG Business Commerce Reference Application (Motorprise), as described in *ATG Business Commerce Reference Application Guide*
- ATG Commerce Sample Catalog, as described in the section About the ATG Commerce Sample Catalog

### *About the ATG Commerce Sample Catalog*

The commerce sample catalog is a set of sample JSPs that constitute a stripped-down but functional commerce site. As you develop your sites, you can refer to the sample catalog for simple code examples that illustrate common ATG Commerce features, such as the following:

- Using dynamic pricing and inventory
- Navigating the product catalog
- Searching the product catalog
- Adding items to a shopping cart or gift list
- Checking out orders with a single shipping group and payment group
- Managing multiple shopping carts within one person's session

Additionally, you can use the sample catalog JSPs as a starting point for your own JSP templates.

You can access the sample catalog through the ATG Business Commerce reference application. The sample pages use a B2B-style user profile and catalogs. To access the sample catalog, when you assemble your application, append the `DCSSampleCatalog` module to the list of modules:

```
MotorpriseJSP DCSSampleCatalog
```

You can use your own Commerce application rather than `MotorpriseJSP`.

**Note**: For detailed information on assembling an ATG application, see the *ATG Programming Guide*.

The `DCSSampleCatalog` module includes a `sampleCatalog` Web application that contains the sample catalog JSPs but no additional code or configuration properties. In order to access the Web application, first you need to deploy the `sampleCatalog.ear` file provided in `<ATG10dir>/DCSSampleCatalog/j2ee-apps/` on your application server, then deploy the application. Note that the sample catalog JSPs are predominantly intended as simple illustrations of common ATG Commerce features; they are not guaranteed to work in every environment or with every application.

The sample catalog JSPs are located at `<ATG10dir>/DCSSampleCatalog/j2ee-apps/sampleCatalog/web-app/`. Once the sample catalog module has been deployed, you can view the pages by pointing your browser to `http://`*`hostname:port`*`/sample_catalog/`. The port you use depends on your application server and how it is configured. For example, on JBoss the default URL is:

```
http://hostname:8080/sample_catalog/
```

See the *ATG Installation and Configuration Guide* for default port information for other application servers.

You can also access the sample catalog JSPs via the Pages and Components > J2EE Pages area of the ATG Control Center (ACC). Via the J2EE Pages task area, you can open a specific JSP in the ACC's Document Editor, and, if the sample catalog application is running, preview it in your browser.

## Finding What You Need

ATG 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. Here is a key to finding the information you need:

| **Tasks** | **Audience** | **Refer To** |
|---|---|---|
| Creating a catalog and populating it with categories, products, and SKUs; configuring the fulfillment and inventory tools provided by ATG Commerce out-the-box | Business Users | ATG Commerce Catalog Administration and Inventory and Fulfillment Administration chapters of this guide. |
| Working with the out-of the-box promotion, price list, scenarios, abandoned order, and cost center tools | Business Users | This guide. |

| | | |
|---|---|---|
| Developing a catalog and its categories, products, and SKUs in a publishing environment that uses projects to manage the tasks you perform and maintains versions of the commerce assets you edit | Business Users | *ATG Merchandising Guide for Business Users* |
| Building JSPs that use commerce servlet beans | Page Developers | Covered in several chapters in this guide. |
| Extending ATG Commerce programmatically by creating subclasses and modifying repositories | Programmers | *ATG Commerce Programming Guide* |
| Integrating ATG Commerce with Payflow Pro, CyberSource, and TAXWARE; communicating with ATG Commerce through Web Services | Programmers | *ATG Commerce Programming Guide* |
| Assembling an application that includes ATG Commerce and reference application modules | Site Administrators | *ATG Programming Guide* |
| Installing ATG Commerce databases in a production environment | Site Administrators | *ATG Commerce Programming Guide* unless users also have ATG Merchandising, in which case they should see the *ATG Merchandising Administration Guide* instead. |
| Installing database tables in support of ATG Merchandising | Site Administrators | *ATG Merchandising Administration Guide* |
| Database tables, session backup, JMS messages, and recorders | Site Administrators | *ATG Commerce Programming Guide* |
| Working with the Motorprise Business Commerce Reference Application | All | *ATG Business Commerce Reference Application Guide* |
| Working with the ATG Commerce Sample Catalog | Page Developers, Programmers | See the Reference Applications section in this chapter. |
| Viewing ATG Commerce reports | Business Users | See the *Guide to ATG Commerce Reports*. |

# 2 ATG Commerce Catalog Administration

A product catalog is composed of related items that form an organizational and navigational framework, enabling customers to locate and purchase items. A product catalog is usually built from a hierarchical tree of categories and products.

This chapter discusses how to use the ATG Control Center (ACC) to create and modify a catalog for your commerce site. Note that this chapter assumes that you are using the default ATG Commerce product catalog and the default repository editor. If the product catalog or repository editor has been customized at your sites, administering the catalog may work differently from the way this chapter describes.

Users who have ATG Merchandising and ATG Content Administration create catalogs and populate them in a content management environment instead of the ACC, so some sections in this chapter will be relevant for them and some won't. See the first two sections to learn about the commerce assets you will use and how to organize them in your catalog. See the *ATG Merchandising Guide for Business Users* for information on how to work with catalogs in ATG Merchandising.

This chapter covers the following topics:

**Organizing Your Product Catalog**

**Commerce Catalog Item Types**

**Viewing Catalogs**

**Creating Catalog Items**

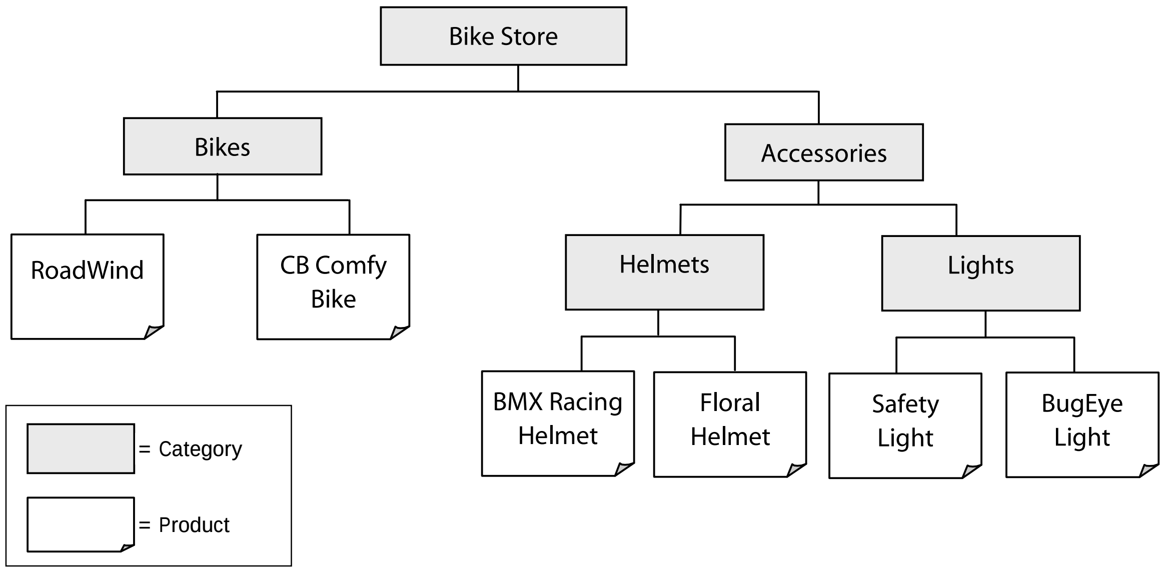
**Adding Templates and Images to the Catalog**

**Searching for Items in the ACC**

**Preventing Version Conflict**

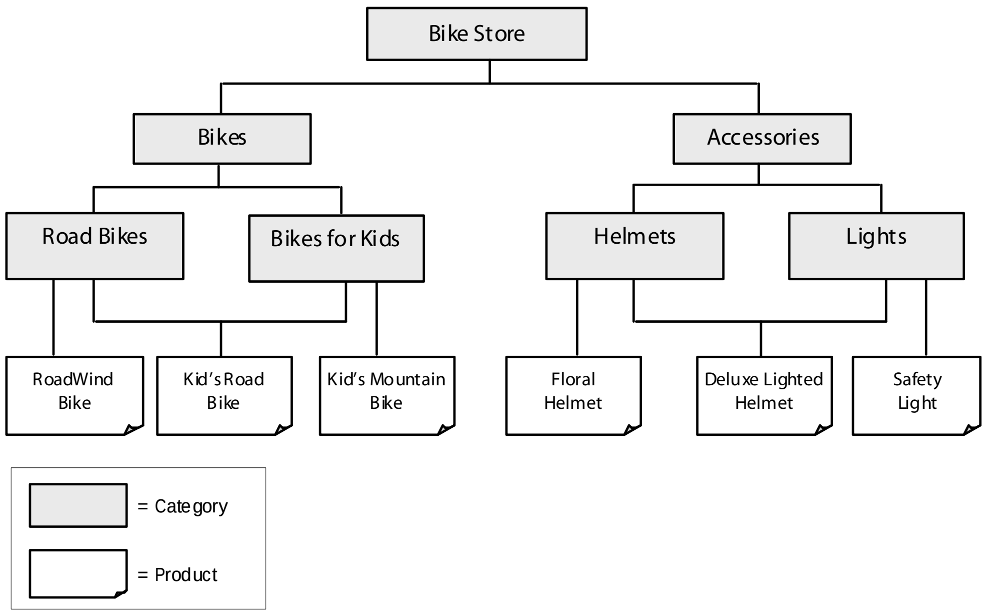
## Organizing Your Product Catalog

There are several organizational models you can use for your product catalog in ATG Commerce. For example, you arrange your items in a simple tree-like structure. In this model, each category contains products or other categories as its children, and each category or product has one parent category, as shown by the following diagram.

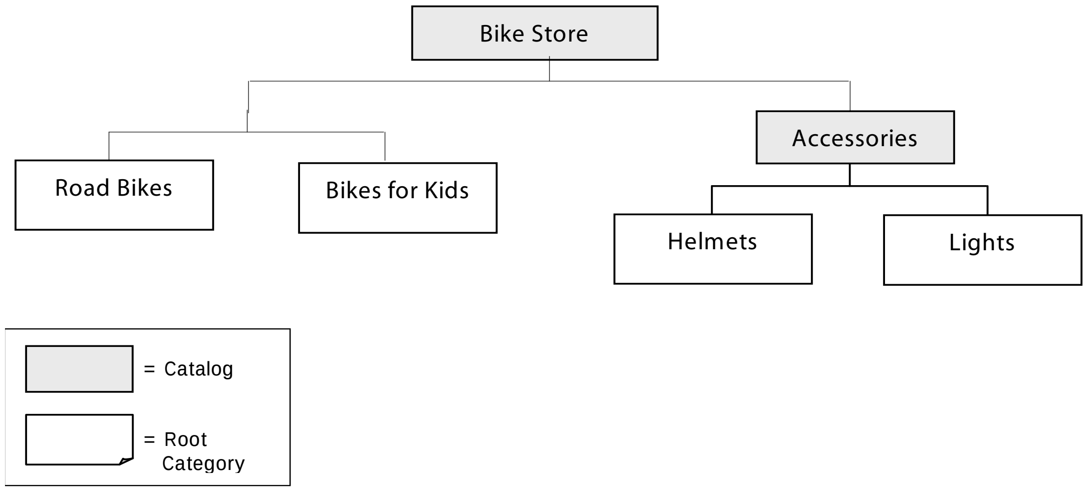


Note that there is a single navigational path to any given product. For example, to get to the BMX Racing Helmet product, the customer must select **Accessories**, then **Helmets**, and then **BMX Racing Helmet**.

Alternatively, you can offer your customers multiple navigation paths to reach a given product. The following diagram illustrates a product catalog in which products have multiple parent categories.



As a third option, you can present entirely different catalogs to different users.



When users with access to the Bike Store catalog view the root categories, they see all of the root categories in the Accessories subcatalog as well as the root categories of the Bike Store catalog itself. A user who only has access to the Accessories catalog, however, sees only the Helmets and Lights root categories.

## Commerce Catalog Item Types

In ATG Commerce, the product catalog is an ATG repository, and the elements of the catalog (such as folders, categories, products, and images) are repository items. You build the product catalog by adding new repository items and defining relationships between them.

The catalog item type represents different versions of your store for users to shop in. Categories are like store departments, and products represent the individual products for sale. SKUs represent different versions of the product, and are the actual items sold. For example, a product that represents a specific shirt might have many associated SKUs, representing different combinations of size and color. Folders are used for organizing items in the catalog.

Commerce also has media item types, which represent the JSP template pages used in the site, and images, which can be displayed along with categories, products, or SKUs.

## Viewing Catalogs

The ATG Control Center allows you to manage your product catalogs using a graphical user interface. You can use the ACC to perform catalog management tasks such as creating and modifying folders, catalogs, categories, and products, importing images, and searching for items. This chapter describes how to use the ACC to manage your product catalog. For more information about the ATG Control Center, see the online help.

## Viewing Catalogs as a Hierarchy

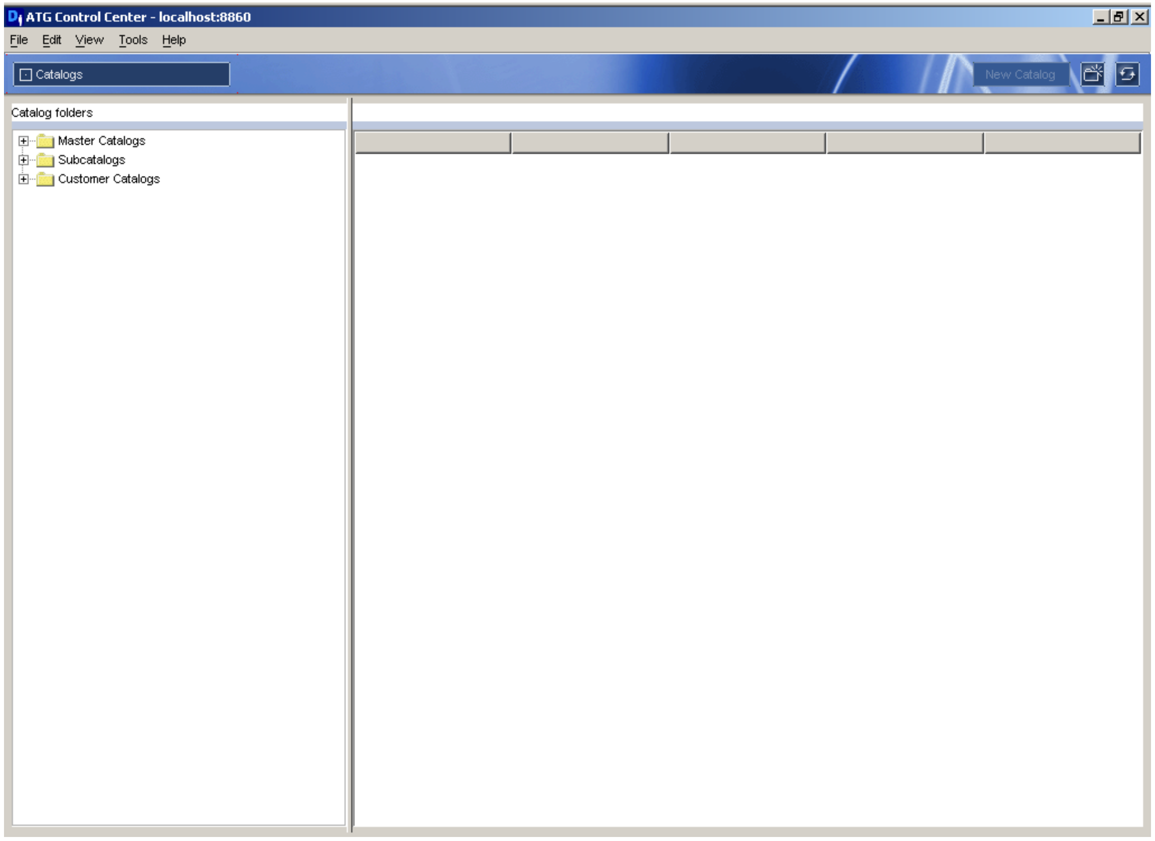
The child categories and products of each catalog determine the catalog structure. For example, the child categories of Baked Goods could be Cakes, Cookies, and Breads, and the child products of Breads could be Rye, Whole Wheat, and White. A category can have both child categories and child products. For example, Fruits might have Apples and Pears as child products, and Citrus Fruits as a child category.

Note that a category or product can be the child of more than one category. This makes the catalog more flexible, but can complicate navigation. This is especially the case if the customer accesses a category or product through a search facility rather than by traversing the catalog hierarchy; if the customer then wants to move up the hierarchy, you must determine which parent category to move to. Therefore, products and categories have a Parent category property that you can use to specify the default parent category for each item.

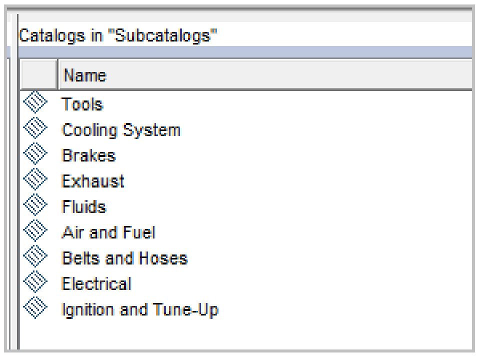
For more information about the individual properties of categories and products, see the *Using and Extending the Product Catalog* chapter in the *ATG Commerce Programming Guide*.

You can view your catalogs from the ATG Control Center. To access the product catalog:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.

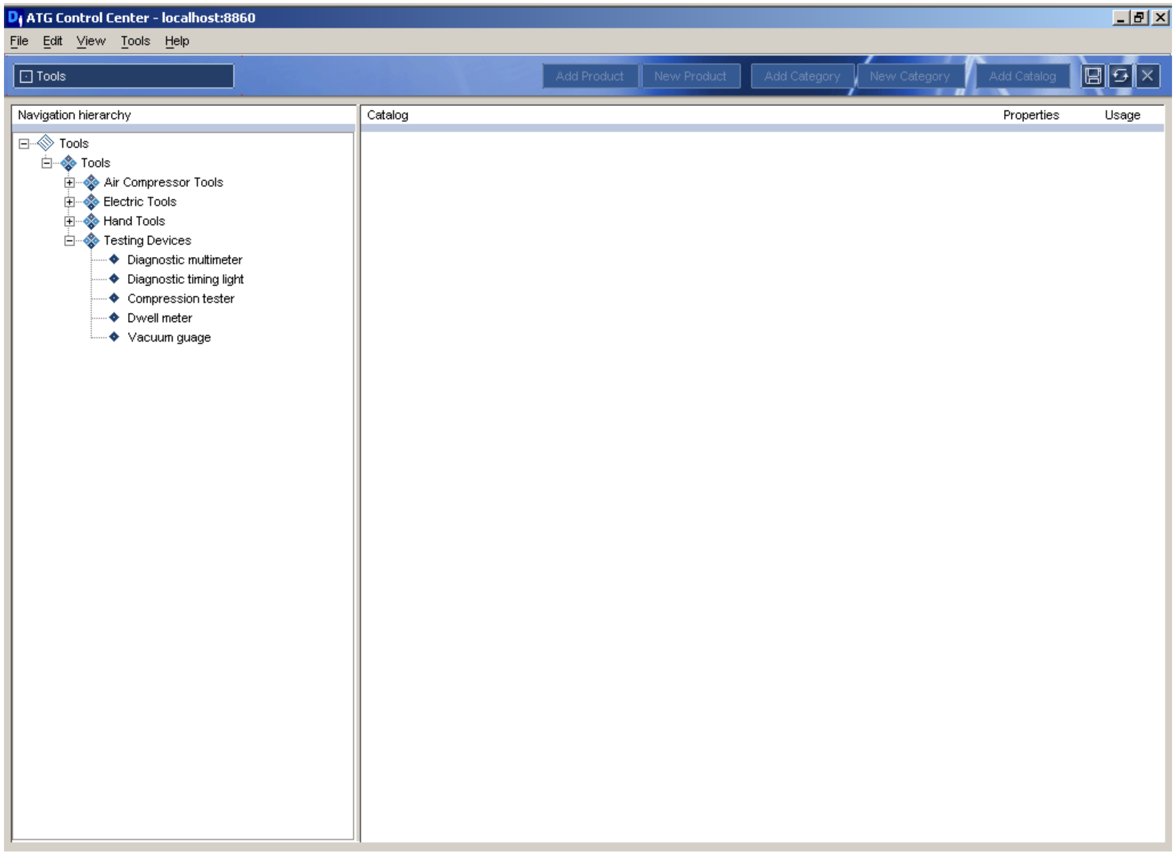


Existing catalog folders are listed in the left-hand panel. Select a folder; existing catalogs within that folder appear in the right-hand panel.


Catalogs in "Subcatalogs"

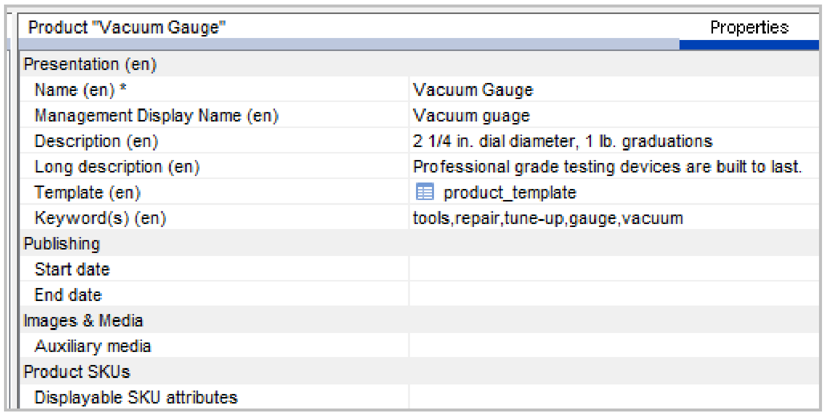
| Name |
|---|
| Tools |
| Cooling System |
| Brakes |
| Exhaust |
| Fluids |
| Air and Fuel |
| Belts and Hoses |
| Electrical |
| Ignition and Tune-Up |

2. Click the catalog name (such as Tools) to view it.



Catalogs appear as tree structures in the left-hand panel. You can expand any item to see its child items by clicking the plus sign (+).

When you select an item, the right-hand panel displays the names and current values of the selected item's properties. Depending on the item type you select, additional information may be available. For example, products have a section for associated SKUs and one for cross-selling information, while categories have only properties and associated images. The image that follows shows part of a product properties panel.



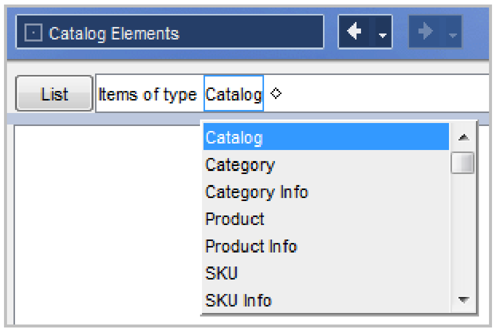
| Product "Vacuum Gauge" | Properties |
|---|---|
| Presentation (en) | |
| Name (en) * | Vacuum Gauge |
| Management Display Name (en) | Vacuum guage |
| Description (en) | 2 1/4 in. dial diameter, 1 lb. graduations |
| Long description (en) | Professional grade testing devices are built to last. |
| Template (en) | product_template |
| Keyword(s) (en) | tools,repair,tune-up,gauge,vacuum |
| Publishing | |
| Start date | |
| End date | |
| Images & Media | |
| Auxiliary media | |
| Product SKUs | |
| Displayable SKU attributes | |

Grayed-out properties are set automatically and are read-only. Also, the property names shown are their display names. To see the name used in the repository for any property, move the cursor over the display name; the property name appears as a tool tip. You must use the actual property name when referring to the property in a JSP.

## Viewing Catalogs as Lists

You can also access a nonhierarchical view of the product catalog.

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Select the item type you want to view from the Items of Type drop-down box, then click the List button. To add conditions to your query, click the diamond-shaped drop-down box.



# Creating Catalog Items

You can create the following catalog items using the ACC:

- folder—Contains any other type of item to form an organizational grouping. Folders appear in the ACC, but do not affect how users interact with your catalog.
- catalog—Holds any number of other catalogs or categories; a catalog can be the child of another catalog, or of a category.
- category—Can be either the child of one or more other categories or a root category. A root category is a starting point in the navigational structure of the catalog.
- product—A navigational end-point in the catalog. However, customers actually purchase the SKUs associated with the product, not the product itself. A product can have several associated SKUs, representing different varieties, sizes, and colors.

**Note:** If you make changes to the product catalog on your staging server, run the `AncestorGeneratorService` before copying your changes to the live version. See *Running the Catalog Maintenance System* in the *ATG Commerce Programming Guide*.

## Creating Catalog Folders

Catalog folders are an organizational tool provided in the ACC. They do not affect how customers interact with your catalog.

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the folder to which you want to add the new catalog folder.
3. Click the New Folder button.
4. Provide a name for the new folder.
5. Click OK.

## Creating Catalogs

Catalogs can include other catalogs or categories. To add a catalog as a subcatalog to another catalog, see the Adding Subcatalogs to Catalogs section of this guide.

To create a new catalog:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the folder to which you want to add the new catalog, or click the New Folder button to add a new folder for your catalog.
3. Click the New Catalog button in the upper right corner of the ACC. The New Item dialog box appears.
4. Enter a name for the new catalog, or click OK to accept the default name.
5. Click OK.

## Creating Root Categories

To create a root category:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the folder in which the catalog to which you want to add a category exists.
3. Click the catalog to which you want to add a category, so the catalog name moves to the left-hand panel.
4. Select the catalog, so the name appears highlighted.
5. To create a new category, click the New Category button. To add an existing category to the current catalog as root, click Add Category.
6. If adding a new category, fill in the Name field (required) and any optional fields, then click OK.

The category is automatically added to the `rootCategories` property of the catalog.

## Creating Child Categories

To create a child category:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the folder in which the catalog to which you want to add a category exists.
3. Click the catalog to which you want to add a category, so the catalog name moves to the left-hand panel.
4. Select the category to which you want to add the child category, so the name appears highlighted.
5. To create a new category, click the New Category button. To add an existing category to the current category, click Add Category.
6. If adding a new category, fill in the Name field (required) and any optional fields, then click OK.

The category appears as a child category of the parent category in the catalog.

## Creating Products

To create a child product:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the folder in which the catalog to which you want to add a product exists.
3. Click the catalog to which you want to add a product, so the catalog name moves to the left-hand panel.
4. Select the category to which you want to add a product, so the name appears highlighted.

5. To create a new product, click the New Product button. To add an existing product to the current category, click Add Product.
6. If adding a new product, fill in the Name field (required) and any optional fields, then click OK.

The product appears as a child product of the parent category in the catalog.

## Creating SKUs

To create a new SKU for a product in a catalog:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the product in the catalog tree.
3. From the menu at the top of the window, select Skus.
4. Click the Add Sku button.
5. In the dialog box, select Sku from the Item Type menu**.**
6. In the dialog box, click New Item.
7. Fill in the Name field (required) and any optional fields.
8. Click OK.

## Creating Configurable SKUs

While a regular SKU represents a single item, such as a size medium blue shirt, a configurable SKU represents an object that is sold as a single item, but has variable components, such as a computer system or a car. Configurable SKUs are created the same way as regular SKUs, but you must select Configurable Sku from the Item Type drop-down menu in the New Item dialog box.

To create a configurable SKU:

1. Create an item of the type Configurable Sku. Example: Computer.
2. Create as many items of the type Configurable Property as you need. Configurable properties represent an individual aspect of the configurable item, such as Hard drive or RAM in our computer example.

   To create a configurable property, go to Catalog Management > Catalog Elements. Select Configurable Property from the Items of Type drop-down list, then click the New Item button in the upper right.
3. Create as many items of the type Configurable Option as you need. Configurable options represent the individual options for a configurable property, such as 10GB SCSI hard drive from Company XYZ in our computer example.

   Configurable options can be linked to an existing SKU. If linked to an existing SKU, the price you set in the configurable option overrides the SKU price.

To create a configurable option, go to Catalog Management > Catalog Elements. Select Configurable Option from the Items of Type drop-down list, then click the New Item button in the upper right.

4. Go back to the configurable property you created and add the configurable options to it.
5. Go back to the configurable SKU and add the configurable properties to it.

## Creating SKU Bundles

SKU bundles are SKUs composed of several other SKUs. Bundles allow a group of items to be purchased as a single item, although it is treated as multiple items in order fulfillment. They differ from configurable SKUs in that the component SKUs of a bundle are always the same.

Creating a SKU bundle is a three-part process:

1. Create the individual SKUs that the SKU bundle is composed of.
2. Create SKU links from the SKUs. A SKU link is an item type that consists of an individual SKU and a quantity.
3. Create a SKU whose Bundle links property consists of one or more SKU links.

For example, suppose you want to create a SKU bundle that consists of six #2 pencils and a pencil case:

1. Create a SKU that represents a #2 pencil, and create another SKU that represents a pencil case.
2. Create a SKU link that represents six of the #2 pencil SKUs, and create another SKU link that represents one of the pencil case SKUs.
3. Create a SKU bundle by creating a SKU that consists of these two SKU links.

### *Creating SKU Links*

To create a SKU link:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Click the New Item button in the upper right. The New Item dialog box displays.
3. Select SKU Link from the Item Type drop-down menu.
4. Select the item field in the table and click the ... button. A dialog box opens for selecting a SKU.
5. Click the List button to see a list of available SKUs. Select a SKU from the list and click OK.
6. Fill in the `displayName` and `quantity` fields (required).
7. Fill in any of the optional fields you want and click OK.

### *Creating SKUs from SKU Links*

Once you have created one or more SKU links, you can create a SKU that combines the links:

1. Create a SKU, as described in Creating SKUs.
2. Select the Bundle links field and click the ... button. A dialog box opens for specifying SKU links.
3. Click Add. The New Item dialog box displays.
4. Click List to display a list of SKU links.
5. Select the SKU links from the list. You can select multiple SKU links from the list by holding down the Ctrl key while selecting items.
6. Click OK to close the New Item dialog box, then click OK to close the Bundle links window.

## Adding Subcatalogs to Catalogs

You must create the subcatalog you want to include (see Creating Catalogs) before you can add it to another catalog as a subcatalog.

To add a subcatalog to an existing catalog:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the folder in which the catalog to which you want to add a subcatalog exists.
3. Click the catalog to which you want to add a subcatalog, so the catalog name moves to the left-hand panel and its properties appear in the right-hand panel.
4. Click the Add Catalog button to see a full list of the catalogs you can add.
5. Select the catalog you want to designate as a subcatalog of your main catalog.
6. Click OK.

## Adding Catalogs to Categories

Categories can contain catalogs as well as products. To add a catalog to an existing category:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the category to which you want to add a catalog.
3. Click the Add Catalog button to see a full list of the catalogs you can add.
4. Select the catalog you want to add to the category.
5. Click OK.

## Editing Catalog Items

When you create a catalog folder, catalog, category, product, or SKU, you can specify values for its attributes in the New Item dialog box. After you create the item, you can modify these values.

**Note:** For some properties, you cannot enter the value directly. For example, some properties are Boolean values, in which ATG Commerce provides a drop-down menu where you select either True or False. If the property represents another repository item or is a collection of repository items, you cannot enter a value in the field directly. Instead, click in the field, and then click the ... button to open a dialog box where you can specify the item or items.

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the item you want to edit in the catalog tree. If editing a SKU, select the product, then select Skus from the menu at the top of the window.
3. Select Properties from the menu.

The ATG Control Center displays a table where you can edit the attribute values directly.

4. Enter a value next to the property you want to edit.
5. After editing the catalog item, select File > Save.

## Moving Items

To move a category or product from one location in the hierarchy to another, drag the item from its current location in the catalog tree to the desired location.

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the category or product in the catalog tree.
3. With the cursor positioned over the item, hold down the left mouse button.
4. Drag the item on top of the category you want to place it in.
5. Release the mouse button.
6. Select File > Save.

If you hold down the Ctrl key while dragging the item, the item becomes the child of both its original parent category and the new category you drag it to. This operation does not create a new copy of the item.

## Duplicating Items

You can create a duplicate of an item which has the same property values as the original item, but which is a completely separate repository item. To do this:

1. Right-click the item.
2. Select Duplicate from the pop-up menu.

If you duplicate a category, it appears as a child category of the original category. Products appear at the same level in the hierarchy as the original item.

The new item has the same property values as the original, including the name. Although the two items have identical properties, they are distinct repository items that are stored separately in the database. To minimize confusion, immediately rename the new item.

### Deleting Items

To delete an item from the repository:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Right-click the item in the catalog tree.
3. Select Delete from the pop-up menu. A confirmation dialog box appears.
4. Click Yes to confirm the deletion.

To remove an item from its place in the catalog structure without removing it from the repository:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Right-click the item in the catalog tree.
3. Select Delete Link from the pop-up menu.

## Adding Templates and Images to the Catalog

Templates and images are media items used by your commerce site. They are stored as repository items of type Media. A template is a JSP page used to display catalog items. An image is a graphic file used to illustrate a category, product, or SKU.

Once you have added an image or template to the catalog, you can specify it as the value of a property of one or more categories, products, or SKUs. For example, you could add a template for displaying certain products, and set the Template property of each of those products to the name of that template.

The ATG Control Center uses folders to store and organize templates and images. Folders are not part of the actual commerce site.

### Creating Image and Template Folders

To create a new folder for storing images and templates:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Beside the List Items of Type selector, choose the folder type.

3. Click the New Item button in the upper right. The New Folder dialog box displays.
4. Fill in the name field. If you want the new folder to be the child of an existing folder, use the `parentFolder` field to specify the parent folder; otherwise leave this field empty.
5. Fill optional fields as needed and click OK.

## Adding Images

Images are represented by catalog items of type Media. You can either create an item of type Media - Internal Binary by importing the actual image into the database, or create an item of type Media - External by referring to the image file's URL.

To import an image into the database:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Click the New Item button in the upper right. The New Item dialog box displays.
3. Select Media - Internal Binary from the Item Type drop-down menu.
4. Click the ... button. A file selection box opens.
5. Navigate to the image file you want to import and then click OK. The image displays at the bottom of the dialog box.
6. Fill in the name field, and use the `parentFolder` field to specify the folder in which to store the item (required).

   **Note:** The name you specify is the name for the media-internal-binary repository item and is distinct from the name of the image file you import.
7. Fill in any of the optional fields you want and click OK.

To create a reference to an external image:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Click the New Item button in the upper right. The New Item dialog box displays.
3. Select Media - External from the Item type drop-down menu.
4. Fill in the name and URL fields, and use the `parentFolder` field to specify the folder to store the item in (required).

   **Note:** The name you specify is the name for the media-external repository item, and is distinct from the name of the image file referenced by the URL.
5. Fill in any of the optional fields you want and click OK.

## Adding Templates

Like images, templates are represented by catalog items of type Media. You can either create an item of type Media -Internal Text by storing the actual template text in the database, or create an item of type Media - External by referring to the template file's URL.

To store a template in the database:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Click the New Item button in the upper right. The New Item dialog box displays.
3. Select Media - Internal Text from the Item Type drop-down menu.
4. Fill in the name field, and use the `parentFolder` field to specify the folder to store the item in (required).
5. Enter the actual template text in the large text box at the bottom of the dialog box. For example, you could enter or paste in a JSP page.
6. Fill in any of the optional fields you want and click OK.

To create a reference to an external template:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalog Elements.
2. Click the New Item button in the upper right. The New Item dialog box displays.
3. Select Media - External from the Item type drop-down menu.
4. Fill in the name and URL fields, and use the `parentFolder` field to specify the folder to store the item in (required).

   Note that the name you specify is the name for the media-external repository item, and is distinct from the name of the JSP file referenced by the URL.
5. Fill in any of the optional fields you want and click OK.

## Associating Images and Templates with Catalog Items

Once you have added your templates and images to the ACC, you can associate them with categories, products, or SKUs. The example that follows shows how to add a small image to a category or product.

To specify a small image in a catalog:

1. From the ATG Control Center main navigation menu, select Catalog Management > Catalogs.
2. Select the category or product in the catalog tree.
3. From the menu, select Images.

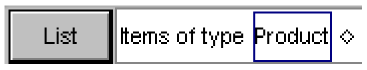
4. From the drop-down menu, select Small.
5. Click the Add Image button. The New Item dialog box appears.
6. From the Item Type drop-down menu, select Media - Internal Binary.
7. Click the List button, and then select an image from the list. Click OK.
8. Select File > Save.

## Searching for Items in the ACC

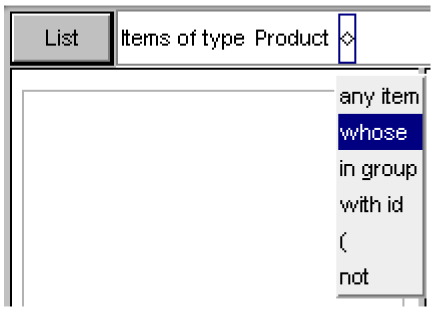
The catalog for a large commerce site may have thousands of items. If you are administering a large catalog, you may find it difficult to find specific items, or items that meet a specific set of criteria.

To help you locate items in the catalog, the ATG Control Center includes a powerful search facility that you can use to query the catalog. For example, you can construct a query that finds all products whose name includes the word “shoe.”

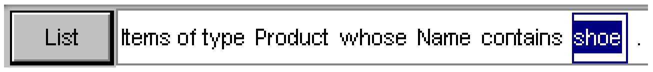
To access this search facility, select the Catalog Management > Catalog Elements window, and select the item type from the Items of Type drop-down menu. Notice the diamond to the right of the item type. For example, if you select Product, the screen should look like this:



If you click the List button now, a list of all products in the catalog displays. If you want to restrict the set of products displayed, click the diamond to display this drop-down menu:



The options on this menu help you construct complex queries. When you select an option from the menu, additional menus appear on the right for specifying the search criteria. For example, in the following query, each word to the right of “Product” is selected from a separate drop-down menu:



The choices you make on the left determine the options available as you move to the right. Even the period after “shoe” is a drop-down menu that allows you to specify additional criteria. For example, you could construct a query to find all products whose name contains “shoe” and whose creation date is before August 17, 2009.

Once you have constructed your query, click the List button, and a list of the items that match the query is displayed.

## Preventing Version Conflict

When you edit items in the product catalog, it is possible for another administrator to edit items at the same time; however, the ACC prevents multiple administrators from overwriting each other’s changes.

**Note**: The system described here is different from the versioning feature in ATG Content Administration and ATG Merchandising. For information on that feature, see the *ATG Content Administration Programming Guide*.

The ACC detects when changes are submitted that are not synchronized with the current values in the database. The catalog repository maintains a `version` property for each item. The value of that property is an integer that ATG Commerce increments automatically each time the item is modified.

For example, suppose you create a new item. The version property is 1. The next day, you open this item in the ACC and begin modifying its properties. The changes you make exist only in memory until you save the item. While you are editing the item, administrator Bob also opens the item. Because you have not saved your changes yet, the version Bob opens is still version 1.

You finish making your changes, and save the item. ATG Commerce sets the value of the version property to 2.

When Bob tries to save his changes, ATG Commerce detects that the current version in the database is not the same version that Bob has been modifying, and rejects his changes. Version 2 (which includes your changes) is then loaded into Bob’s editor.

Note that this system does not guarantee that the first person to open the item will be able to save his or her changes. If Bob had saved his changes before you tried to, your changes would have been rejected.

# 3 Inventory and Fulfillment Administration

This chapter includes information on the following ATG Commerce Administration topics:

**ATG Commerce Inventory Administration**
Describes how to use the Inventory Administration page to view and manage inventory or send notifications of changes to inventory.

**ATG Commerce Fulfillment Administration**
Describes how to use the Fulfillment Administration page to manage and process shipping groups.

## ATG Commerce Inventory Administration

Use the Inventory Administration page to view and manage inventory or to send notifications of changes to inventory.

This section contains information on the following Inventory Administration topics:

- Accessing the Inventory Administration Page
- Viewing the Inventory Display
- Updating the Inventory
- Sending Inventory Update Notifications

### Accessing the Inventory Administration Page

Follow these steps to access the Inventory administration page.

Assemble an application that includes the ATG platform, ATG Commerce, and the ATG Dynamo Server Admin. For more information, see the *ATG Programming Guide*. Then, deploy the application.

1. Access the ATG Commerce Administration Page by pointing your browser to the URL appropriate for your application server. For example, JBoss users use this URL by default:

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

See the *ATG Installation and Configuration Guide* for default port information for other application servers.

1. Click the Inventory Administration link.

   The Inventory administration page opens.

## Viewing the Inventory Display

The Inventory Administration page displays the current inventory in a table. The table lists ten items at a time in alphabetical order. Navigate through the listing by clicking on the Next and Previous links.

You can filter the list by the item name to decrease the number of items in the table. Enter letters into the two fields at the top of the screen to narrow your item search. For example, if you want to view all items with names starting with the letters R and S, enter R in the first field and S in the second field, then click the View button. The table displays items that start with R and S, listing 10 items at a time.

**Note:** Inventory filtering is case-sensitive. Capital and lowercase must match with the display name entry in the product catalog.

## Updating the Inventory

You can use the Inventory administration page to update the inventory configuration of the available commerce items. For more information on these values, see the *Inventory Framework* chapter in the *ATG Commerce Programming Guide*.

Follow these steps to update the inventory configuration:

1. Access the Inventory administration page.
2. Click the Update Inventory link at the top of the screen.
3. Enter the SKU ID of the item you want to update in the Sku id field.
4. Enter the new value for the property you want to update in the New Value field.

   Select an Inventory Manager property from the first column. For more information on the properties of the Inventory Manager, see the *Inventory Framework* chapter in the *ATG Commerce Programming Guide*.
5. Select one of the following from the second column:
   - set: sets the property to the value specified in the in the value field
   - increases: increases the current value by the value specified in the value field
   - decrease: decreases the current value by the value specified in the value field
6. Click the Update button.

   The selected inventory property is set to the specified value.

## Sending Inventory Update Notifications

You can use the Inventory Administration page to send your fulfillment system notifications of updated inventory items. Follow these steps to make the notification:

1. Access the Inventory administration page.
2. Click the Inventory Update Notification link at the top of the page.
3. Enter the SKU ID of each item with new inventory available. Separate multiple SKU IDs with spaces.
4. Click the Notify button.
5. A Java Message Service (JMS) message is sent as notification of the Inventory update.

# ATG Commerce Fulfillment Administration

Use the Fulfillment Administration page to manage and process shipping groups. This section contains information on the following Fulfillment Administration topics:

- Accessing the Fulfillment Administration Page
- Notifying Fulfillment of Order Shipment
- Reprocessing Shipping Groups
- Printing an Order

## Accessing the Fulfillment Administration Page

Follow these steps to access the Fulfillment Administration page.

1. Assemble and deploy an application that includes modules for ATG platform, ATG Commerce, fulfillment, and Dynamo Administration UI. For more information on assembling applications, see the *ATG Programming Guide*.
2. Access the ATG Commerce Administration Page by pointing your browser to the URL appropriate for your application server. For example, JBoss users use this URL by default:

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

   See the *ATG Installation and Configuration Guide* for the default URL.
3. Click the Fulfillment Administration link.

   The Fulfillment Administration page opens.

## Notifying Fulfillment of Order Shipment

On the Fulfillment Administration page, you can view all the shipping groups in the repository that are ready to be shipped. The Shippable Groups section at the top of the page includes a link that retrieves all shipping groups that are ready to be shipped. After viewing the list, you can also use this screen to notify fulfillment that a shipping group has been shipped.

Follow these steps to view a list of shipping groups and notify fulfillment that the shipping groups have been shipped:

1. Access the Fulfillment Administration page. For more information, see the Accessing the Fulfillment Administration Page section.
2. Click the "Click here" link in sentence "Click here to get a list of all the shipping groups in the repository that are ready to be shipped."

   ATG Commerce displays a list of all shipping groups with a PENDING_SHIPMENT status.
3. Enter the Order ID of the Order that contains the shipping groups.
4. Enter shipping group IDs of the shipping groups that are ready to be shipped.
5. Click the Ship button.

   A JMS message is sent notifying the fulfillment system the specified groups were shipped.

## Reprocessing Shipping Groups

Use this section to send a `ShippingGroupUpdate` message that notifies the fulfillment system that the given shipping groups have been changed and need to be reprocessed.

1. Enter the Order ID of the Order that contains the shipping groups.
2. Enter shipping group IDs of the shipping groups that are ready to be shipped.
3. Click the **Send** button.

   A JMS message is sent notifying the fulfillment system the specified groups have been changed and need to be reprocessed.

## Printing an Order

The Print an Order section of the Fulfillment Administration page allows you to print order information. This feature is useful if you need a paper record of an order's shipping groups and changes.

1. Enter the Order ID of the Order that contains the shipping groups.
2. Click the Print Order button.

   The Order is displayed in your browser window.
3. Print the order from your browser as you normally print from your browser.

# 4 Managing Price Lists

Price Lists allow you to target a specific set of prices to a specific group of customers. Price lists are managed through a single interface in the ACC. 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.

The following price list tasks can be performed by business users using the ACC.

**Viewing Existing Price Lists**

**Creating a New Price List Folder**

**Creating a New Price List**

**Changing Prices in an Existing Price List**

**Copying Prices Between Price Lists**

**Setting Bulk and Tiered Pricing**

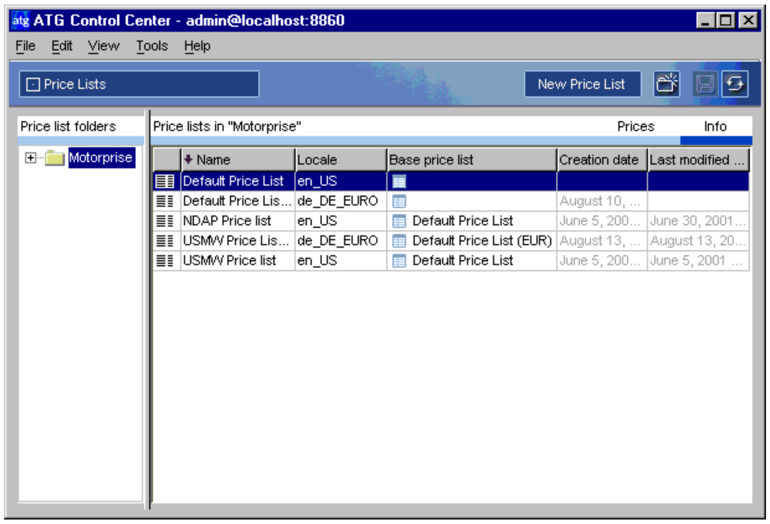
**Deleting a Price List**

**Assigning Price Lists to Users**

## Viewing Existing Price Lists

Follow these steps to view existing price lists in the ACC.

1. Select Pricing from the main ACC navigation bar.
2. Select Price Lists from the Pricing choices.
3. Select a folder from the Price List folders list and click Info on the right side of the screen. A list of the price lists in the folder displays in the main section of the ACC.


ATG Control Center - admin@localhost:8860
File Edit View Tools Help
Price Lists
New Price List
Price list folders
Motorprise
Prices
Info

Price lists in "Motorprise"

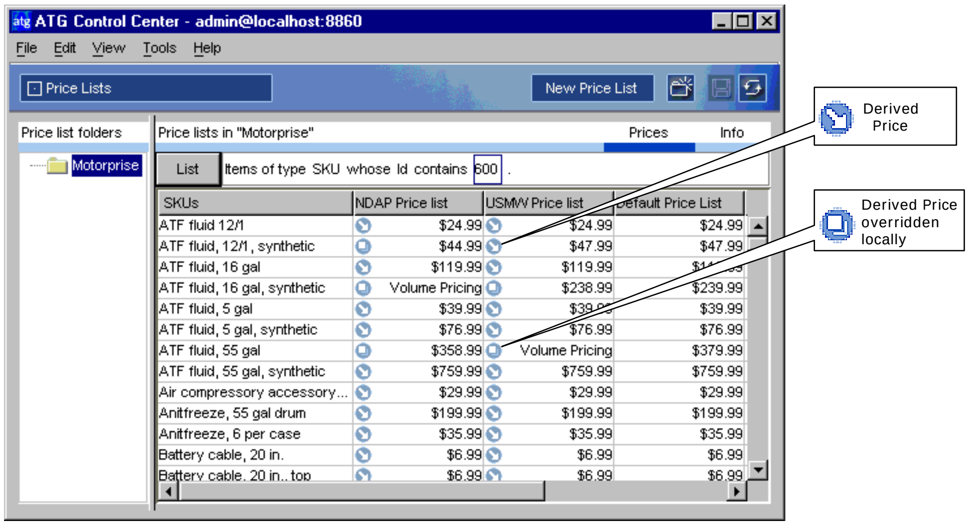
| | Name | Locale | Base price list | Creation date | Last modified ... |
|---|---|---|---|---|---|
| | Default Price List | en_US | | | |
| | Default Price Lis... | de_DE_EURO | | August 10, ... | |
| | NDAP Price list | en_US | Default Price List | June 5, 200... | June 30, 2001 ... |
| | USMW Price Lis... | de_DE_EURO | Default Price List (EUR) | August 13, ... | August 13, 20... |
| | USMW Price list | en_US | Default Price List | June 5, 200... | June 5, 2001 ... |

**Note:** You can change the Name or the locale of a price list from this screen if necessary. You can only change the locale of a base price list. You cannot change the locale of a list that has a parent list. If you change the locale of a parent, the locales of the children automatically change. You must click Refresh to see these changes.

**4.** Click Prices to view a list of the prices in the list.

**5.** Click the drop-down menu to change the search to “Find Items of type SKU.”

**6.** Click Find. A list of all SKUs in the price list and their associated prices displays.


ATG Control Center - admin@localhost:8860
File Edit View Tools Help
Price Lists
New Price List
Price list folders
Motorprise
Price lists in "Motorprise"
Prices
Info
List
Items of type SKU whose Id contains 600 .
Derived Price
Derived Price overridden locally

| SKUs | NDAP Price list | USMW Price list | Default Price List |
|---|---|---|---|
| ATF fluid 12/1 | $24.99 | $24.99 | $24.99 |
| ATF fluid, 12/1, synthetic | $44.99 | $47.99 | $47.99 |
| ATF fluid, 16 gal | $119.99 | $119.99 | [illegible] |
| ATF fluid, 16 gal, synthetic | Volume Pricing | $238.99 | $239.99 |
| ATF fluid, 5 gal | $39.99 | $39.99 | $39.99 |
| ATF fluid, 5 gal, synthetic | $76.99 | $76.99 | $76.99 |
| ATF fluid, 55 gal | $358.99 | Volume Pricing | $379.99 |
| ATF fluid, 55 gal, synthetic | $759.99 | $759.99 | $759.99 |
| Air compressory accessory... | $29.99 | $29.99 | $29.99 |
| Anitfreeze, 55 gal drum | $199.99 | $199.99 | $199.99 |
| Anitfreeze, 6 per case | $35.99 | $35.99 | $35.99 |
| Battery cable, 20 in. | $6.99 | $6.99 | $6.99 |
| Battery cable, 20 in., top | $6.99 | $6.99 | $6.99 |

## Creating a New Price List Folder

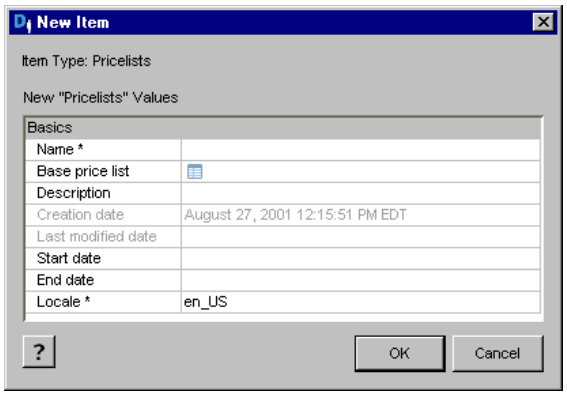
Create a new price list folder by selecting a location for the new folder in the Price List Folder tree and click the New Folder button in the toolbar.

**Note:** If you want to create a folder at the root level, you must first remove focus from the folder tree by holding down the Ctrl key and clicking the folder that is currently selected.

## Creating a New Price List

Follow these steps to create a new price list in the ACC

1. Select Pricing from the main ACC navigation bar.
2. Select Price Lists from the Pricing choices.
3. Select the folder in which to create the new price list from the Price List folders list or create a new folder. For more information on creating a new folder, see the Creating a New Price List Folder section.
4. Click the New Price List button. The New Item box displays:



Item Type: Pricelists

New "Pricelists" Values

Basics

| | |
|---|---|
| Name * | |
| Base price list | |
| Description | |
| Creation date | August 27, 2001 12:15:51 PM EDT |
| Last modified date | |
| Start date | |
| End date | |
| Locale * | en_US |

5. Enter a name and description for the price list.

6. Select a Base price list (optional). If specified, items in the price list will take their prices from this list by default.

   **Note:** If you are using price lists only, all products must be represented in the price list. If you are using a combination of price lists and SKU-based pricing, if there is no price list price, the catalog price is used as the default. See *Using Price Lists in Combination with SKU-Based Pricing* in the *ATG Commerce Programming Guide* for information on using this feature.

7. Change the locale, if necessary. Change the locale by clicking on the locale field and selecting the locale from the drop-down menu.

8. Click OK. The new price list is in the selected folder.

## Changing Prices in an Existing Price List

Follow these steps to change prices in an existing price list. This section describes how to directly edit a price in a price list. You can also change a price to use volume pricing. See the Setting Volume Pricing section for more information.

1. Open the existing price list and view prices of SKUs in the list. See Viewing Existing Price Lists for more information.

2. Double click directly on the price you want to change. If multiple price lists are visible, make sure you select from the appropriate list. You can also set volume pricing. For more information, see the Setting Volume Pricing section.

3. Enter the new price in the field and press Enter. The new price appears in the list.

   **Note:** Prices must be entered in the convention of the locale assigned to the price list. For example, if the locale is `de_DE_Euro` for European currency, then a price must be entered with a comma, such as 2,79.

## Copying Prices Between Price Lists

Follow these steps to copy prices between price lists.

**Note:** You can only copy prices between price lists with the same locale.

1. Highlight price field(s) you want to copy.
2. Right-click and select Copy from the menu.
3. Highlight the field(s) into which to paste.
4. Right-click and select Paste from the menu. The copied prices are pasted into the selected field.

   **Note:** You can only paste into fields that are in the same configuration as the fields from which you copied the prices or multiples of the same field configuration. For example, if you copy three fields in from a column, you can only paste these fields by selecting three other fields in a column or six other fields in a column. The Paste option will not appear on the right-click menu if you do not select a matching field configuration.

## Setting Bulk and Tiered Pricing

Price lists can be used to implement many pricing models. You can use volume pricing to set the price of a product based on the number of items purchased. Bulk and Tiered pricing are two styles of volume pricing.

Bulk pricing calculates the price of a product based on the total quantity ordered. In the following example, hammers are priced at $20 each when you buy 1 to 10 hammers, $17 each when you buy 11 to 20 hammers, and $15 each when you buy 21 or more hammers.

| Number of hammers purchased | Bulk Price per item | Bulk Price Total |
|---|---|---|
| 7 hammers | 7 hammers @ $20 each | $140 |
| 14 hammers | 14 hammers @ $17 each | $238 |
| 23 hammers | 23 hammers @ $15 each | $345 |

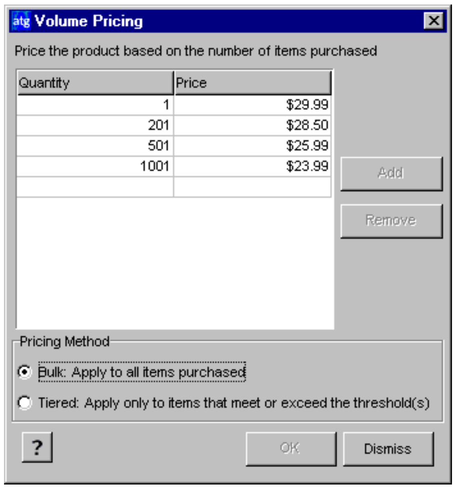
Tiered pricing calculates the price of a product using fixed quantity or weight at different pricing levels. In the following example, hammers are priced at $20 for the first 10 purchased, $17 each for the next 10 purchased, and $15 each for any additional hammers.

| Number of hammers purchased | Tiered Price per item | Tiered Price Total |
|---|---|---|
| 7 hammers | 7 hammers @ $20 each | $140 |
| 15 hammers | 10 hammers @ $20 each<br>5 hammers @ $17 each | $285 |
| 23 hammers | 10 hammers @ $20 each<br>10 hammers @ $17 each<br>3 hammers @ $15 each | $415 |

## Viewing Volume Pricing

Follow these steps to use the ACC to view the volume pricing set for a SKU in a price list.

1. View the SKU pricing for your available price lists. For more information, see the Viewing Existing Price Lists section.
2. Select a price field that is set for Volume Pricing.
3. Right-click the field and select View Volume Price from the menu. The Volume Pricing box opens.



Price the product based on the number of items purchased

| Quantity | Price |
|---|---|
| 1 | $29.99 |
| 201 | $28.50 |
| 501 | $25.99 |
| 1001 | $23.99 |

Pricing Method

[✓ Bulk: Apply to all items purchased]

Tiered: Apply only to items that meet or exceed the threshold(s)

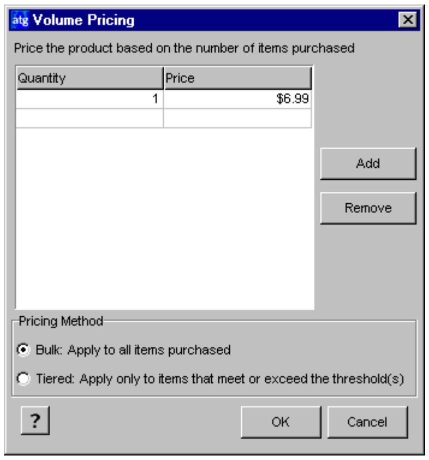
4. Click Dismiss to close the Volume Pricing box. See the Setting Volume Pricing section for information on how to edit the information in this box.

## Setting Volume Pricing

Follow these steps to use the ACC to set or edit the volume price of a SKU in a price list.

1. View the SKU pricing for your available price lists. For more information, see the Viewing Existing Price Lists in this chapter.
2. Select a price field in a price list.

3. Right-click the field and select Set/Edit Volume Price from the menu. The Volume Pricing box opens.



Price the product based on the number of items purchased

| Quantity | Price |
|---|---|
| 1 | $6.99 |

Pricing Method
[✓ Bulk: Apply to all items purchased]
Tiered: Apply only to items that meet or exceed the threshold(s)

4. Select Bulk or Tiered from the Pricing Method area at the bottom of the box.
5. Enter the quantities and prices of the pricing scheme in the columns provided. Use the Add and Remove buttons for adding and removing rows.
6. Click OK.

## Deleting a Price List

Follow these steps to delete a price list.

1. View the list of available price list by accessing price lists and clicking on the Info tab on the right side of the screen. For more information, see Viewing Existing Price Lists.
2. Select the row that lists the price list you want to delete.
3. Select File > Delete Price List. A prompt displays asking you to confirm to action.
4. Click Yes to delete the price list.

## Assigning Price Lists to Users

Follow these steps to assign price list of a user using the ACC.

1. Select People and Organization from the main ACC navigation bar.
2. Select Profile Repository from the People and Organizations choices.
3. Select “Item of type User” and click List to view a list of all users.

4. Select a user from the list. The user's information displays in the main section of the screen.
5. Find the Commerce- Contracts section, and click the "…" button beside the Price List property.
6. In the Price List dialog box, click List to display all price lists.
7. Click the name of a price list, then click OK.

# 5 Creating and Maintaining Promotions

As an optional part of the process of creating and maintaining a commerce-based Web site, you can set up promotions that you use periodically to offer discounts on specific products or groups of products. For example, you might decide to promote a range of products for Mother's Day by highlighting them with an image on your site's "Welcome!" page and offering a 10 percent discount if customers order them by a specific date. Alternatively, you might want to offer a discount to encourage visitors to register at your sites – perhaps you offer two products for the price of one to anyone who registers today.

The following are some examples of the types of discounts you can set up through promotions in ATG Commerce:

- A specific amount off a particular product
- A specific amount off a whole order
- A percentage off a particular product
- A percentage off a whole order
- A specific amount or percentage off a product based on its attributes (for example, its color)
- Free or discounted shipping

This chapter includes the following sections:

**How Promotions Work**

**Creating Promotions**

**Setting up Upselling Incentives**

**Disabling Promotions**

**Displaying Promotion Media**

**Setting Up Coupon Promotions**

**Delivering Promotions via a URL**

## How Promotions Work

You can set up promotions in the Pricing > Promotions area of the ATG Control Center. Promotions specify options that define the circumstances under which site visitors will be offered promotions. For example, you can specify the type of discount calculation to use for each promotion (fixed amount off the regular price or a percentage off the regular price) and the period of time for which it applies. You can also specify other options such as the number of times a visitor can use the same discount.

After setting up a promotion, you must set up a scenario that determines the visitors who qualify for the promotion. (For more information, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.) The scenario tells the system how to determine the people that qualify, and then it marks their visitor profiles accordingly by adding the promotion to their `activePromotions` profile attribute. You can also set up promotions that are provided to all customers automatically– for these, you do not need to set up a scenario.

When a customer visits a page that contains a product and its associated price, or performs some other action that involves requesting a price from the system, ATG Commerce checks his or her visitor profile and looks at the `activePromotions` attribute to see whether the customer currently qualifies to receive any of the promotions you have set up. It also checks to see whether you have set up any automatic promotions. ATG Commerce then uses those discounts to calculate the price of the product for the customer, and it adjusts the price accordingly.

You can inform site visitors about promotions in several ways. For example, you could set up a discount for a product without advertising it in any special way; the visitor simply sees the adjusted price when he or she displays the checkout page. You could include some text that describes the offer on, for example, the "Welcome!" page and make that text a link to the regular catalog page for the product. You could use the `GetApplicablePromotions` droplet to identify promotions that apply to particular items, and display this information on the product page. Or you could send an e-mail that describes the promotion, perhaps including a discount coupon code in the message.

## Creating Promotions

The process of setting up a promotion includes the following steps:

1. The page developer creates an HTML file that contains the text and graphics, if any, that you want to use to advertise the promotion on your Web sites. (Text and graphics for a promotion are called the "promotion media.")
2. You then do the following in the ATG Control Center:
    - Add the new promotion to the Promotions repository.

      When you add the new promotion, you also define its properties by specifying a range of information, such as the type of discount to offer, the amount of the discount, and the location of the file that contains the media for it. See Adding a New Promotion.
    - Create a discount rule for the promotion.

As part of adding the new promotion, you create the discount rule that defines the product or products associated with the promotion and the circumstances under which it is applied, for example, “If a visitor buys any mountain bike, offer the specified discount on helmets”. See Creating a Discount Rule.

- Create and enable a scenario that defines the people to whom you want the promotion to apply. See Specifying the People Who Receive the Promotion.

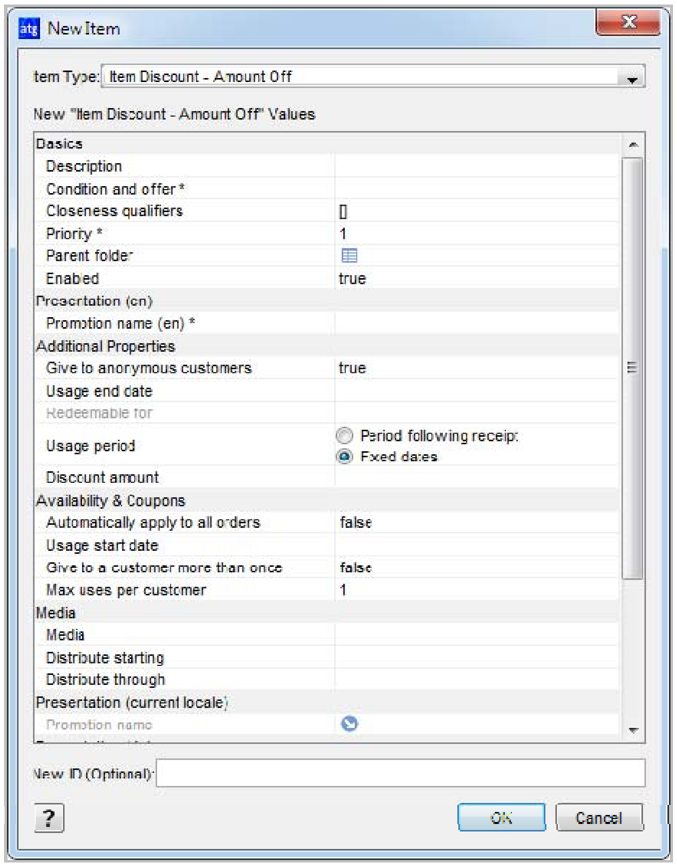
## Adding a New Promotion

Promotions are stored in the Promotions repository, which is one of the default ATG Commerce repositories. When you create a new promotion, you must add it to the Promotions repository by doing the following:

1. Select Pricing from the main ATG Control Center (ACC) menu.
2. Select Promotions from the submenu.

   The Promotions screen is displayed. You can click the List button to display all existing promotions. You can also narrow the list of displayed promotions by using the Item Type drop-down list to select a specific type of promotion to list. For example, you can list all the Item Discount promotions or, to narrow the list even further, only the Item Discount – Percent Off promotions.

   Note that the type of promotions listed on the Promotions screen directly affects the type of promotion that you can create using the ACC menu commands and buttons. For example, if only the Item Discount –Percent Off promotions are listed, and you select File > New Item from the menu, you can only create a promotion of the same type. In contrast, if promotions of all types are listed on the screen, and you select File > New Item from the menu, you can create a promotion of any type.
3. Select File > New Item. The New Item dialog box appears:



Item Type: Item Discount - Amount Off

New "Item Discount - Amount Off" Values

| | |
|---|---|
| Basics | |
| Description | |
| Condition and offer * | |
| Closeness qualifiers | [] |
| Priority * | 1 |
| Parent folder | |
| Enabled | true |
| Presentation (en) | |
| Promotion name (en) * | |
| Additional Properties | |
| Give to anonymous customers | true |
| Usage end date | |
| Redeemable for | |
| Usage period | Period following receipt [✓ Fixed dates] |
| Discount amount | |
| Availability & Coupons | |
| Automatically apply to all orders | false |
| Usage start date | |
| Give to a customer more than once | false |
| Max uses per customer | 1 |
| Media | |
| Media | |
| Distribute starting | |
| Distribute through | |
| Presentation (current locale) | |
| Promotion name | |

New ID (Optional):

4. From the Item Type drop-down list, select the type of promotion you want to create. Default choices are as follows:

| Type of promotion | Use for the following type of discount |
|---|---|
| Item discount – Amount off | A specific amount off the regular price of a product. For example, you could offer any UltraLight T-Bike for $100 off the regular price. |
| Item discount – Fixed price | A specific product for a fixed price. For example, you could offer any UltraLight T-Bike for $500. Another example would be a “buy one item, get a special price on another” discount; for example you could offer a price of $5 for any helmet to any customer who buys a bike. |
| Item discount – Percent off | A specific percentage off the regular price of a product. For example, you could offer any UltraLight T-Bike for 10% off the regular price. |

| Type of promotion | Use for the following type of discount |
|---|---|
| Order discount – Amount off | A specific amount off the total order. For example, you could offer a $50 coupon ("Use this coupon by August 31st and get $50 dollars off your total order."). Another example would be a discount of $20 on any order over $100. |
| Order discount – Fixed price | The customer pays a fixed price for his or her order. For example, you could offer any product for $50 only. |
| Order discount – Percent off | A specific percentage off the total order. For example, you could offer a 10% discount off the total cost of an order. |
| Shipping discount – Amount off | A specific amount off the price of shipping. For example, you could offer $5 off the regular shipping charges for any order over $200. |
| Shipping discount – Fixed price | Shipping for a specific amount. For example, you could offer free shipping to customers who buy any bike today. |
| Shipping discount – Percent off | A specific percentage off the price of shipping. For example, you could offer a 50% discount on your regular shipping charges to anyone who buys any three items. |

The options described above are provided with the system. However, the application developers working on your Web sites can add others if needed. For information on how to do this, see the *Creating Promotions* section of the *Using and Extending Pricing Services* chapter in the *ATG Commerce Programming Guide*.

**5.** After you choose the type of promotion you want to add, the system displays a form at the bottom of the dialog box. Use it to specify the properties for this new promotion.

The following table describes the properties:

| **Property** | **Purpose** |
|---|---|
| Automatically apply to all orders | Use this property to identify the promotion as an "automatic promotion." Automatic promotions are given to all visitors whereas non-automatic promotions are given to a restricted grouping of visitors who meet the criteria specified by additional promotion property values.<br><br>If you set this property to `true`, all visitors who meet the criteria in the promotion's discount rule (if specified) qualify for the promotion. That means the promotion is offered an **unlimited** number of times, to **all** visitors (including anonymous visitors), for use on an **unlimited** number of orders, for as long as the promotion is enabled.<br><br>If you set this property to `false`, you can specify values to the following properties in order to restrict who will receive the promotion:<br><br>- Give to a customer more than once<br>- Give to anonymous customers<br>- Number of uses allowed per customer<br>- Usage period<br>- Redeemable for<br>- Usage start date<br>- Usage end date<br><br>Also, if you set this property to `false`, you must create a scenario that offers the promotion to the appropriate visitors. For more information on how to define who qualifies for the promotion, see Specifying the People Who Receive the Promotion. |
| Closeness qualifiers | A list of products that can be used as part of closeness qualifier messaging. For example, if you want to promote a line of luggage, you may add them to the Closeness qualifier, then use them in product pages, emails, or other communications when a customer is close to qualifying for a promotion. See Setting up Upselling Incentives |
| Condition and offer | This field allows you to set up the discount rule that controls the circumstances under which the system gives the promotion to a visitor (for example, a specific product to which the discount is applied or the number of items the customer must buy to receive the discount). For information on how to use this field, see Creating a Discount Rule. |
| Creation date | The time and date that you added this new promotion to the Promotions repository. The system fills in this field automatically. |

| Property | Purpose |
|---|---|
| Description | Type a short display name for this promotion. The system can use this name to identify the promotion to customers on, for example, a checkout page or order form. |
| Discount price/amount/ percentage | Specify the amount of the discount. The type of value you enter depends on the promotion; for example, for a discount in which you are offering a percentage of the usual price as the reduction, you specify the percentage here. |
| Discount type | The item type for this promotion. The value is the type of new promotion that you chose to create in step 4 above. The system fills in this field automatically. |
| Distribute starting | Specify the date and time you want to start displaying this promotion on your sites.<br><br>Contrast this property with the Usage Start Date property below. |
| Distribute through | Specify the date and time you want to stop displaying this promotion on your sites.<br><br>Contrast this property with the Usage End Date property below. |
| Enabled | Specify `True` when you are ready to activate this promotion. If you specify `True`, the promotion does not start displaying on your site until the Distribute Starting date and time, and it stops displaying on your sites on the Distribute Through date and time.<br><br>If you specify `False`, the promotion **never** takes effect regardless of the Distribute Starting and Distribute Through dates and times. |
| Enable promotion upsells | If true, the Closeness qualifiers you specify can be used as part of promotion upsells. |

| Property | Purpose |
|---|---|
| Give to a customer more than once | Tells the system whether to deliver this promotion (for example, by displaying it on a page) to each visitor only once. If you specify `False`, the system delivers the promotion only once. If you specify `True`, the system delivers the promotion every time the site visitor performs an action that qualifies him or her to receive it -- for example, he or she visits the page on which you have displayed the promotion.<br><br>**Note 1:** If you specify `True`, then a visitor may have multiple copies of the promotion. Consequently, a single order may be discounted by multiple copies of the promotion.<br><br>**Note 2:** The system ignores this property if the Automatically Apply to All Orders property is set to `True`.<br><br>Contrast the use of this property with the Max Uses Per Customer property. |
| Give to anonymous customers | Tells the system whether to apply this discount to only visitors who have registered or to all visitors.<br><br>If this property is `False` and the Automatically Apply to All Orders property is `False`, then only registered visitors who qualify can use the promotion. If this property is `True` and the Automatically Apply to All Orders property is `False`, then anonymous visitors and registered visitors who qualify can use the promotion.<br><br>**Note:** The system ignores this property if the Automatically Apply to All Orders property is set to `True`. |
| Max uses per customer | Determines the number of **orders** for a given customer to which the promotion can be applied. If you specify `1`, for example, the customer can use the promotion on only one order. If you specify `infinite`, the customer can use the promotion on an unlimited number of orders during the period for which the promotion is usable.<br><br>By default, the values you can choose for this property are `infinite`, `1`, `2`, `3`, `4`, `5`, `10`, `25`, `50`, and `100`. However, your application developers can alter these choices to suit the needs of your Web sites.<br><br>**Note 1:** 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 One Use Per Order property to `True`. (See below.)<br><br>**Note 2:** The system ignores this property if the Automatically Apply to All Orders property is set to `True`. |

| Property | Purpose |
|---|---|
| Media | Specify the presentation media (for example, text or an image) you want to associate with this promotion. The associated Distribute starting and Distribute through fields allow you to set date limits on media usage. |
| 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 you specify `True`, the system permits the shipping promotion to be applied to an order only once. If you specify `False`, the system permits the shipping promotion to be applied to an order for the number of times for which it meets the criteria of the discount rule. |
| Parent folder | Specify a parent folder for the promotion, if desired. |
| Priority | If a visitor qualifies for more than one promotion, the system applies them cumulatively, calculating the new price for the order by combining the different discounts that the promotions contain. The order in which the system combines the discounts is vital any time more than one promotion can be applied. For example, you give two discounts to the same customer, one (Promotion A) offering $10 off any product and another (Promotion B) offering 50% off any helmet. Suppose the customer orders a helmet that usually costs $30. If the system applies the promotions in the order A-B, the final price of the helmet is $10. If the order is B-A, the final price is $5.<br><br>Use this property to define the order in which the discounts are applied. Enter `1` to have the system calculate this discount first, `2` for second, and so on.<br><br>If you give the same number to more than one promotion, the system calculates randomly the discounts for those promotions but preserves any order between those and other promotions with different numbers.<br><br>**Note:** To use this property, you must turn off the `filterForTargetDiscountedByAnyDiscountId` property in `/atg/commerce/pricing/QualifierService.` This property filters out items that have already received a discount, preventing them from being discounted again, and must be `false` in order for combined discounts to work. |
| Promotion name | Specify a name that you can use to identify this promotion elsewhere in the ATG Control Center (for example, in the Give Promotions element in scenarios). |

| Property | Purpose |
|---|---|
| Redeemable for | Used in conjunction with the Usage Period property when its value is “Period following receipt.”<br><br>Specify the number of minutes for which this promotion is valid. The start date and time of a promotion of this type is set when the visitor receives the promotion; that is, the promotion is added to the list of promotions in the visitor’s `activePromotions` attribute in his or her profile. The end date and time for the promotion is then determined by the start date/time and the number of minutes you specify in this property.<br><br>**Note:** The system ignores this property if the Automatically Apply to All Orders property is set to `True`. |
| Sites | If you are using ATG’s multisite feature, select sites on which the promotion can be used. If nothing is specified, the promotion is valid on all sites. |
| Site groups | Select site groups on which the promotion can be used. Note that if you select both a site group and a site within that group, the site does not act as a restriction; all sites in that group are eligible for the promotion. |
| Usage period | Defines the duration of time for which this discount or coupon is valid.<br><br>You can specify a duration of time that follows the visitor’s receipt of the promotion; in this case, select “Period following receipt” as the value. Or, you can specify a duration of time with a fixed start date and end date; in this case, select “Fixed dates” as the value.<br><br>**Note:** The system ignores this property if the promotion is automatic as indicated by the Automatically Apply to All Orders property. Automatic promotions are usable for the duration of the distribution period. |
| Usage start date | Used in conjunction with the Usage Period property when its value is “Fixed dates.”<br><br>Defines the beginning of the period for which this discount or coupon is valid. Specify the date and time that visitors can either start purchasing a product with this discount or start using this coupon.<br><br>**Note:** The system ignores this property if the Automatically Apply to All Orders property is set to `True`. |

| Property | Purpose |
|---|---|
| Usage end date | Used in conjunction with the Usage Period property when its value is "Fixed dates."<br><br>Defines the end of the period for which this discount or coupon is valid. Specify the last date and time that visitors can either purchase a product with this discount or use this coupon.<br><br>**Note:** The system ignores this property if the Automatically Apply to All Orders property is set to `True`. |
| Version | The system uses this property to protect against data corruption that might be caused if two ATG Control Center users attempt to edit this repository item at the same time. The system updates the value automatically. |

The most important properties of a promotion are the Discount price/amount/percentage and the Discount rule. These properties combine to define the actual amount of the discount and the products to which it applies. See the next section for more information on how these properties work together.

**Note:** Several factors can affect whether a new promotion is available to a user at the desired time or at all. These factors include when the new promotion was added, when the user's session was created, and the schedule for updating automatic promotions. Your developers can take steps to ensure that site visitors receive a promotion at the time you intend. For more information, refer to the *ATG Commerce Programming Guide*.

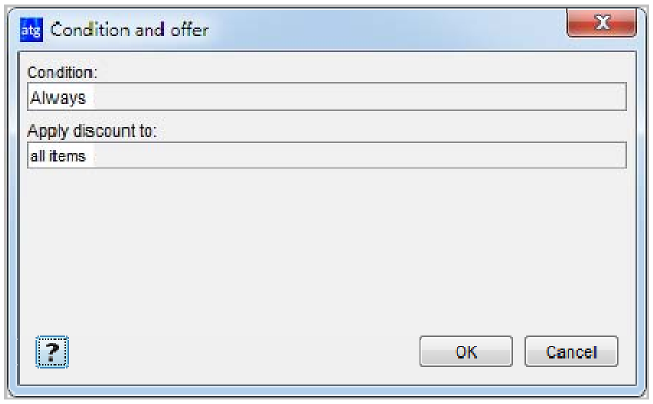
6. When you are finished defining the properties of the promotion, click the OK button to save the promotion.

## Creating a Discount Rule

As part of the process of defining the properties of a new promotion, you create a discount rule, which specifies the circumstances under which to apply the promotion.

To create a discount rule, click the empty box in the Condition and offer property for this promotion in order to activate the box; then click the corresponding button that displays.

When you click the button, the Condition and offer dialog box appears, as follows:



Condition:
Always
Apply discount to:
all items

Discount rules contain two parts, as represented by the two fields in the dialog box:

- A Condition statement, which defines when the discount applies, for example "`Always`" or "`When order contains at least one (product named Shatterproof Helmet)`."
- An Apply Discount To statement, which specifies the target items to which you want to apply the discount, for example, "`Apply discount to every (product in category named BMX Bikes)`."

The default Condition is "Always." The default Apply Discount To statement depends on the discount type. In the example shown above, the discount type is Item Discount - Percent Off, so the default is "Apply discount to all items." If you leave this rule as it is, the system applies the discount to all items in the product catalog. The exact amount and type of the discount are specified by the Discount Type and Discount Price/Amount/Percentage/Multiplier properties.

For example, you might have a promotion that offers a 10% discount on any products to anyone who registers at the site. This promotion has its Discount Type and Discount Percentage properties set as follows:

- Discount Type: Item Discount - Percent Off
- Discount Percentage: 10

The rule for this promotion is the default rule shown above: "Always discount all items." In combination with the other properties, the rule tells the system to give a 10% discount on all items. (For information on how to specify the people to whom the promotion applies, see Specifying the People Who Receive the Promotion.)

In most cases, however, you probably want to change the rule so that you narrow the set of circumstances under which the discount is applied. To change the rule, click its parts so that menus appear, and then select the items that define the rule as required.

If you use "and" in the "Apply Discount to" statement of a promotion, all of the conditions in the statement must be true for the discount to be applied. The use of "and" in the following example would cause problems as described below:

```
    Apply discount to: up to 1 (product named Blue Shirt and product named
    Green Shorts)
```

The discount is **not** applied to all products named “blue shirt” and all products named “green shorts.” The discount is only applied to items named both “blue shirt” and “green shorts.” Therefore, this discount target would never apply a discount because products only have one name.

The rest of this section describes more examples of discount rules.

### *Example 1: Discounting Items from a Specific Category*

```
Condition:
Always
Apply discount to:
every (product in category named BMX Bikes)
```

Key properties in this promotion are as follows:

- Discount Type: Item Discount - Percent Off
- Discount Percentage: 25

The rule tells the system to apply a 25% discount to all products in the category called “BMX Bikes.”

### *Example 2: Discounting a Given Amount from a Total Order*

```
Condition:
Always
Apply discount to:
Order Total
```

Key properties in this promotion are as follows:

- Discount Type: Order Discount - Percent Off
- Discount Percentage: 20

The rule tells the system to apply a 20% discount to the customer’s total order.

### *Example 3: Offering a Promotion on Shipping*

```
Condition:
When Order's priceInfo's amount is greater than 399.99
Apply discount to:
Shipping Group
```

Key properties in this promotion are as follows:

- Discount Type: Shipping Discount - Fixed Price
- Discount Price: 0

The rule tells the system to charge a fixed shipping price of $0 (in other words, to provide free shipping) to any orders over $399.99. Note that, in this example, the first parameter of the Condition statement is "When" rather than "Always."

### *Example 4: Offering a Discount on One Product if a Customer Buys A Different Product*

```
Condition:
When order contains at least 1 (product in category named Whole Bikes)
Apply discount to:
up to 1 (product in category named Helmets)
```

Key properties in this promotion are as follows:

- Discount Type: Percent Off
- Discount Percentage: 40

The rule tells the system to offer a 40% discount on any helmet to customers who buy any bike.

### *Example 5: Offering a 2-for-1 Discount*

```
Condition:
When Order contains at least 3 (product in category named Lights)
Apply discount to:
Up to 1 (product in category named Lights)
```

Key properties in this promotion are as follows:

- Discount Type: Item Discount - Fixed Price
- Discount Price: 0

The rule tells the system to give one free light to anyone who orders at least three lights.

**Note:** There are various ways to set up this type of promotion. In this example, a customer must order three items in order to get one for free; to make sure that the customer does in fact order three items, you could add a note explaining the process to the order form or to the promotion media itself. In addition, this example uses the structure "at least three," which means that a customer would also receive one free light if he ordered, for example, five. For examples of other ways to set up this type of promotion, please refer to the *ATG Commerce Programming Guide*.

## Specifying the People Who Receive the Promotion

You use a scenario to define the people who receive the promotion. (For detailed information on scenarios, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.) The scenario tells the system to watch for specific visitor actions or behavior and apply the promotion to those visitors. For example, you decide that you want to offer a special promotion called "New Members" to people who have just registered; the promotion gives them 10% off their first order. You create the promotion as described in this chapter. Then you set up a scenario that tells the system to do the following:

- Watch for site visitors to register.

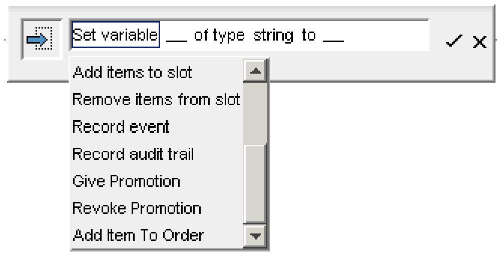
- For those visitors, add the “New Members” promotion to the `activePromotions` attribute in their profiles.

When those visitors order a product, the system checks their profile, sees the “New Members” promotion in the `activePromotions` attribute, and deducts 10% from the total price of the order.

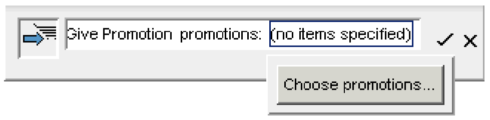
**Note:** If, when you set up the promotion, you set the Automatically Apply to All Orders attribute to `True`, you do not need to create a scenario. The system automatically makes this promotion available to all site visitors and checks to see if any such promotions exist whenever a site visitor performs an action that involves a price request.

To create a scenario for a promotion, do the following:

1. Create a new empty scenario (or a new segment in an existing scenario, if you have one that is appropriate). For more information, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.
2. Add elements that define any visitor behavior or profile attributes that you want to use to trigger the promotion. For example, if you want to give the promotion to people in a particular profile group, add a Condition element that specifies the profile group.
3. Add an Action element that gives the promotion to the specified visitors. To do this, select Action to display the a list of actions that appears as follows:

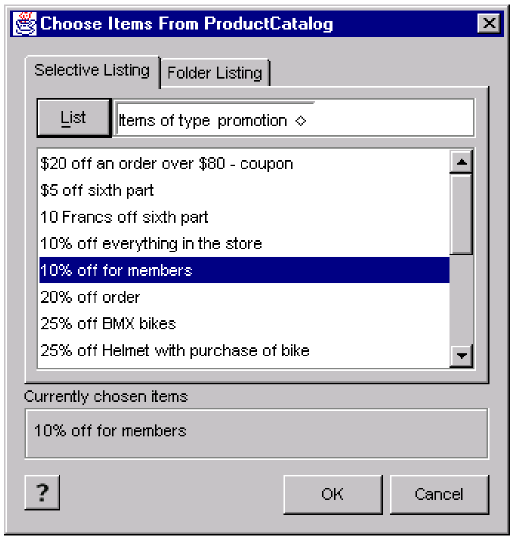


4. Select Give Promotion from the list of actions.
5. Click No Items Specified and then click Choose Promotions, as follows:



   The Choose Items From Product Catalog dialog box appears.

6. Click List to display all the promotions you have previously set up in your Promotions repository.



Items of type promotion

- $20 off an order over $80 - coupon
- $5 off sixth part
- 10 Francs off sixth part
- 10% off everything in the store
- 10% off for members
- 20% off order
- 25% off BMX bikes
- 25% off Helmet with purchase of bike

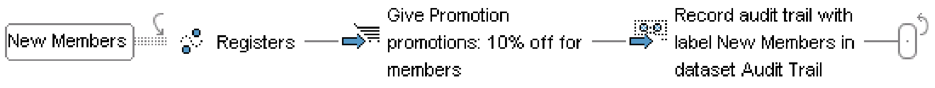
Currently chosen items

10% off for members

7. Select the promotion you want to include in this scenario and then click OK. You can include more than one promotion by holding down the Ctrl key and clicking the promotions you want.
8. Click the checkmark at the end of the element to add it to the scenario.
9. Enable the scenario and then copy it to the production server. For more information, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.

You can add other elements to this scenario if needed. For example, you can add more promotions by adding other Give Promotion elements. Or you can include different elements that extend the scope of the scenario; for example, you could add a Send Email element that sends a message giving information about the promotion to the people to whom it now applies. You could also add an Audit Trail element so that you can keep track of the promotions that you offer to site visitors. (For more information on audit trails, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.)

The following shows a simple scenario that makes a promotion called "10% off order" available to anyone who registers to become a member:


New Members
Registers
Give Promotion promotions: 10% off for members
Record audit trail with label New Members in dataset Audit Trail

In this scenario, the system watches for any site visitor to register. Then it adds the promotion called "10% off for members" to the `activePromotions` attribute in the profiles of the people who just registered. Finally, this scenario includes an Audit Trail element that records the Give Promotion action in the predefined Audit Trail dataset.

**Note:** You can also deliver a promotion to site visitors via a URL on a JSP. Your page or application developer can embed the ID of the promotion you want to offer in a URL. When a site visitor clicks that URL, the promotion is added to the visitor's `activePromotions` Profile property. For more information, see Delivering Promotions via a URL.

### *Withdrawing Promotions*

You can limit the duration of promotional campaigns in various ways; ideally, you include a Time element in the promotion scenario that limits the period during which the system can apply the discount.

You can also use a scenario Action element called Revoke Promotion to withdraw a promotion for specific visitors under certain circumstances. For example, you might set up a complex series of promotions that, in rare cases, would allow a visitor to receive a double discount on a particular item, and you might decide that this behavior is undesirable. To correct it, you could set up a Revoke Promotion element that would cancel one of the promotions if necessary.

The Revoke Promotion element works the same way as the Give Promotion element. You set up a scenario segment that defines the visitor behavior or the circumstances that will trigger the withdrawal of the promotion, and then you add the Revoke Promotion element, specifying the name of the promotion to withdraw.

## Creating Closeness Qualifiers

Each promotion can have any number of Closeness Qualifiers. Consider a “free shipping for orders over $2500” promotion. One Closeness Qualifier may display when an order equals $2400 and users have a bike of brand X in their cart. This upsell message might say “Spend another $100 to qualify for free shipping. Check out our brand X windbreakers” assuming that brand X windbreakers all cost $100 or more. Another Closeness Qualifier for the same promotion might be targeted to users with orders worth $2000. This Closeness Qualifier may say “Shipping costs for the products in your shopping cart is w. Spend another $500 and shipping is free.”

When you create a Closeness Qualifier, you describe the circumstances under which it applies by creating a PMDL rule. Creating this rule is similar to creating a promotion discount rule. You can also specify a media item, such as a graphic that entices users to purchase some product in order to receive a promotion. Such products are designated in the Upsell Action, which is associated to the Closeness Qualifier. Each Closeness Qualifier also has a priority.

Because a user could qualify for more than one Closeness Qualifiers for a given promotion, each Closeness Qualifier is given a priority setting. For a given promotion, the ATG platform evaluates the highest priority Closeness Qualifier to see if it is appropriate for the user and progresses through the list of Closeness Qualifiers until it finds one that is. When one is found, the Closeness Qualifier is assigned to the user.

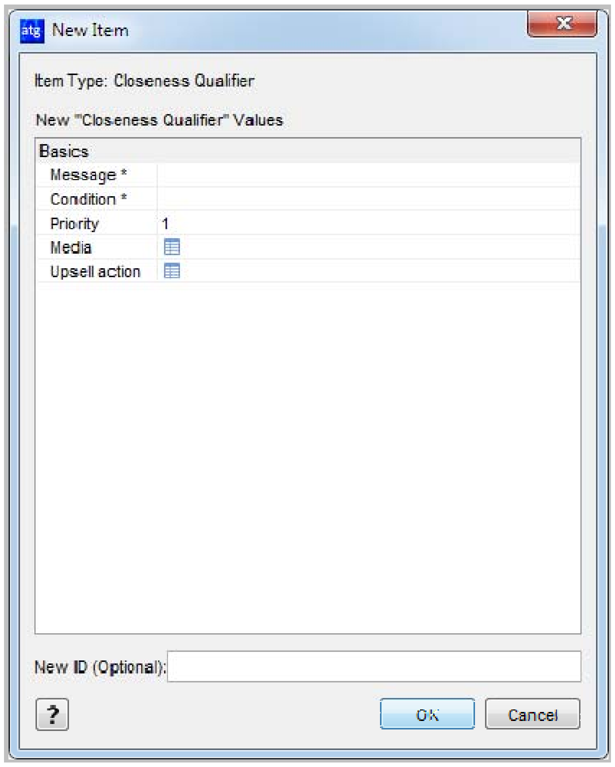
In general, users can receive one Closeness Qualifier per promotion at a given time. Since a promotion is likely to have multiple Closeness Qualifiers, assigning priorities ensures that users who qualify for two Closeness Qualifiers obtain the correct one. For example, if you have two Closeness Qualifiers, you would prioritize them as orders that are at least $100 (priority 1), and orders that are at least $50 (priority 2). That way, a user with an order of $100, who qualifies for both Closeness Qualifiers, will only be provided with the one closest to his or her total.

At times, you may want a user to be eligible for more than one Closeness Qualifier for a promotion. As demonstrated above, some Closeness Qualifiers may be based on certain products in a user’s shopping cart. You may have two Closeness Qualifiers, each of which aims to upsell a specific brand when a product of that brand is in the user's shopping cart. If both brands are represented by products in the user's shopping cart, why shouldn't the user see both Closeness Qualifiers?

You can give multiple Closeness Qualifiers in a given promotion the same priority number so that a user is able to receive both of them. When the ATG platform finds one Closeness Qualifier that is fitting for a user, it first evaluates all other Closeness Qualifiers with the same priority before ending the operation. A user is granted all Closeness Qualifiers with the same priority that apply to him or her. Other Closeness Qualifiers on that promotion with lower priority are skipped.

Create a Closeness Qualifier as follows:

1. In the Catalog Management > Catalog Elements task area, display items of type Closeness Qualifier.
2. Select File > New Item to open the New Item dialog box.



3. In the Message field, provide a name to the Closeness Qualifier that uniquely identifies it.
4. In the Condition field, specify the rule that controls the circumstances under which the ATG platform provides the Closeness Qualifier to the user. Create the rule in the same way you created the discount rule as described in Creating a Discount Rule. Once you are finished, click OK.

5. In the Priority field, specify a priority for the Closeness Qualifier that determines when, in the context of all Closeness Qualifiers for that promotion, this Closeness Qualifier should be executed. The highest priority is 1.
6. Click beside the Media field, and then click the "…" button. The Media dialog box displays.
7. Locate the media item created by a page developer for this Closeness Qualifier. A media item could be a graphic or string of text that alerts a user to their proximity to the promotion. You can find a media item in two ways:
   - Search by `repositoryItem` type in the Selective Listing tab using the List button.
   - Navigate to the media item in the folder hierarchy provided in the Folder Listing.
8. Once you have selected the media item, click OK.
9. If you want to specify a list of products as an Upsell Action, click the … button for that field. See Associating an Upsell Action to a Closeness Qualifier for instructions.
10. Click OK.

## Updating the Promotion

To make a promotion able to use Closeness Qualifiers, do the following:

1. In the Pricing > Promotions task area, locate the promotion and select it.
2. Under the Promotion Upsell label, find the Closeness Qualifiers property and confirm that it holds the appropriate Closeness Qualifiers for this promotion.
3. If the Closeness Qualifiers list needs to be modified, click the field beside the Closeness Qualifier property and then click the "…" button within it. Once the Closeness Qualifier dialog box opens, use the Add and Remove buttons to modify the list before clicking OK.
4. Under the Promotion Upsell label, locate the Enable Closeness Qualifiers property and toggle it to `true`.

## Detecting a Closeness Qualifier

Once you have created Closeness Qualifiers for a promotion by deciding when they should display to users and the products associated with them, you need to create a trigger action that signals ATG Commerce to check whether a Closeness Qualifier is appropriate for a user.

There are two ways of doing this. Instruct a page developer to use the `ClosenessQualiferDroplet` servlet bean on a page to check to see if any of the Closeness Qualifiers are applicable to the active user and to display the media item on that page. For example, if this servlet bean were used in a checkout page, when applicable Closeness Qualifiers are detected, you can display a media item beside a list of products in the shopping cart and their totals. For instructions on using the `ClosenessQualiferDroplet` servlet bean, see `ClosenessQualifierDroplet`.

Another way to detect when a user deserves Closeness Qualifier is to use scenarios. You can create a scenario that determines when a user qualifies for a Closeness Qualifier and then performs some other action, such as adding an item to a slot or sending an email. You can also use a scenario to determine when a user no longer qualifies for a Closeness Qualifier and react accordingly (for example, remove an item from a slot). Here are scenario elements that you should know about:

- The Promotion Closeness Qualification event determines when a user obtains a Closeness Qualifier. You can use this event to listen for any Closeness Qualifier or for a Closeness Qualifier with a certain `repositoryId`. See Promotion Closeness Qualification for more information.
- The Promotion Closeness Disqualification event is activated when a user no longer qualifies for a Closeness Qualifier. You can focus this event to monitor all Closeness Qualifiers or a Closeness Qualifier with a certain `repositoryId`. See Promotion Closeness Disqualification for more information.
- The Add Item to Slot action allows you to add Closeness Qualifiers using `PromotionUpsellTargeter` to a slot. Only Closeness Qualifiers that are relevant for the specified event are added here.
- The Add Item to Slot action allows you to add products that are part of an Upsell Action to a product slot using `PromotionUpsellProductTargeter`. Only Upsell Actions that are relevant for the specified event's Closeness Qualifiers are added here.

The sections below describe sample scenarios that provide upsell incentives.

## Adding an Item to a Slot When Users Qualifies for a Closeness Qualifier

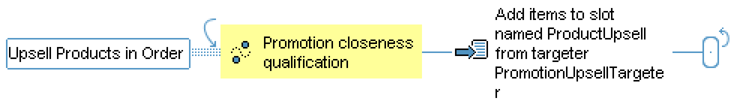
The following scenario specifies that, when a user has one or more Closeness Qualifiers, those Closeness Qualifiers are added to a slot. Then, a page developer can access the media items designed for each Closeness Qualifiers and display them to users in rotating succession.

Follow these steps to create the scenario:

1. In the Scenarios task area, create a new empty scenario (or a new segment in an existing scenario, if you have one that is appropriate). For more information, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.
2. Add an Event element to the scenario.
3. Select Promotion Closeness Qualification from the list of events. This event checks to see whether the user has any Closeness Qualifiers.
4. Click the check mark in the scenario element to close it.
5. Add an Action element to the scenario.
6. Select Add Items to Slot from the list of actions.
7. Select each of the following values from the successive drop down lists in the element editor:
   - `name`
   - a slot able to hold Closeness Qualifiers

- `from targeter`
- `PromotionUpsellTargeter`

8. Click the check mark in the scenario element to close it.

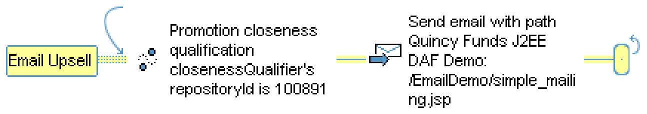

Upsell Products in Order
Promotion closeness qualification
Add items to slot named ProductUpsell from targeter PromotionUpsellTargete r

## Sending an Email When Users Qualify for A Closeness Qualifier

This example demonstrates how to create a scenario that sends an email to a user when that user qualifies for a particular Closeness Qualifier.

1. In order to create the scenario, you need to know the ID for the Closeness Qualifier. To find this information, locate the Closeness Qualifier in the Catalog Management > Catalog Elements task area by causing Items of type Closeness Qualifier to display.
2. Select the Closeness Qualifier. The `repositoryID` is in the ID field. If the ID field is not visible, go to Tools > Preferences on the menu, and select the Show expert-level information checkbox. Take note of the `repositoryID` for future reference.
3. In the Scenarios task area, create a new empty scenario (or a new segment in an existing scenario, if you have one that is appropriate). For more information, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.
4. Add an Event element to the scenario.
5. Select Promotion Closeness Qualification from the list of events. This event checks to see whether the user has any Closeness Qualifiers.
6. Specify the particular Closeness Qualifier by selecting each of the following values from the successive drop down lists in the element editor:
   - `where`
   - `closenessQualifier's` (Be sure that you select `closenessQualifier's`, not the default selection `closenessQualifer`.)
   - `repositoryId`
   - `is`
   - `Type a string`
7. In the space provided, enter the ID for the Closeness Qualifier that you noted earlier and click the check mark in the scenario element to close it.
8. Add an Action element to the scenario.
9. Select Send email from the list of actions. This action allows you to specify an email template that will be sent to all qualifying users.

10. Depending on where the email template lives, select With Path or With Dynamo Path. Use With Path when your template is on your file system. Use With Dynamo Path when your template exists in a context path defined within the ATG platform.
11. Click Choose mail template to open the Select a Document dialog box.
12. In the top portion, select the context path folder in which your template lives.
13. In the bottom portion, select the template. Click OK.
14. Click the check mark in the scenario element to close it.


Email Upsell

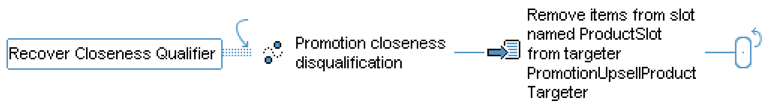
Promotion closeness qualification closenessQualifier's repositoryId is 100891

Send email with path Quincy Funds J2EE DAF Demo: /EmailDemo/simple_mailing.jsp

## Removing Closeness Qualifiers From a Slot When They No Longer Apply

Assume that each Closeness Qualifier appropriate for a user is added to a slot. If a user performs an action that makes him or her lose that Closeness Qualifier, you should remove the Closeness Qualifier from the slot.

1. In the Scenarios task area, create a new empty scenario (or a new segment in an existing scenario, if you have one that is appropriate). For more information, see the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.
2. Add an Event element to the scenario.
3. Select Promotion Closeness Disqualification from the list of events. This event checks to see whether the user has previously qualified for Closeness Qualifiers that he or she no longer qualifies for.
4. Click the check mark in the scenario element to close it.
5. Add an Action element to the scenario.
6. Select Remove Items from Slot from the list of actions.
7. Select each of the following values from the successive drop down lists in the element editor:
   - `name`
   - a slot able to hold Products
   - `from targeter`
   - `PromotionUpsellProductTargeter`
8. Click the check mark in the scenario element to close it.


Recover Closeness Qualifier
Promotion closeness disqualification
Remove items from slot named ProductSlot from targeter PromotionUpsellProduct Targeter

# Setting up Upselling Incentives

Users who don't qualify for promotions present an upselling opportunity: you can make them aware of what they need to do in order to receive a promotion. For example, you can encourage users to purchase more products by displaying a "Buy one more box of light bulbs and get the next one free!" message on a checkout page.

In order to display upsell incentives for a given promotion, you need to describe the circumstances under which those incentives are appropriate. For example, if a promotion gives free shipping to orders over $2500, you might want to display an upsell incentive when orders are $2400 that says "Free shipping if you spend $100!" You specify these circumstances as Closeness Qualifier items. A Closeness Qualifier has three main parts: a rule that says when it applies (orders greater than $2400), media items that display when the rule is met ("Free shipping if you spend $100!"), and a priority. When the Closeness Qualifier rule fits for a user's circumstance, the user's profile receives a reference to that Closeness Qualifier.

You can also define an Upsell Action, which is a list of upsell products. For example, products that cost $100 could make up your Upsell Action. Buying one such product qualifies a user for the free shipping promotion.

Once you have defined the Closeness Qualifiers and Upsell Actions for a promotion, you need to decide how those Closeness Qualifiers will be detected. You can design scenarios to add items to a slot that are accessed in a page or send an email when a user obtains a Closeness Qualifier.

Follow these steps in this order to create alert notices about a user's proximity to a promotion:

1. The Page Developer creates the text and media that advertises how close a user is to qualifying for the promotion. These items are called media items and you will need to know where they are located in order to complete the steps below.
2. Create the promotion. See Creating Promotions for instructions.
3. Describe the Closeness Qualifier, when users qualify for it, and the promotion it is associated with. See Creating Closeness Qualifiers.
4. Specify upsell products in an Upsell Action. See Working with Upsell Actions.
5. Enable the upsell option on the promotion. See Updating the Promotion.
6. Activate the upsell notification by creating a scenario that uses it or instruct a page developer to incorporate it into a JSP. See Detecting a Closeness Qualifier.

## Sample Upsell Incentives

Here are two promotions, each of which has two Closeness Qualifiers.

**Promotion: Free Shipping to Orders Over $2500**

The goal of the Closeness Qualifier Spend $100: encourages users whose cart contents cost between $2400-$2499 to purchase products that cost $100. To implement this Closeness Qualifier, you would need to do the following:

1. Create a Closeness Qualifier in the ACC and update its properties:
   - Qualifier: `When order's priceInfo's amount is greater than or equal to $2400`.
   - Closeness Qualifier Priority: `1`
2. This Closeness Qualifier requires an Upsell Action that has upsell products with a certain price. You can add products that cost $100 directly to the Fixed Upsell Products property on the Upsell Action.
3. A scenario might add this Closeness Qualifier to a slot so that a media item would advertise the upsell products to all qualifying users.

The goal of the Closeness Qualifier Buy This Brand: encourages users whose cart includes items from Manufacturer Z to purchase additional products from Manufacturer Z. To implement this Closeness Qualifier, you would need to do the following:

1. Create a Closeness Qualifier in the ACC and update its properties:
   - Qualifier: `When order contains at least 1 product whose Manufacturer's Manufacturer name is Z`.
   - Closeness Qualifier Priority: `1`

   Note the two Closeness Qualifiers have the same priority so that users are able to qualify for both of them. Two Closeness Qualifiers can have the same priority when each one has a unique, non-competing rule.
2. This Closeness Qualifier requires an Upsell Action to associate upsell products with Manufacturer Z. Create a product content group with the following rule:
   - `Include this item: items whose Manufacturer.displayName is Z`
3. A scenario might add this Closeness Qualifier to a slot so that a media item would advertise the upsell products to all qualifying users. You could use the same slot you specified for the other Closeness Qualifier.

**Promotion: Buy 3 product Xs and get 1 Y free**

The goal of Closeness Qualifier Buy One More X: encourages users who have two Xs in their carts to purchase a third in order to receive a free Y.

1. Create a Closeness Qualifier in the ACC and set the Qualifier property to `When order contains exactly 2 Xs`.
2. This Closeness Qualifier requires an Upsell Action to associate upsell products with a certain price. Create an Upsell Action and set the Fixed Upsell Products property to X.
3. A scenario might send an email to users who have two product Xs, explaining with the purchase of one more X they receive Y for free.

The goal of Closeness Qualifier Buy two More Xs: encourages users who have one X and one Y in their carts to purchase two more Xs so the Y is free.

1. Create a Closeness Qualifier in the ACC and set the Qualifier property to `When order contains exactly 1 X and order contains exactly 1 Y`.

   Note that priority is not important here because the two Closeness Qualifiers are not in direct competition. Based on their respective qualifier rules, it would be acceptable for a user to receive both Closeness Qualifiers, but their having different delivery mechanisms (email and scenario slot) may be reason for not wanting a user to receive both. Receiving two Closeness Qualifiers through different channels could overwhelm users. By not setting a priority on either, you specify that users are eligible to receive one Closeness Qualifier for this promotion.

2. This Closeness Qualifier requires an Upsell Action to associate upsell products with a certain price. Create an Upsell Action and set the Fixed Upsell Products property to X.

3. A scenario might add an advertisement to a slot that says “How can you buy just one X? Buy 2 more and your Y is free!”

## Working with Upsell Actions

An Upsell Action holds the products that you want to upsell. Each Closeness Qualifier specifies one Upsell Action that holds all products it aims to upsell. By purchasing products in the Upsell Action, a user could qualify for the promotion. Using an Upsell Action is optional. When you include an Upsell Action, you can create scenarios that add all products for an Upsell Action to a slot. A page developer can use an Upsell Action to determine the media items to display on a page.

There are two ways to specify products in an Upsell Action. You can specify a fixed group of products. It’s best to specify product IDs directly when you are upselling a few key products. Another option available to you is to create a content group. Content groups allow you to create a group of product SKUs based on the complex rules you define.

There are three tasks you need to do:

1. Create a content group if your Upsell Action requires one.
2. Create an Upsell Action.
3. Associate the Upsell Action to a Closeness Qualifier.

### *Creating a Content Group*

A content group is a collection of items that share the same `repositoryItem` type and satisfy a set of conditions. Using a content group allows you to form a collection of products that share some similarity without needing to know anything else about those products. If you’d like to use a content group, create it before creating your Upsell Action by following the instructions in the *Creating Content Groups* chapter of the *ATG Personalization Guide for Business Users*. When you are asked to supply a content source, specify `CustomProductCatalog` as the source and Product as the content type.

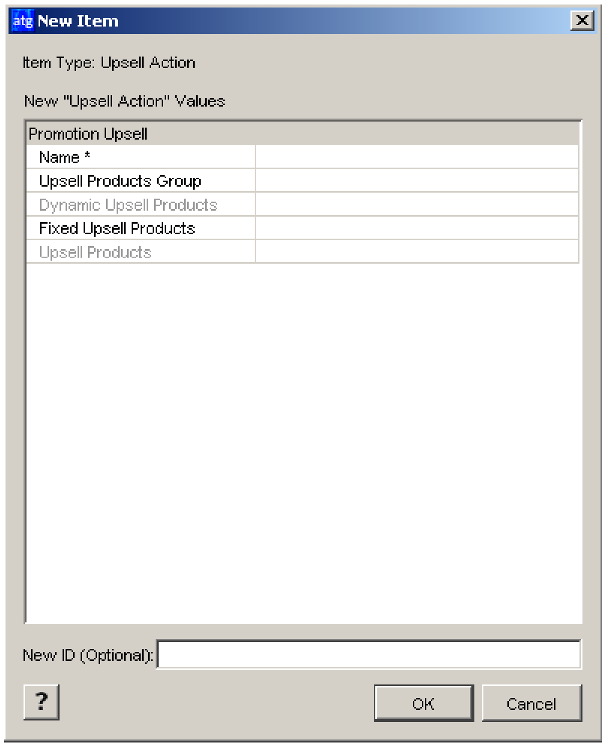
The groups you create are based on inclusion (all products with this start date) or exclusion (all products that aren’t on sale) rules.

Once you create the Upsell Action, you assign the content group to the Upsell Action's `upsellProductsGroup` property.

### *Creating an Upsell Action*

To define an Upsell Action:

1. In the Catalog Management > Catalog Elements task area, display items of type Upsell Action.
2. Select File > New Item to open the New Item dialog box.



3. In the Name field, provide a name to the Upsell Action that uniquely identifies it.
4. If you created a content group for your products, enter that content group name in the Upsell Products Group field. You can use only one content group per Upsell Action.
5. If you want to define a fixed set of products, click the field beside the Fixed Upsell Products property and then click the "…" button within it.

   The Fixed Upsell Products dialog box displays with Add and Remove buttons for modifying the list of products.
6. Click Add.

   The New Item dialog box displays.
7. Locate the products in one of two ways:

- Search by `repositoryItem` type in the Selective Listing tab using the List button.
- Navigate to the product in the folder hierarchy provided in the Folder Listing.

8. Select the products you want to add and click OK in the New Item dialog box.
9. Click OK in the Fixed Upsell Products dialog box.
10. Click OK in the New Item Dialog box.

**Note:** Ignore the two disabled properties Dynamic Upsell Products and Upsell Products. These properties are populated at runtime when the Upsell Action is used. When you specify a content group, the Dynamic Upsell Products property is populated by the products that make up the group. The Upsell Products property holds the complete set of products specified using the two methods (assembled by content groups or entered directly) for an Upsell Action.

### *Associating an Upsell Action to a Closeness Qualifier*

Associate an Upsell Action to a Closeness Qualifier as follows:

1. In the Catalog Management > Catalog Elements task area, display items of type Closeness Qualifier.
2. Select the Closeness Qualifier to which you want to assign the Upsell Action.
3. Click beside the Upsell Action field, and then click the "…"button. The Upsell dialog box displays with two tabs.
4. Locate the Upsell Action by displaying all items of type Upsell Action.
5. Once you selected the Upsell Action, click OK.

## Disabling Promotions

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 orders, which produces errors.

However, if you are certain that the promotion has not been used in orders, you can safely delete the promotion.

## Displaying Promotion Media

If you choose to create any media (text or images) for a promotion, the page developers working on your Web sites can display them on the appropriate site page by using any of the standard ATG techniques for displaying content. For example, they could set up a scenario that uses a slot element to show the promotion media.

## Setting Up Coupon Promotions

ATG Commerce treats coupons as a type of promotion (sometimes, coupons are referred to as "claimable promotions"). Coupons work as follows:

1. Add the coupon in the form of a promotion to the Promotion repository, following the same general steps that you take when you create a regular promotion item (see Creating Promotions). You set up the discount as appropriate for the coupon. For example, if you wanted the coupon to give 20% off any order, you would specify Order Discount – Percent Off as the Item Type, and then specify 20 as the Discount Percentage. You would set up a simple Discount Rule that applies the discount to any order:

   ```
   Condition:
   Always
   Apply discount to:
   Order Total
   ```

2. Add the coupon to the Gift Certificates and Coupons repository, associating it with the promotion you created in step 1. (The Gift Certificates and Coupons repository is part of the Claimable repository, which is described in the *Configuring Commerce Services* chapter in the *ATG Commerce Programming Guide*.) As part of this step, you also create a claim code for the coupon. For more information, see Adding a Coupon.

3. The page or application developer then sets up a form field where customers can enter the code. For example, he or she might add a field on the Checkout page and give it a label that says "Enter any coupon codes here:"

   The developer then hooks this field up to the part of ATG Commerce that handles coupons (initially, the `CouponFormHandler` component).

4. The page developer sets up an e-mail message (a JSP) to send to the customers you want to use the coupon. The message contains the claim code for the coupon as well as any additional text that you want to include. For information on setting up an e-mail message, see the *Working with Targeted E-Mail* chapter in the *ATG Personalization Guide for Business Users*.

5. You define the list of people whom you want to receive the message (in other words, the people you want to be able to use the coupon). One way to do this is by creating and enabling a scenario containing elements that define the group of people who will receive the message, and a Send Email element that specifies the message to send. For more information, see Specifying the People Who Receive the Promotion.

When customers want to use the coupon, they do the following:

1. They visit the Web site and order the products or services to which the coupon applies. During the ordering process, they type the coupon claim code in the appropriate field on the page (see step 3 above).

2. ATG Commerce references both the coupon repository and the corresponding item in the Promotions repository to determine, for example, whether the claim code is valid and the amount or type of discount to apply.

## Two Types of Coupons

In the ACC, you can see two types of coupons: unversioned coupons and coupons. The two types are identical in all ways but one: coupons are supported in a versioning system (such as Content Administration) while unversioned coupons are not.

If you have ATG Merchandising, you can create multiple versions of a coupon and publish the appropriate one to your sites using ATG Content Administration. So, if you have ATG Merchandising, use coupons; otherwise, use unversioned coupons.

## Adding a Coupon

This section explains how to add a coupon to the Gift Certificates and Coupons repository and create a claim code for it.

If you are using ATG'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.

1. In the ATG Control Center, select Purchases and Payments > Gift Certificates and Coupons.
2. In the search query box at the top of the window, specify `Items of type Coupon` and then click List. Any coupons already in your system appear in the left pane of the window.
3. Select File > New Item. The New Item dialog box appears, with `Coupon` selected as the item type.
4. In the New ID field, type the string you want to use as the coupon's claim code. This code is what customers enter when they want to apply the coupon to an order.

   Note that the claim codes for coupons are not case-sensitive.

   **Note:** Avoid creating coupon codes that are easy to guess. For example, it is not recommended that you create short or sequential codes (100, 101, and 102, for example). If you leave this field blank, ATG Commerce uses its internal ID generation system to create a random coupon code for you. It is often preferable to have ATG Commerce create the code than to do it yourself.
5. Specify the entries in the New "Coupon" Values area of the dialog box.
   - Display Name: enter a descriptive name for the coupon.
   - Expiration date: enter the date and time on which you want the coupon to become unusable. To do this, click the corresponding text box to display the "…" button, click the "…" button, and then select the date and time from the calendar that appears. This value is optional.

     Note that a coupon's expiration date is independent of the corresponding promotion's usage end date. Consequently, it is possible that a coupon may be invalid but the promotion to which it refers is still valid (and can still be delivered via a scenario). If you specify a value for this property, make sure it is consistent with the usage end date for the promotion to which it applies.
   - Parent folder: Select a coupon folder in which to store the coupon, if desired.

- Promotions: Click the corresponding text box to display the "…" button, click the "…" button, and then select the promotions to use with this coupon.

6. Click OK. The system adds the new coupon to the repository and displays it by its promotion name in the panel on the left of the window. You can hold your mouse over the promotion name to display the coupon claim code (or ID) in a pop-up window.

For more information on coupons, please refer to the *ATG Commerce Programming Guide*.

## Delivering Promotions via a URL

You can deliver a promotion to site visitors in a variety of ways, such as via a scenario or coupon. The preceding sections describe how to create promotions and use these delivery methods. This section provides information on another way to deliver a promotion to site visitors – via a URL on a JSP. You embed the ID of the promotion you want to offer in the URL. When a site visitor clicks that URL, the promotion is added to the visitor's `activePromotions` Profile property.

To deliver a promotion via a URL, do the following:

1. Enable the `/atg/commerce/promotion/PromotionServlet` by setting its `enabled` property to `true` in its `.properties` file. By default, ATG Commerce inserts `PromotionServlet` in the request-handling pipeline, but you must enable it for use.

2. Include a URL on the desired JSP using an anchor tag like the following:

```
<dsp:a href="../../samplepage.jsp" encode="true"><dsp:param
      value="promo10102" name="PROMO"/>Click here to get a 20% discount
      on shirts.</dsp:a>
```

   where `../../samplepage.jsp` is the page to which you want to link, and the value of the PROMO request parameter is the ID of the promotion in the `/atg/commerce/pricing/Promotions` repository.

   Note that `encode=true` is required in the anchor tag. This causes the PROMO parameter subtag to be encoded into the URL as a URL parameter and not a query parameter. In turn, `PromotionServlet` calls `getURLParameter()` on the request to retrieve the PROMO parameter value.

   Also note that the item descriptor of the promotion must be one of the item descriptors in `PromotionServlet.promotionItemDescriptorNames`.

When a request is passed into `PromotionServlet`, if the servlet is disabled, it simply passes the request to the next servlet in the pipeline. If it is enabled, it looks for a PROMO parameter in the URL parameters of the request and retrieves the associated promotion ID. Next, the servlet looks for a promotion in the Promotions repository whose ID is the given ID and whose item descriptor is included in `PromotionServlet.promotionItemDescriptorNames`. If the promotion exists, then `PromotionServlet` checks that either the profile is persistent or the promotion's Give To Anonymous Customers (`giveToAnonymousProfiles`) property is set to true. If either condition is true, the promotion is added to the user's `activePromotions` Profile property.

# 6 Managing Cost Centers

Cost Centers are an ATG Business Commerce feature that allows specific site customers to track their internal costs by designating certain parts of their organization as cost centers. ATG Business Commerce tracks which items and orders belong to each cost center.

For example, an insurance company could set up accounts with an office supply site so individual employees of the insurance company could log on and purchase supplies. The insurance company could set up several cost centers for each department that will be ordering supplies through the web site. When an insurance company employee logs in to purchase supplies, they would specify the department to which they belong. The insurance company could then track costs by department and run related reports.

There are two major parts to the cost center feature, assigning cost centers to profiles and assigning costs to cost centers within an order. The concept of assigning cost centers to profiles allows a site to define what cost centers a given user is allowed to assign costs. A user is assigned a list of cost centers. Adding costs to cost centers allows a user to create `CostCenter` objects within their order using the data from their profile. Then they can assign item, shipping, or tax costs to the cost centers.

This chapter includes the following sections:

**Viewing Existing Cost Centers**

**Adding New Cost Centers**

**Assigning a Default Cost Center to a User**

**Adding, Modifying, and Deleting Cost Centers in a Profile**

**Adding Cost Centers to an Order**

**Adding Items to a Cost Center**

**Tracking Orders by Cost Center**

**Cost Center Classes**

**Using the CostCenterFormHandler Framework**

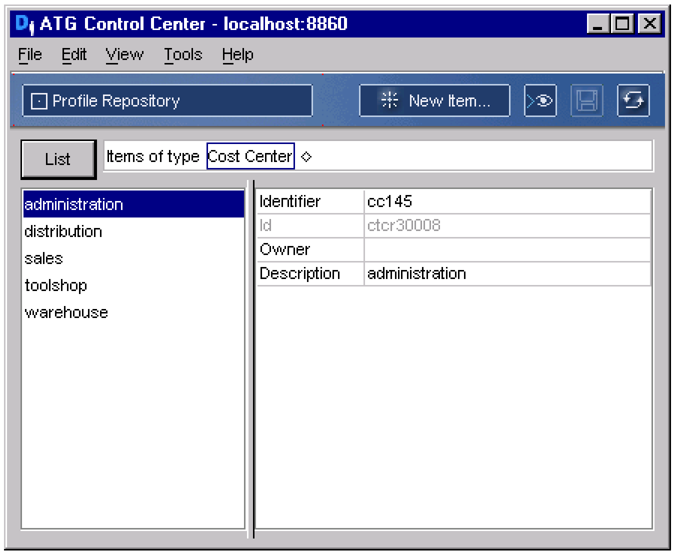
## Viewing Existing Cost Centers

Follow these steps to view the available cost centers in the ACC.

1. Select People and Organizations from the main task bar.
2. Click Profile Repository.

3. Select “Items of type Cost Center” from the search menu and click List.

A list of all available cost centers appears.


ATG Control Center - localhost:8860
File Edit View Tools Help
Profile Repository
New Item...
List
Items of type Cost Center

administration
distribution
sales
toolshop
warehouse

| | |
|---|---|
| Identifier | cc145 |
| Id | ctcr30008 |
| Owner | |
| Description | administration |

## Adding New Cost Centers

Follow these steps to add a new cost center using the ACC.

1. Search for existing cost centers in the profile repository. For more information, see Viewing Existing Cost Centers.
2. Click the New Item button.
3. Click New Item.
4. Enter and Identifier, Owner, and Description.
5. Click OK. The new cost center appears in the list of available cost centers.

## Assigning a Default Cost Center to a User

Assigning cost centers to user profiles allows you to define what cost centers a given user is allowed to assign costs. A user can be assigned a list of possible cost centers. Follow these steps to assign cost centers to a user using the ACC.

1. Select People and Organization from the main ACC navigation bar.
2. Select Profile Repository from the People and Organizations choices.
3. Select “Item of type User” and click List to view a list of all users.

4. Select a user from the list. The user's information displays in the main section of the screen.
5. Add a cost center to the Default Cost Center field in the Billing and Shipping section of the profile information. The cost center can be selected from a list of available cost centers by clicking on the "…" button.
6. Click Save to save the cost center the profile.

## Adding, Modifying, and Deleting Cost Centers in a Profile

You can add, modify, and delete cost center data for a user's profile through the `B2BCommerceProfileTools` class. Cost centers will be assigned to either a user or an organization (or sub-organization). A cost center will consist of an identifier and a description.

There are two ways in which cost centers can be added, edited, or deleted within a profile:

- Set up web forms that allow customers to manipulate the cost centers within their own profiles
- The site administrator can use the ATG Control Center to maintain cost center information.

The methods of `B2BCommerceProfileTools` allow you to add, modify, and delete of cost centers from a profile.

Setting up HTML forms for customers to use to make changes to the cost centers within their own profile is done using the `B2BCommerceProfileFormHandler` class, which extends `CommerceProfileFormHandler`. The `ProfileFormHandler` component is configured to point to the `B2BCommerceProfileFormHandler` class. The following table describes the properties of the `B2BCommerceProfileFormHandler` class:

| **Property** | **Description** |
|---|---|
| `AddCostCenterIdentifier` | Stores the identifier, or name, of a new cost center to be added to the repository. |
| `AddCostCenterDescription` | Stores the description of a new cost center to be added. |
| `EditCostCenterIdentifier` | Stores the identifier of a cost center that is to be modified. |
| `EditCostCenterDescription` | Stores the description that will be the new value for the description of the cost center identified by `EditCostCenterIdentifier`. |
| `RemoveCostCenterIdentifier` | Stores the identifier of the cost center to remove. |

| | |
|---|---|
| `DefaultCostCenter` | Boolean value that stores whether or not the given cost center should become the user's default cost center. |
| `AddCostCenterSuccessURL` | Stores the name of the JSP the user is to be taken to after successfully adding a cost center. |
| `AddCostCenterErrorURL` | Stores the name of the JSP the user is to be taken to after an error occurs while attempting to add a cost center. |
| `EditCostCenterSuccessURL` | Stores the name of the JSP the user is to be taken to after successfully editing a cost center. |
| `EditCostCenterErrorURL` | Stores the name of the JSP the user is to be taken to after an error occurs while attempting to edit a cost center. |
| `RemoveCostCenterSuccessURL` | Stores the name of the JSP the user is to be taken to after successfully removing a cost center. |
| `RemoveCostCenterErrorURL` | Stores the name of the JSP the user is to be taken to after an error occurs while attempting to remove a cost center. |

The adding, editing and removing cost centers can be performed by calling `handleAddCostCenter`, `handleEditCostCenter`, and `handleRemoveCostCenter`.

The following JSP example creates a form that allows a customer to add a cost center:

```
<h1>Add Cost Center</h1>
<dsp:form action="done.jsp" method="post">
  <dsp:input bean="ProfileFormHandler.AddCostCenterSuccessURL" value="success.jsp"
             type="hidden"/>
  <dsp:input bean="ProfileFormHandler.AddCostCenterErrorURL" value="error.jsp"
             type="hidden"/>
  Name: <dsp:input bean="ProfileFormHandler.AddCostCenterIdentifier"
             type="text"/><P>
  Description: <dsp:input bean="ProfileFormHandler.AddCostCenterDescription"
             type="text"/><P>
  <dsp:input bean="ProfileFormHandler.DefaultCostCenter" value="true"
             type="checkbox"/> Make default<P>
  <dsp:input bean="ProfileFormHandler.AddCostCenter" value="Add Cost Center"
             type="submit"/><P>
</dsp:form>
```

The following JSP example creates a form that allows a customer to edit a cost center:

```
<h1>Edit Cost Center</h1>
<dsp:form action="done.jsp" method="post">
  <dsp:input bean="ProfileFormHandler.EditCostCenterSuccessURL"
            value="success.jsp" type="hidden"/>
  <dsp:input bean="ProfileFormHandler.EditCostCenterErrorURL" value="error.jsp"
             type="hidden"/>
  Name: <dsp:input bean="ProfileFormHandler.EditCostCenterIdentifier"
                   type="text"/><P>
  New Description: <dsp:input bean="ProfileFormHandler.EditCostCenterDescription"
                              type="text"/><P>
  <dsp:input bean="ProfileFormHandler.DefaultCostCenter" value="true"
             type="checkbox"/>
  Make default<P>
  <dsp:input bean="ProfileFormHandler.EditCostCenter" value="Edit Cost Center"
             type="submit"/><P>
</dsp:form>
```

The following JSP example creates a form that allows a customer to remove a cost center:

```
<h1>Remove Cost Center</h1>
<dsp:form action="done.jsp" method="post">
  <dsp:input bean="ProfileFormHandler.RemoveCostCenterSuccessURL"
            value="success.jsp" type="hidden"/>
  <dsp:input bean="ProfileFormHandler.RemoveCostCenterErrorURL" value="error.jsp"
             type="hidden"/>
  Name: <dsp:input bean="ProfileFormHandler.RemoveCostCenterIdentifier"
                   type="text"/><P>
  <dsp:input bean="ProfileFormHandler.RemoveCostCenter" value="Remove Cost Center"
             type="submit"/><P>
</dsp:form>
```

## Adding Cost Centers to an Order

When a site customer is purchasing a product on a site that implements cost centers, the customer can be given the option of adding cost centers to an order. The `CostCenterDroplet` servlet bean can be used to display any cost centers assigned to the user as options for the user to choose. If there are no cost centers assigned to the user, the repository is configured to then search the parent organization for the user and use those cost centers as options. If, again, there are no cost centers, the repository will continue to the parent organization of that organization, and so on.

When a user assigns cost centers to an order, the cost center repository ID is included as part of the order repository item.

## Adding Items to a Cost Center

If more than one cost center is assigned to an order, it is necessary to choose which items in the order belong to each cost center. Items will be assigned to cost centers through a type of relationship called a `CostCenterCommerceItemRelationship`.

The `CostCenterShippingGroupRelationship` and `CostCenterOrderRelationship` can be used if costs are more conveniently tied to shipping groups or orders, rather than commerce items. This will also allow tracking of shipping and tax costs.

## Tracking Orders by Cost Center

Because the main purpose of cost centers is to allow organizations to better track their costs, a search method that get a list of orders associated with a given cost center.

Use the `getOrdersforCostCenter()` method in the `CostCenterManager` class to retrieve orders associated with a specific cost center.

## Cost Center Classes

The following section briefly describes the classes related to cost centers. For more information on each of these classes, refer to the *ATG API Reference*.

### *CostCenter Interface*

The `CostCenter` interface represents all the information included in a cost center. The `CostCenter` interface contains the following:

- `CostCenterClassType`
- `Identifier`
- `Description`
- `Amount`

`CostCenterImpl` is the default implementation of the `CostCenter` interface.

### *CostCenterManager*

`CostCenterManager` includes methods for manipulation of cost centers within the context of an order. This includes methods for adding, removing, and editing `CostCenter` and `CostCenterRelationship` objects. As well as methods for getting order information based on cost centers.

### *CostCenterContainer*

Handles access to a group of cost centers. `CostCenterContainerImpl` will be the default implementation of `CostCenterContainer`. This interface provides methods for managing a list of cost centers within an order.

### *CostCenterRelationship*

`CostCenterRelationship` represents a part of a relationship between items and a cost center. The `CostCenterRelationship` interface consists of two properties:

- `CostCenter`: references the relevant Cost Center
- `Amount`: indicates the total price attributed to that cost center within the given order.

### *CostCenterCommerceItemRelationship*

The `CostCenterCommerceItemRelationship` defines a relationship between a commerce item and the cost center to which it belongs. It implements both `CostCenterRelationship` and `CommerceItemRelationship`. This relationship has four possible types:

- `Amount`: indicates that the Amount property tells how much of the total cost of the item should be attributed to the given cost center.
- `AmountRemaining`: indicates that all costs of the item that are not assigned to a different cost center in a separate relationship will be attributed to the cost center in the given relationship.
- `Quantity`: indicates that the Quantity property tells how many of a given commerce item should be assigned to cost center.
- `QuantityRemaining`: indicates that all items of the item that are not assigned to a different cost center in a separate relationship will be attributed to the cost center in the given relationship.

### *CostCenterShippingGroupRelationship*

The `CostCenterShippingGroupRelationship` is used to tie the cost center to the shipping group, rather than each item within the shipping group. This relationship also allows assignment of shipping charges to cost centers. This class implements both `CostCenterRelationship` and `ShippingGroupRelationship`.

The `CostCenterShippingGroupRelationship` can be of two possible types:

- `ShippingAmount`
- `ShippingAmountRemaining`

### *CostCenterOrderRelationship*

`CostCenterOrderRelationship` is used to tie the cost center to the order, rather than each item within the shipping group, or to a specific shipping group. This class implements both `CostCenterRelationship` and `OrderRelationship`. This relationship will also allow assignment of tax and shipping charges to cost centers.

This relationship has four possible types:

- `TaxAmount`
- `TaxAmountRemaining`
- `OrderAmount`
- `OrderAmountRemaining`

## Using the CostCenterFormHandler Framework

The `CostCenterFormHandler` framework enables customers to gather complex `CostCenter` information from a user during the purchase process. The primary objective of this framework permits the user to associate their authorized cost centers with the order's various `CommerceIdentifiers`.

In order to facilitate the processing of this information, the `CostCenterFormHandler` framework utilizes the following helper classes:

- `CommerceIdentifierCostCenter`
- `CommerceIdentifierCostCenterContainer`
- `CostCenterMapContainer`
- `CostCenterContainerService`
- `CostCenterDroplet`
- `CostCenterFormHandler`

### *CommerceIdentifierCostCenter*

The `CommerceIdentifierCostCenter` object stores the information needed to associate cost centers, referenced by name, and `CommerceIdentifiers`. It contains the following properties:

| **Property name** | **Type** |
|---|---|
| `CommerceIdentifier` | `CommerceIdentifier` |
| `CostCenterName` | `String` |
| `RelationshipType` | `String` |
| `Amount` | `double` |
| `SplitAmount` | `double` |
| `Quantity` | `long` |
| `SplitQuantity` | `long` |
| `SplitCostCenterName` | `String` |

### CommerceIdentifierCostCenterContainer

The `CommerceIdentifierCostCenterContainer` interface keeps track of all the `CommerceIdentifierCostCenters` associated with the various Order `CommerceIdentifiers`.

### CostCenterMapContainer

The `CostCenterMapContainer` interface keeps track of all the user's cost centers by name, as well as a default cost center.

### CostCenterContainerService

The `CostCenterContainerService` is a `GenericService` that implements both container interfaces and comprises a convenient session-scoped component.

### CostCenterDroplet

The `CostCenterDroplet` servlet bean includes a request-scoped component whose service method is responsible for the following tasks:

- `CostCenter` initialization – The user's authorized cost centers are created and added to the `CostCenterMapContainer`.
- `CommerceIdentifierCostCenter` initialization - New `CommerceIdentifierCostCenter` instances are created specific to the current Order and added to the `CommerceIdentifierCostCenterContainer`.

During initialization, the `CostCenterDroplet` optionally creates one `CommerceIdentifierCostCenter` object for each `CommerceIdentifier` type (for example, each `CommerceItem`, each `ShippingGroup`, the Order, and the Tax) and with the following default properties:

| **Property** | **Description** |
|---|---|
| `CommerceIdentifier` | Set to reference the `CommerceItem`, `ShippingGroup`, or `Order` |
| `RelationshipType` | Set to the appropriate `RelationshipTypes` String property CCAMOUNT_STR, CCQUANTITY_STR, CCSHIPPINGAMOUNT_STR, CCORDERAMOUNT_STR, or CCTAXAMOUNT_STR. `CostCenterCommerceItemRelationships` default to CCQUANTITY_STR unless the boolean droplet parameter `useAmount` is true. |
| `CostCenterName` | Set to the `DefaultCostCenterName` of the `CostCenterMapContainer`. |
| `Amount` | Set to the Amount property of the `PriceInfo` of the `CommerceIdentifier`, if the `PriceInfo` exists |
| `Quantity` | Set to the `Quantity` property of the `CommerceItem`. |

For example, a simpler cost center page might permit the customer to assign a cost center to the entire Order, or split the entire Order cost into multiple cost centers. This simple page will not permit the user to assign cost centers to `CommerceItems`, `ShippingGroups`, or the `Tax`.

A more advanced `CostCenter` page could permit the customer to assign and split `CostCenter` associations among the `CommerceItem`, `ShippingGroup`, and the `Tax`. This is implemented by the same `CostCenterDroplet` servlet bean with different initialization parameters.

The `CostCenterDroplet` takes the following parameters:

| **Parameter** | **Description** |
|---|---|
| `initItemCostCenters` | Boolean which toggles `CommerceIdentifierCostCenter` initialization for `CommerceItems` |
| `InitShippingCostCenters` | Boolean which toggles `CommerceIdentifierCostCenter` initialization for `ShippingGroups` |
| `initOrderCostCenters` | Boolean which toggles `CommerceIdentifierCostCenter` initialization for the order |
| `InitTaxCostCenters` | Boolean which toggles `CommerceIdentifierCostCenter` initialization for the tax |
| `InitCostCenters` | Boolean which toggles placing the user's authorized `CostCenters` into the `CostCenterMapContainer` |
| `ClearCostCenterMap` | Boolean which toggles clearing the `CostCenters` in the `CostCenterMapContainer` |
| `ClearCostCenterContainer` | Boolean which toggles clearing the `CommerceIdentifierCostCenters` in the `CommerceIdentifierostCenterContainer` |
| `ClearAll` | Boolean which toggles clearing both containers |

Refer to the following code example of the `CommerceCenterDroplet` servlet bean:

```
<dsp:droplet name="CostCenterDroplet">
  <dsp:param bean="ShoppingCartModifier.order" name="order"/>
  <dsp:param value="false" name="clearAll"/>
  <dsp:param value="false" name="clearCostCenterMap"/>
  <dsp:param value="false" name="clearCostCenterContainer"/>
  <dsp:param value="true" name="initCostCenters"/>
  <dsp:param value="true" name="initItemCostCenters"/>
  <dsp:param value="true" name="initShippingCostCenters"/>
  <dsp:param value="true" name="initTaxCostCenters"/>
  <dsp:param value="false" name="useAmount"/>
  <dsp:oparam name="output">
  </dsp:oparam>
 </dsp:droplet>
```

### CostCenterFormHandler

The `CostCenterFormHandler` extends the `PurchaseProcessFormHandler`. It is a request-scoped component with two handler methods:

- `handleSplitCostCenters`

  This handler relies on the `splitCostCenter`, `splitAmount` and `splitQuantity` properties of the `CommerceIdentifierCostCenter` to split extra `CommerceIdentifierCostCenter` objects by quantity or by amount.

  In a form, the user might decide to split $50 of an original `CommerceIdentifier` amount of $100 onto a separate cost center. This will create a new `CommerceIdentifierCostCenter` object, and adjust the amount of both the original and the new `CommerceIdentifierCostCenter` objects to add up to the original `CommerceIdentifier` total amount.

- `handleApplyCostCenters`

  This handler, whose invocation is the objective of this entire framework, takes the information found in the containers and applies it to the current order. The `CommerceIdentifierCostCenter` associations created by the user are first scrutinized and the appropriate business methods are called in the `OrderManager` family based on the `RelationshipType` of the `CommerceIdentifierCostCenter`. Second, any `DefaultCostCenterName` of the `CostCenterMapContainer` is used to determine if any remaining Order amount is added to a cost center. This readily facilitates applications that apply a default cost center to any remaining Order amount not explicitly covered by other cost centers.

The following code sample is an example of how to use `CostCenterFormHandler`. The resulting JSP lists the items in a customer's order and has a text box and a drop-down list of available cost centers next to each item. Customers can use this page to specify the quantity of an item and associate a cost center with the items.

```
<dsp:form action="cost_centers_line_item.jsp" method="post">
 <tr>
  <td><br>
  <table border=0 cellpadding=6 cellspacing=0>
   <tr>
    <td></td>
    <td colspan=2><span class="big">Cost Centers</span>
    <dsp:include page="../common/FormError.jsp"></dsp:include></td>
   </tr>

   <tr valign=top>
    <td width=40><dsp:img hspace="20" src="../images/d.gif"/></td>
    <td>
```

```
<table border=0 cellpadding=4 cellspacing=1 width=85%>
 <tr>
  <td colspan=13><span class=help>Assign each line item to a cost center. You
   can also divide a line item between cost centers by entering the number of
   items to assign to the new cost center.
  </span></td>
 </tr>

 <tr valign=bottom bgcolor="#666666">
  <td colspan=2><span class=smallbw>Part #</span></td>
  <td colspan=2><span class=smallbw>Name</span></td>
  <td colspan=3><span class=smallbw>Qty</span></td>
  <td colspan=2><span class=smallbw>Qty. to move</span></td>
  <td colspan=2><span class=smallbw>Price</span></td>
  <td colspan=2><span class=smallbw>Cost Center</span></td>
  <!--<td colspan=2><span class=smallbw>Current Cost Center</span></td>-->
 </tr>
 <dsp:droplet name="ForEach">
  <dsp:param param="order.commerceItems" name="array"/>
  <dsp:oparam name="output">
   <dsp:setvalue paramvalue="element" param="commerceItem"/>
   <dsp:droplet name="BeanProperty">
    <dsp:param param="ciccMap" name="bean"/>
    <dsp:param param="commerceItem.id" name="propertyName"/>
    <dsp:oparam name="output">
     <dsp:setvalue paramvalue="propertyValue" param="ciccList"/>
     <dsp:droplet name="ForEach">
      <dsp:param param="ciccList" name="array"/>
      <dsp:oparam name="output">
       <!-- begin line item -->
       <tr valign=top>
        <td><dsp:valueof param="commerceItem.catalogRefId"/></td>
        <td></td>
        <td><dsp:a href="../catalog/product.jsp?navAction=jump">
            <dsp:param param="commerceItem.auxiliaryData.productId"
                  name="id"/>
            <dsp:valueof
                 param="commerceItem.auxiliaryData.productRef.displayName"/>
            </dsp:a></td>
        <td></td>

        <td>&nbsp;</td>
        <td align=right><dsp:valueof param="element.quantity"/></td>
        <td>&nbsp;</td>


        <td>
        <dsp:input
             bean='<%="CostCenterDroplet.CostCenterMapContainer.
```

```
                        CommerceIdentifierCostCenterMap." +
                        request.getParameter("commerceItem.id")+
                        "[param:index].splitQuantity"%>' size="4" value="0"
                         type="text"/></td>
               <td>&nbsp;</td>

                <td align=right><dsp:valueof param="element.amount"
                                      converter="currency"/></td>
               <td>&nbsp;</td>

               <td>
                <dsp:select bean='<%="CostCenterDroplet.CostCenterMapContainer.
                                     CommerceIdentifierCostCenterMap." +
                                     request.getParameter("commerceItem.id")+
                                     "[param:index].splitCostCenterName"%>'>
                <dsp:droplet name="ForEach">
                 <dsp:param param="costCenters" name="array"/>
                 <dsp:oparam name="output">
                  <dsp:getvalueof id="option178" param="element"
                       idtype="java.lang.String">
<dsp:option value="<%=option178%>"/>
</dsp:getvalueof><dsp:valueof param="element"/>
                 </dsp:oparam>
                </dsp:droplet>
                </dsp:select>
               </td>
               <td>&nbsp;</td>

               <td>&nbsp;
               </td>
               <td>
                 <dsp:valueof param="element.costCenterName">
                   <dsp:valueof bean="CostCenterDroplet.
                                     CostCenterMapContainer.defaultCostCenterName"/>
                  </dsp:valueof>

               </td>
              </tr>
              <!-- end line item -->
             </dsp:oparam>
            </dsp:droplet>
           </dsp:oparam>
          </dsp:droplet>
         </dsp:oparam>
        </dsp:droplet>

        <dsp:droplet name="ForEach">
         <dsp:param param="order.shippingGroups" name="array"/>
         <dsp:oparam name="output">
          <dsp:setvalue paramvalue="element" param="shippingGroup"/>
```

```
       <dsp:droplet name="BeanProperty">
        <dsp:param param="ciccMap" name="bean"/>
        <dsp:param param="shippingGroup.id" name="propertyName"/>
        <dsp:oparam name="output">
         <dsp:setvalue paramvalue="propertyValue" param="ciccList"/>
         <dsp:droplet name="ForEach">
          <dsp:param param="ciccList" name="array"/>
          <dsp:oparam name="output">
           <!-- begin shipping line item -->
           <tr valign=top>
            <td colspan=7>
            <dsp:valueof param="shippingGroup.description"/></td>
            <td></td>
            <td></td>

            <td align=right><dsp:valueof param="element.amount"
                                     converter="currency"/></td>
            <td></td>

            <td>
             <dsp:select bean='<%="CostCenterDroplet.CostCenterMapContainer.
                  CommerceIdentifierCostCenterMap." +
                  request.getParameter("shippingGroup.id")+
                  "[param:index].splitCostCenterName"%>'>
             <dsp:droplet name="ForEach">
              <dsp:param param="costCenters" name="array"/>
              <dsp:oparam name="output">
               <dsp:getvalueof id="option279" param="element"
                    idtype="java.lang.String">
<dsp:option value="<%=option279%>"/>
</dsp:getvalueof><dsp:valueof param="element"/>
              </dsp:oparam>
             </dsp:droplet>
             </dsp:select>
            </td>
            <td>&nbsp;</td>
            <td>&nbsp;
            </td>
            <td>
              <dsp:valueof param="element.costCenterName">
               <dsp:valueof bean="CostCenterDroplet.CostCenterMapContainer.
                    defaultCostCenterName"/>
            </dsp:valueof>
            </td>
           </tr>
           <!-- end shipping line item -->
          </dsp:oparam>
         </dsp:droplet>
        </dsp:oparam>
       </dsp:droplet>
```

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

                 <dsp:droplet name="BeanProperty">
          <dsp:param param="ciccMap" name="bean"/>
          <dsp:param param="order.id" name="propertyName"/>
          <dsp:oparam name="output">
           <dsp:setvalue paramvalue="propertyValue" param="ciccList"/>
           <dsp:droplet name="ForEach">
            <dsp:param param="ciccList" name="array"/>
            <dsp:oparam name="output">
             <!-- begin tax line item -->


                 <dsp:droplet name="Switch">
                 <dsp:param param="element.RelationshipType" name="value"/>
                 <dsp:oparam name="CCTAXAMOUNT">
       <tr valign=top>
              <td colspan=7>Tax</td>
              <td></td>
              <td></td>

              <td align=right><dsp:valueof param="element.amount"
              converter="currency"/></td>

              <td></td>
              <td>
               <dsp:select bean='<%="CostCenterDroplet.CostCenterMapContainer.
                                     CommerceIdentifierCostCenterMap." +
                                     request.getParameter("order.id")+
                                     "[param:index].splitCostCenterName"%>'>
               <dsp:droplet name="ForEach">
                <dsp:param param="costCenters" name="array"/>
                <dsp:oparam name="output">
                 <dsp:getvalueof id="option376" param="element"
                             idtype="java.lang.String">
    <dsp:option value="<%=option376%>"/>
    </dsp:getvalueof><dsp:valueof param="element"/>
                </dsp:oparam>
               </dsp:droplet>
               </dsp:select>
              </td>
              <td>&nbsp;</td>
              <td>&nbsp;
              </td>
              <td>
                <dsp:valueof param="element.costCenterName">
                 <dsp:valueof bean="CostCenterDroplet.
                                   CostCenterMapContainer.defaultCostCenterName"/>
              </dsp:valueof>
              </td>
              <td>&nbsp;</td>
             </tr>
             <!-- end tax line item -->
              </dsp:oparam>
              </dsp:droplet>

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

         <tr>
          <td colspan=13>
          <table border=0 cellpadding=0 cellspacing=0 width=100%>
           <tr bgcolor="#666666">
            <td><dsp:img src="../images/d.gif"/></td>
           </tr>
          </table>
          </td>
         </tr>
       </table>
       </td>
      </tr>
      <tr>
       <td></td>
         <dsp:input bean="CostCenterFormHandler.applyCostCentersSuccessURL"
                   value="confirmation.jsp" type="hidden"/>
         <dsp:input bean="CostCenterFormHandler.splitCostCentersSuccessURL"
                   value="cost_centers_line_item.jsp?init=false" type="hidden"/>
       <td><span class=help>You must save changes before continuing.</span><p>
    <!--     <dsp:input bean="CostCenterFormHandler.order" type="hidden"
                   beanvalue="ShoppingCart.current"/> -->
     <dsp:input bean="CostCenterFormHandler.splitCostCenters" value="Save changes"
                type="submit"/>
     <dsp:input bean="CostCenterFormHandler.applyCostCenters" value="Continue"
                type="submit"/>

       </td>
      </tr>
     </table>
     </td>
    </tr>
    </table>
    </dsp:form>
```

# 7 Using Commerce Elements in Scenarios

ATG Commerce provides a number of Event, Condition, and Action elements that you can use when creating scenarios for your commerce site. This chapter describes these ATG Commerce elements and includes the following sections:

**Using Commerce Event Elements in Scenarios**
Describes the Event elements provided with ATG Commerce.

**Using Commerce Condition Elements in Scenarios**
Describes the Condition elements provided with ATG Commerce.

**Using Commerce Action Elements in Scenarios**
Describes the Action elements provided with ATG Commerce.

**Using Scenarios to Cross-Sell and Up-Sell Products**
Describes the scenarios, scenario templates, and supporting elements that you can use to cross-sell and up-sell products based on the customer's current shopping cart.

For information about elements provided for the Motorprise Store, refer to the *ATG Business Commerce Reference Application Guide*. For information about the elements provided with the ATG Adaptive Scenario Engine, see the *ATG Personalization Guide for Business Users*.

**Note:** This chapter assumes you have read and understand the information provided in the *Creating Scenarios* chapter in the *ATG Personalization Guide for Business Users*.

## Using Commerce Event Elements in Scenarios

In a scenario, an Event element is the "what" part of the scenario. An Event element defines the visitor behavior that you want to identify and use as a trigger for the next element in the scenario, for example, "visits page in folder /shoes" or "views an item from Products in the category 'shoes'."

This section describes the Event elements provided with ATG Commerce for use in scenarios. These commerce Event elements are in addition to those provided with the Scenarios module. For information about scenarios, how to use Event elements in scenarios, and non-commerce Event elements, refer to the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

If the Event elements provided with ATG Commerce do not meet all of your requirements, your application developers can create custom ones. Custom Event elements appear in the ATG Control Center, and you can use them as you would use any standard Event element.

**Note for developers:** For information about creating custom Event elements, refer to the *Adding Custom Events, Actions, and Conditions to Scenarios* chapter in the *ATG Personalization Programming Guide*. If you need to extend the Scenario module's grammar for use with custom ATG Commerce elements, also refer to the *Commerce-Related Grammar Configuration* section of the *Configuring the ATG Expression Editor* chapter in the same guide.

Also note that Event elements are triggered when the `ScenarioManager` component receives events sent as Dynamo Message System messages. The description of each Event element in this chapter identifies the message that triggers the element, as well as the component or part of the system that is responsible for sending the message. The optional parameters that appear for a specific Event element correspond to the properties of the message bean that represents that event, including properties inherited from any parent classes. For more information about the Dynamo Message System, refer to the *Dynamo Message System* chapter of the *ATG Programming Guide*.

The Event elements provided with ATG Commerce are listed below in alphabetical order.

## Approval Complete Event

(ATG Business Commerce only)

The system watches for the order approval system to determine that an order has completed the approval process. You can use the optional parameters within this element to further define the type of order to watch for.

Technical note: This element is triggered when the `ScenarioManager` receives an `ApprovalComplete` message sent by the `sendApprovalCompleteMessage` processor (class `atg.b2bcommerce.approval.processor.ProcSendApprovalCompleteMessage`). The processor sends this message when an order passes through the `checkApprovalComplete` pipeline chain and has been determined to have completed the order approval process. For more information, see the *Managing the Order Approval Process* chapter in the *ATG Commerce Programming Guide*.

## Approval Required Event

(ATG Business Commerce only)

The system watches for the order approval system to determine that an order requires approval by a qualified approver. You can use the optional parameters within this element to further define the type of order to watch for.

Technical note: This element is triggered when the `ScenarioManager` receives an `ApprovalRequired` message sent by the `sendApprovalRequiredMessage` processor (class `atg.b2bcommerce.approval.processor.ProcSendApprovalRequiredMessage`). The processor sends this message when an order passes through the `approveOrder` pipeline chain and has been determined to require approval. For more information, see the *Managing the Order Approval Process* chapter in the *ATG Commerce Programming Guide*.

## Approval Update Event

(ATG Business Commerce only)

The system watches for the order approval system to indicate that an order has been approved or rejected by an approver. You can use the optional parameters within this element to further define the type of order to watch for.

Technical Note: This element is triggered when the `ScenarioManager` receives an `ApprovalUpdate` message. The `ApprovalUpdate` message is sent by either the `sendApprovalUpdateMessageForApproval` processor (class `atg.b2bcommerce.approval.processor.ProcSendApprovalMessage`) in the `orderApproved` pipeline chain or the `sendApprovalUpdateMessageForRejection` processor (class `atg.b2bcommerce.approval.processor.ProcSendApprovalMessage`) in the `orderRejected` pipeline chain, depending on whether the approver approved or rejected the order. For more information, see the *Managing the Order Approval Process* chapter in the *ATG Commerce Programming Guide*.

## FulfillOrderFragment

The system watches for the order fulfillment system to send shipping group information about a new order to the part of the system that will be responsible for processing it (for example, the `HardgoodFulfiller`). You can use the optional parameters within this element to further define the type of element to watch for.

Technical note: This element is triggered when the `ScenarioManager` receives a `FulfillOrderFragment` message sent by the `OrderFulfiller`. For more information, see the *Configuring the Order Fulfillment Framework* chapter in the *ATG Commerce Programming Guide*.

## Gift Purchased

The system watches for an order that a customer has designated as a gift to be processed. You can use the optional parameters within this element to further define the type of gift purchase to watch for.

Technical note: This element is triggered when the `ScenarioManager` receives a `GiftPurchased` message sent by the `SendGiftPurchasedMessage` processor (class `atg.commerce.order.processor.ProcSendGiftPurchasedMessage`).

## Inventory Threshold Reached

The system watches for the inventory level of items in the catalog to drop below a given value.

Technical note: This element is triggered when the `ScenarioManager` receives an `InventoryThresholdReached` message sent by the `InventoryManager`. This message indicates that the `stockLevel` value for the specific item has dropped below the `stockThreshold` value (or the `backorderLevel` value is less than `backorderThreshold`, or `preorderLevel` is below `preorderThreshold`). For more information, see the *Inventory Framework* chapter in the *ATG Commerce Programming Guide*.

## Invoice Is Created

(ATG Business Commerce only)

The system watches for an invoice to be added to the Invoices repository. You can use the optional parameters within this element to further define the type of invoice to watch for.

Technical Note: This element is triggered when the `ScenarioManager` receives a `CreateInvoice` message sent by a pipeline chain that is invoked by the `InvoiceManager`'s `addInvoice()` method. For more information, see the *Generating Invoices* chapter in the *ATG Commerce Programming Guide*.

## Invoice Is Removed

(ATG Business Commerce only)

The system watches for an invoice to be removed from the Invoices repository. You can use the optional parameters within this element to further define the type of invoice to watch for.

Technical Note: This element is triggered when the `ScenarioManager` receives a `RemoveInvoice` message sent by a pipeline chain that is invoked by the `InvoiceManager`'s `removeInvoice()` method. For more information, see the *Generating Invoices* chapter in the *ATG Commerce Programming Guide*.

## Invoice Is Updated

(ATG Business Commerce only)

The system watches for an invoice in the Invoices repository to be updated. You can use the optional parameters within this element to further define the type of invoice to watch for.

Technical Note: This element is triggered when the `ScenarioManager` receives an `UpdateInvoice` message sent by a pipeline chain that is invoked by the `InvoiceManager`'s `updateInvoice()` method. For more information, see the *Generating Invoices* chapter in the *ATG Commerce Programming Guide*.

## Item Added to Order

The system watches for a customer to add an item to a new or existing order. You can use the optional parameters within this element to further define the combination of item and order that you want to trigger this element.

Example: *Item Added to Order and SKU named Silver Helmet*. This element watches for a catalog item called Silver Helmet to be added to an order.

Technical note: This element is triggered when the `ScenarioManager` component receives an `ItemAddedToOrder` message sent by the `SendScenarioEvent` processor (class `atg.commerce.order.processor.ProcSendScenarioEvent`).

## Item Quantity Changed in Order

The system watches for a customer to change the quantity of an existing item in an order. You can use the optional parameters within this element to further define the type of order or item to watch for.

Technical note: This element is triggered when the `ScenarioManager` receives an `ItemQuantityChanged` message sent by the `SendScenarioEvent` processor. For more information, see the *sendScenarioEvent Pipeline Chain* section in the *Commerce Processor Chains* chapter of the *ATG Commerce Programming Guide*.

## Item Removed from Order

The system watches for a customer to remove an item from a new or existing order. You can use the optional parameters within this element to further define the combination of item and order that you want to trigger this element.

Technical note: This element is triggered when the `ScenarioManager` receives an `ItemRemovedFromOrder` message sent by the `SendScenarioEvent` processor (class `atg.commerce.order.processor.ProcSendScenarioEvent`).

## Modify Order

The system watches for the order fulfillment system to make any type of change to an order that has been submitted for processing. You can use the optional parameters within this element to further define the changes that you want to trigger this element.

Technical note: This element is triggered when the `ScenarioManager` receives a `ModifyOrder` message, which can be sent by any part of the system that requests or handles changes to an order (for example, the `OrderFulfiller`). For more information, see the *Configuring the Order Fulfillment Framework* chapter in the *ATG Commerce Programming Guide*.

## Modify Order Notification

The system watches for the order fulfillment system to indicate that a change has occurred in the status of an order, either as a result of receiving a `ModifyOrder` message (see above) or as part of processing a new order. You can use the optional parameters within this element to further define the notification that you want to trigger this element.

Technical note: This element is triggered when the `ScenarioManager` receives a `ModifyOrderNotification` message, which can be sent by any part of the system that makes changes to an order (for example, the `HardGoodFulfiller`). For more information, see the *Configuring the Order Fulfillment Framework* chapter in the *ATG Commerce Programming Guide*.

## Order Changes

The system watches for the order fulfillment system to indicate that the status of an order has changed in any one of the following ways:

- The order has been completed.

- The order contains an item that is unavailable.
- A customer has cancelled the order.
- The order is in a waiting state, requiring attention from someone involved in the order fulfillment process; for example, the system may have been unable to process the customer's credit card information and must wait for a customer service representative to take appropriate action.

You can use the optional parameters within this element to define the change that you want to trigger this element.

Example: *Order Changes where Sub type is Order Was Removed*. This element watches for ATG Commerce to indicate that a customer has cancelled an order.

Technical note: This element is triggered when the `ScenarioManager` receives an `OrderModified` message sent by the `OrderChangeHandler`. For more information, see the *Configuring the Order Fulfillment Framework* chapter in the *ATG Commerce Programming Guide*.

## Order Submitted

The system watches for a customer to complete the checkout process for an order.

Technical note: This element is triggered when the `ScenarioManager` receives a `SubmitOrder` message sent by the `SendFulFillmentMessage` processor (class `atg.commerce.order.processor.ProcSendFulfillmentMessage`).

## Orders Merged

The system watches for an anonymous user's shopping cart to be merged into a registered user's current shopping cart when the anonymous user logs in as a registered user. You can use the optional parameters within this element to further define the type of event to watch for.

Technical Note: This element is triggered when the `ScenarioManager` receives an `OrdersMerged` message sent by the `SendScenarioEvent` processor. For more information, see the *sendScenarioEvent Pipeline Chain* section in the *Commerce Processor Chains* chapter in the *ATG Commerce Programming Guide*.

## Payment Group Changes

The system watches for ATG Commerce to indicate that a change in status has occurred for the payment information within an order. The source of the change is an external system, such as a Customer Service application. You can use the optional parameters within this element to further define the message that you want to trigger this element.

Technical note: This element is triggered when the `ScenarioManager` receives a `PaymentGroupModified` message sent by the `OrderChangeHandler`. For more information, see the *Configuring the Order Fulfillment Framework* chapter.

## Price Changed

The system watches for the price of an order to change at the order subtotal level (that is, the sum of the item prices has changed due to a change in item quantities and/or the addition or deletion of items). You can use the optional parameters within this element, such as the new order price, to further define the type of order to watch for. Note that, depending on the pricing operation that was performed on the order, the new order price reflects either the order subtotal (that is, no shipping or tax costs) or the order total.

Technical note: This element is triggered when the `ScenarioManager` receives a `PriceChanged` message sent by the `PricingTools` object (class `atg.commerce.pricing.PricingTools`). `PricingTools` sends this message if the `PricingTools.generatePriceChangedEvents` property is true and if, when the order is repriced, the order's subtotal price is found to have changed. Note that, depending on the pricing operation that was performed on the order, the total in the `OrderPriceInfo` object that is contained in the `PriceChanged` message reflects either the order subtotal or the order total. For more information, see the *Using and Extending Pricing Services* chapter of the *ATG Commerce Programming Guide*.

You can extend `PricingTools.createPriceChangedEvent` to respond to more object types than just the order (such as items or shipping groups) and set the `PriceChangeType` to one of the given choices (these are static constants in the `PriceChanged` class).

## Promotion Closeness Disqualification

The system detects when a user no longer qualifies for a Closeness Qualifier. This element can be configured to watch a particular Closeness Qualifier or all Closeness Qualifiers. To set this element to watch for a particular Closeness Qualifier, you need to indicate "where `closenessQualifier's repositoryID` is *ID*" where ID is the actual ID of the Closeness Qualifier. See Detecting a Closeness Qualifier for a sample scenario that use this event.

## Promotion Closeness Qualification

For users who aren't currently receiving a promotion, the system checks if they have a particular Closeness Qualifier or any Closeness Qualifiers, depending on how you set up this element. To set this element to watch for a particular Closeness Qualifier, you need to indicate "where `closenessQualifier's repositoryID` is *ID*" where ID is the actual ID of the Closeness Qualifier. This element is often followed by a Send Email action or an Add Items to a Slot action. See Detecting a Closeness Qualifier for sample scenarios that use this event.

## Promotion Offered

The system watches for a scenario to grant a promotion to a user, or, more specifically, for the scenario to execute a "Give Promotion" action. You can use the optional parameters within this element to further define the type of event to watch for.

For more information on the "Give Promotion" action element, see Using Commerce Action Elements in Scenarios in this chapter.

Technical Note: This element is triggered when the `ScenarioManager` receives a `PromotionGrantedMessage` sent by a `PromotionAction` scenario action.

### Promotion Revoked

The system watches for a scenario to revoke an active promotion from a user, or, more specifically, for the scenario to execute a "Revoke Promotion" action. You can use the optional parameters within this element to further define the type of event to watch for.

For more information on the "Revoke Promotion" action element, see Using Commerce Action Elements in Scenarios in this chapter.

Technical Note: This element is triggered when the `ScenarioManager` receives a `PromotionRevokedMessage` sent by a `RemovePromotionAction` scenario action.

### Scenario Added an Item to an Order

The system watches for a scenario to add an item to an order, or, more specifically, for the scenario to execute an "Add Item to Order" action. You can use the optional parameters within this element to further define the type of order or item to watch for.

For more information on the "Add Item to Order" action element, see Using Commerce Action Elements in Scenarios in this chapter.

Technical note: This element is triggered when the `ScenarioManager` receives a `ScenarioAddedItemToOrder` message sent by the `SendScenarioEvent` processor. To disable the firing of this event, set the `PromotionTools.sendEventOnAddItem` property to false; note that the default value is true.

For more information on the `SendScenarioEvent` processor, see the *sendScenarioEvent Pipeline Chain* section in the *Commerce Processor Chains* chapter in the *ATG Commerce Programming Guide*.

### Scheduled Order Event

The system watches for a scheduled order event, such as the creation, update, or deletion of a scheduled order, or a scheduled order error or failure. You can use the optional parameters within this element to further define the type of scheduled order event to watch for.

Technical Note: This element is triggered when the `ScenarioManager` receives a `ScheduledOrderMessage` message sent by the `ScheduledOrderHandler` form handler. The type of scheduled order event that occurs depends on the value of the message's `action` property. For more information, see the *Scheduling Recurring Orders* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

### Shipping Group Changes

The system watches for ATG Commerce to indicate that a change in status has occurred for the shipping information within an order. The source of the change is an external system, such as a Customer Service application. You can use the optional parameters within this element to further define the message that you want to trigger this element.

Technical note: This element is triggered when the `ScenarioManager` receives a `ShippingGroupModified` message sent by the `OrderChangeHandler`. For more information, see the *Configuring the Order Fulfillment Framework* chapter in the *ATG Commerce Programming Guide*.

### Update Inventory

The system watches for the stock level of preordered, backordered, or out-of-stock catalog items to increase. You can use the optional parameters within this element to further define the type of event to watch for.

Technical note: This element is triggered when the `ScenarioManager` receives an `UpdateInventory` message sent by the `InventoryManager`. For more information, see the *Inventory Framework* chapter in the *ATG Commerce Programming Guide*.

### Uses Promotion

The system watches for a customer to use a predefined discount to purchase a product. You can use the optional parameters within this element to specify a promotion to watch for.

Example: *Uses promotion named Buy 2 Helmets, Get One Free*.

Technical note: This element is triggered when the `ScenarioManager` receives a `PromotionUsed` message sent by the `SendPromotionUsedMessage` processor (class `atg.commerce.order.processor.ProcSendPromotionUsedMessage`). The processor sends this message when an order containing a promotion passes through the `processOrder` pipeline chain. For more information, see the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

## Using Commerce Condition Elements in Scenarios

In a scenario, a Condition element follows and further qualifies an Event element, essentially adding an "if" statement to the Event element. The options that appear for the condition vary according to the event that the condition follows. For example, after a "visits page in folder /shoes" Event element, you could have a Condition element that specifies the exact page: "if page is `/shoes/hikingboots.jsp.`"

This section describes the Condition elements provided with ATG Commerce for use in scenarios. These commerce Condition elements are in addition to those provided with the Scenarios module. For information about scenarios, how to use Condition elements in scenarios, and non-commerce Condition elements, refer to the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

If the Condition elements provided with ATG Commerce do not meet all of your requirements, your application developers can create custom ones. Custom Condition elements appear in the ATG Control Center, and you can use them as you would use any standard Condition element.

**Note for developers:** For information about creating custom Condition elements, refer to the *Adding Custom Events, Actions, and Conditions to Scenarios* chapter in the *ATG Personalization Programming Guide*. If you need to extend the Scenario module's grammar for use with custom ATG Commerce elements, also refer to the *Commerce-Related Grammar Configuration* section of the *Configuring the ATG Expression Editor* chapter in the same guide.

The Condition elements provided with ATG Commerce are described in the sections that follow.

### Item Where

Allows you to further qualify one of the item-related events from ATG Commerce. The condition uses the same criteria as the discount rules for an item-based promotion. Example:

```
Item where product named Shatterproof helmet
```

See the Creating and Maintaining Promotions chapter for more information.

### Order Where

Allows you to further qualify one of the order-related events from ATG Commerce. The condition uses the same criteria as the discount rules for an order-based promotion. Example:

```
Order where order contains at least 1 (product in category named BMXBikes)
```

See the Creating and Maintaining Promotions chapter for more information.

## Using Commerce Action Elements in Scenarios

In a scenario, an Action element extends the "what" part of the scenario. While an Event element defines what the site visitor does, an Action element defines what the system does in response. For example, you can have the system send an e-mail, modify a specific attribute in a user's profile, or display a specific piece of content in a slot. (For more information on slots, see the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.)

This section describes the Action elements provided with ATG Commerce for use in scenarios. These commerce Action elements are in addition to those provided with the Scenarios module. For information about scenarios, how to use Action elements in scenarios, and non-commerce Action elements, refer to the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

If the Action elements provided with ATG Commerce do not meet all of your requirements, your application developers can create custom ones. Custom Action elements appear in the ATG Control Center, and you can use them as you would use any standard Action element.

**Note for developers:** For information about creating custom Action elements, refer to the *Adding Custom Events, Actions, and Conditions to Scenarios* chapter in the *ATG Personalization Programming Guide*. If you need to extend the Scenario module's grammar for use with custom ATG Commerce elements, also refer to the *Commerce-Related Grammar Configuration* section of the *Configuring the ATG Expression Editor* chapter in the same guide.

The Action elements provided with ATG Commerce are listed below in alphabetical order.

### Add Item to Order

Use to add a specific item to a customer's shopping cart.

### Fill Related Items to Slot

Use to add products related to those in the customer's current shopping cart to a given slot. Specify the slot to use and the products to show in it by property type (for example, `relatedProducts`).

By default, related products are **not** filled in the given slot when the shopping cart is empty. To fill the slot with related products even when the shopping cart is empty, specify, "Add these items if Shopping Cart is empty" at the end of the Action element.

For more information on using this Action element, see Using Scenarios to Cross-Sell and Up-Sell Products.

### Give Promotion

Use to make a specific promotion available to the site visitor. The system adds the promotion to the `activePromotions` attribute in the visitor's profile.

### Revoke Promotion

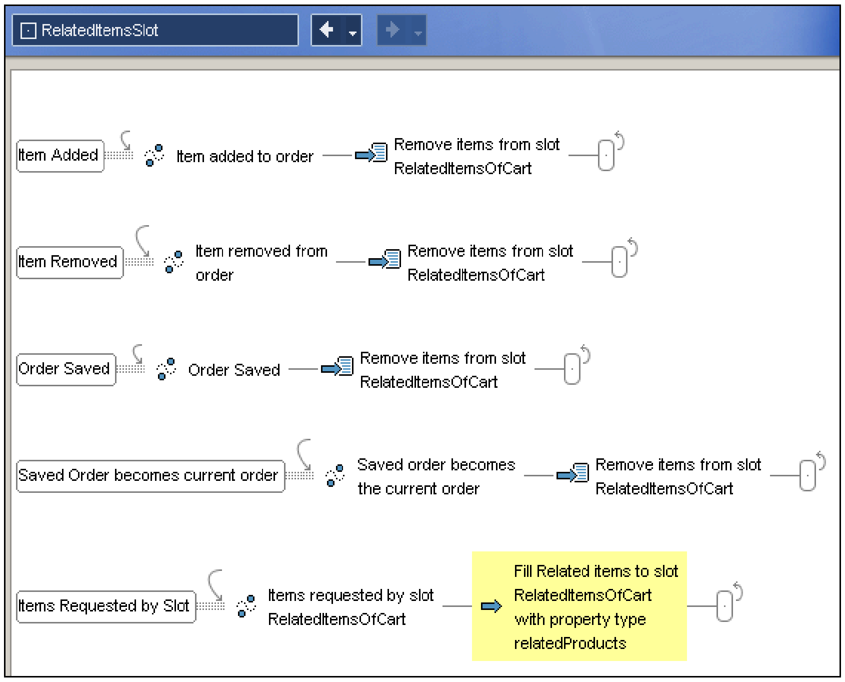
Use to remove a specific promotion from the `activePromotions` attribute in the visitor's profile.

## Using Scenarios to Cross-Sell and Up-Sell Products

ATG Commerce provides you with a predefined scenario named `RelatedItemsSlot` that you can use to cross-sell (or up-sell) products related to those in the customer's current shopping cart. This section describes how the `RelatedItemsSlot` scenario works and explains the components and elements that support it.

The `RelatedItemsSlot` scenario uses a preconfigured, active slot named `RelatedItemsOfCart`, which is provided with ATG Commerce. By default, the `RelatedItemsOfCart` slot is configured to display products from the default ATG Commerce product catalog. When using the `RelatedItemsSlot` scenario in your own commerce application, your page developers would simply configure the `RelatedItemsOfCart` slot to display products from your own product catalog and add it to any page of your application that displays the customer's shopping cart.

Examine the following figure, which illustrates the `RelatedItemsSlot` scenario. (Note that you can access the actual scenario in the Scenarios>Scenarios task area of the ACC.)


RelatedItemsSlot
Item Added
Item added to order
Remove items from slot RelatedItemsOfCart
Item Removed
Item removed from order
Remove items from slot RelatedItemsOfCart
Order Saved
Order Saved
Remove items from slot RelatedItemsOfCart
Saved Order becomes current order
Saved order becomes the current order
Remove items from slot RelatedItemsOfCart
Items Requested by Slot
Items requested by slot RelatedItemsOfCart
Fill Related items to slot RelatedItemsOfCart with property type relatedProducts

*RelatedItemsSlot scenario*

As can be seen in the preceding figure, in the `RelatedItemsSlot` scenario the system watches for any one of four events that indicate that the customer has made a change to his or her current shopping cart (order). The first four segments of the scenario define these events; they are:

- Item added to order.
- Item removed from the order.
- Order saved.
- Saved order becomes the current order.

When one of the above events occurs, the system responds by removing all of the items in the `RelatedItemsOfCart` slot.

It is the last segment of the scenario that dynamically changes the contents of the `RelatedItemsOfCart` slot. In this segment, the system watches for the `RelatedItemsOfCart` slot to request items. Because `RelatedItemsOfCart` is an active slot, it requests items whenever it is empty, and, therefore, it requests items whenever one of the above changes is made to the customer's shopping cart (because any one of those events causes the system to empty the slot).

When the `RelatedItemsOfCart` slot does request content, the `Fill Related Items to Slot` action (highlighted in yellow in the preceding figure) occurs. In this action, the system examines the customer's current shopping cart and fills the `RelatedItemsOfCart` slot with products that relate to those in the customer's cart.

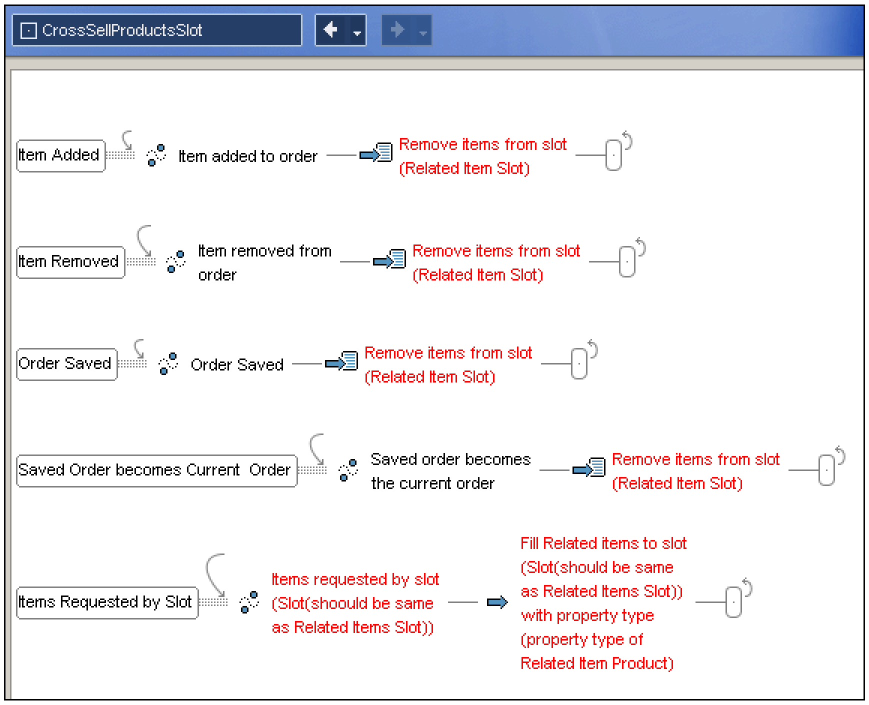
It's important to note that in the scenario, by default, the `Fill Related Items to Slot` action is configured to fill the `RelatedItemsOfCart` slot with "property type `relatedProducts`" in order to cross-sell (related) products. However, you can change this property type value to any property that holds products, such as a custom property created by your application developers for up-selling products.

**Note to Page Developers:** Motorprise, the ATG Business Commerce Reference Application, utilizes the `RelatedItemsSlot` scenario and `RelatedItemsOfCart` slot to cross-sell products on its shopping cart pages. For details on this implementation, refer to the *Commerce Scenarios* section of the *Merchandising* chapter in the *ATG Business Commerce Reference Application Guide*.

### *Creating Additional Scenarios for Cross-Selling and Up-Selling*

ATG Commerce provides you with a scenario template named `CrossSellProductsSlot` to facilitate the quick creation of additional scenarios for cross-selling and up-selling products on the shopping cart pages of your application.

The following figure illustrates the `CrossSellProductsSlot` scenario template. (Note that you can access the actual template in the Scenarios>Scenario Templates task area of the ACC.)


CrossSellProductsSlot
Item Added
Item added to order
Remove items from slot (Related Item Slot)
Item Removed
Item removed from order
Remove items from slot (Related Item Slot)
Order Saved
Order Saved
Remove items from slot (Related Item Slot)
Saved Order becomes Current Order
Saved order becomes the current order
Remove items from slot (Related Item Slot)
Items Requested by Slot
Items requested by slot (Slot(shoould be same as Related Items Slot))
Fill Related items to slot (Slot(should be same as Related Items Slot)) with property type (property type of Related Item Product)

*CrossSellProductsSlot scenario template*

You can see from the figure that the `CrossSellProductsSlot` template closely resembles the `RelatedItemsSlot` scenario. You simply replace the `(Related Item Slot)` placeholder in each segment with the appropriate slot (either `RelatedItemsOfCart` or a custom slot created by your application developers), and then you replace the `(property type of Related Item Product)` placeholder with the property value that you desire -- for example, with the `relatedProducts` property to fill the slot with products to cross-sell, or a custom property created by your application developers to fill the slot with products to up-sell.

For general information on working with scenario templates, see the *Creating Scenarios* chapter of the *ATG Personalization Guide for Business Users*.

# 8 Managing Abandoned Orders

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 ATG 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 merchants and business users responsible for creating campaigns that respond to order abandonment activity. It includes the following sections:

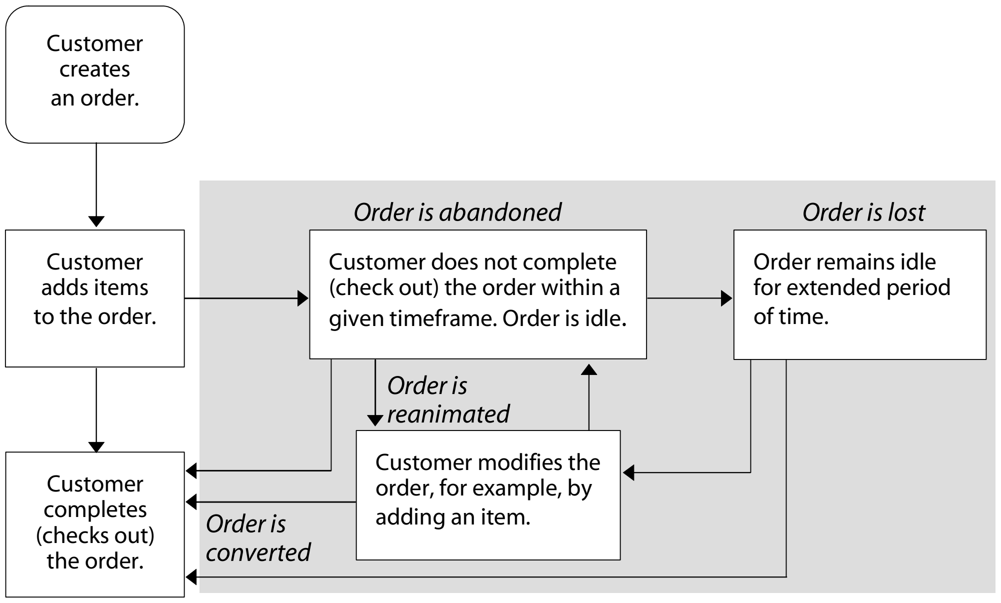
**Understanding Order Abandonment**

**Responding to Order Abandonment Activity**

**Important:** For information on related tasks that are typically performed by developers, such as configuring the module, see the *Using Abandoned Order Services* chapter in the *ATG Commerce Programming Guide*.

## Understanding Order Abandonment

Examine the following process flow diagram, which illustrates the various paths an order can take once created by a customer.


Customer creates an order.
Customer adds items to the order.
Customer completes (checks out) the order.
Order is abandoned
Customer does not complete (check out) the order within a given timeframe. Order is idle.
Order is lost
Order remains idle for extended period of time.
Order is reanimated
Customer modifies the order, for example, by adding an item.
Order is converted

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 | These are incomplete orders that have not been checked out by customers and instead have remained idle for a duration of time.<br><br>Your developers can configure the module to use whatever criteria you require for defining what orders should be considered abandoned. Out-of-the-box the following criteria can be used:<br><br>-- number of idle days<br><br>-- minimum amount (optional)<br><br>For example, the system can be configured to detect incomplete orders that have been idle for 10 days and identify them as abandoned. Alternatively, you could narrow the criteria by also specifying that the orders must cost a minimum amount of $25.00 to be identified as abandoned.<br><br>It’s important to note that while the default system supports only one type of abandoned order and only the criteria listed above, it can be configured to support multiple types of abandoned orders and additional criteria. For example, you may want the system to identify and differentiate two types of abandoned orders: high-cost and low-cost incomplete orders. This would enable you to create campaigns (scenarios, emails, and so on) that are tailored for each type.<br><br>Once you have identified your requirements with respect to abandoned orders, you should confer with your developers, who must configure the system to periodically search for and identify these orders. |
| Reanimated orders | These are previously abandoned orders that have since been modified by the customer in some way, such as adding items or changing item quantities.<br><br>For your convenience, a scenario is included out-of-the-box that watches for users to modify their previously abandoned orders and then identifies the orders as reanimated. See Responding to Order Abandonment Activity in this chapter. |
| Converted orders | These are previously abandoned orders that subsequently have been checked out by the customer.<br><br>For your convenience, a scenario is included out-of-the-box that watches for users to check out their previously abandoned orders and then identifies the orders as reanimated. See Responding to Order Abandonment Activity in this chapter. |

| | |
|---|---|
| Lost orders | These are abandoned orders that have been abandoned for so long that reanimation of the order is no longer considered realistic. The default system uses the same criteria to identify lost orders as it does abandoned orders:<br><br>-- number of idle days<br><br>-- minimum amount (optional)<br><br>For example, the system can be configured to detect incomplete orders that have been idle for 25 days and identify them as lost.<br><br>As with abandoned orders, the criteria above is supported out-of-the-box; however, your developers can configure the system to support additional criteria and multiple types of lost orders.<br><br>Once you have identified your requirements with respect to lost orders, you should confer with your developers, who must configure the system to periodically search for and identify these orders. |

Finally, note in the diagram that the process flow with respect to order abandonment activity is not always linear. For example, an order can be abandoned, then reanimated, then abandoned again. Remembering this is particularly useful when creating campaigns to entice users to return to their abandoned orders and complete them. For information on this, see the next section, Responding to Order Abandonment Activity.

## Responding to Order Abandonment Activity

Scenarios that incorporate promotions and templated email are your key tools to encourage customers to reanimate and convert their abandoned orders. Consequently, this section provides important information on creating and testing scenarios for this purpose.

**Creating Scenarios that Respond to Abandonment Activity**
Describes the out-of-the-box scenario that updates abandoned and previously abandoned orders to reflect user activity. Also includes an example scenario whose purpose is to entice users to reanimate and convert their abandoned orders.

**Testing Scenarios that Respond to Abandonment Activity**
Describes how to test abandonment-related scenarios via the Commerce Administration user interface.

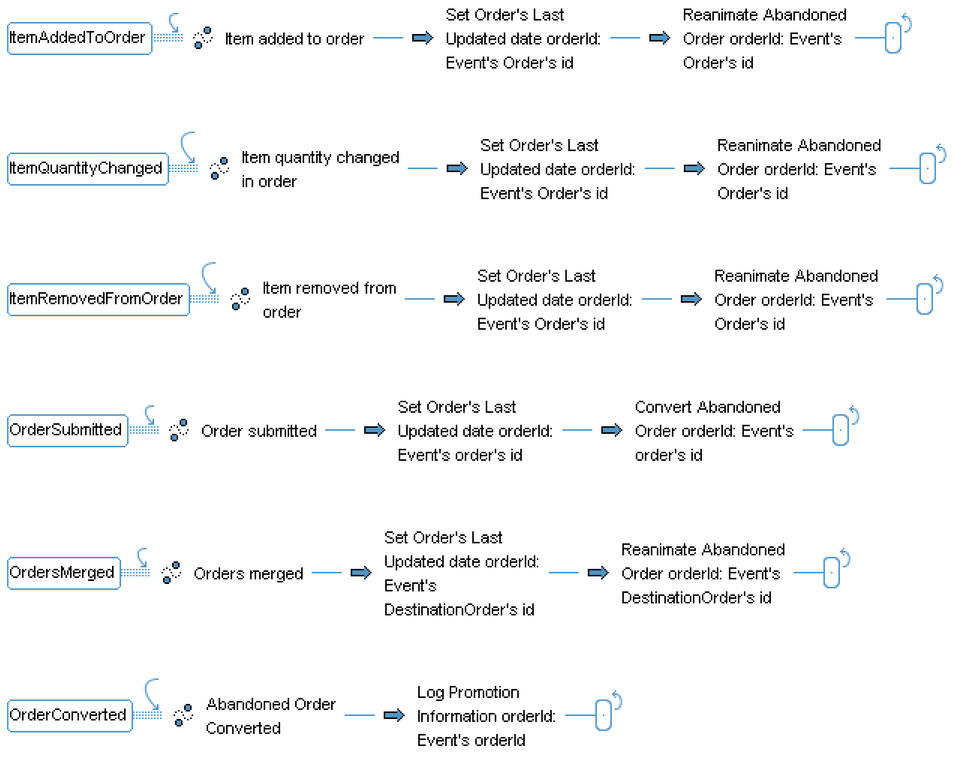
**Scenario Event Elements**
Describes the event elements that can be used in abandonment-related scenarios.

**Scenario Action Elements**
Describes the action elements that can be used in abandonment-related scenarios.

## Creating Scenarios that Respond to Abandonment Activity

If your Web sites run the Abandoned Order Services module, your developers have already configured the module to periodically search for and identify orders as abandoned and lost.

As a merchant or business user, your task is to create scenarios that respond to order abandonment activity. For your convenience a scenario that watches for user activity on abandoned and previously abandoned orders is provided for you. You can examine that scenario, named Abandoned Orders, in the Scenarios > Scenarios task area of the ATG Control Center; the scenario is located in the Abandoned Orders folder. It looks as follows:


ItemAddedToOrder
Item added to order
Set Order's Last Updated date orderId: Event's Order's id
Reanimate Abandoned Order orderId: Event's Order's id
ItemQuantityChanged
Item quantity changed in order
Set Order's Last Updated date orderId: Event's Order's id
Reanimate Abandoned Order orderId: Event's Order's id
ItemRemovedFromOrder
Item removed from order
Set Order's Last Updated date orderId: Event's Order's id
Reanimate Abandoned Order orderId: Event's Order's id
OrderSubmitted
Order submitted
Set Order's Last Updated date orderId: Event's order's id
Convert Abandoned Order orderId: Event's order's id
OrdersMerged
Orders merged
Set Order's Last Updated date orderId: Event's DestinationOrder's id
Reanimate Abandoned Order orderId: Event's DestinationOrder's id
OrderConverted
Abandoned Order Converted
Log Promotion Information orderId: Event's orderId

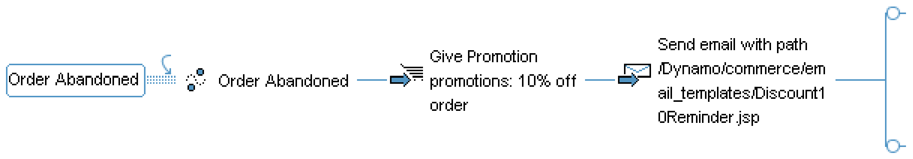
*Abandoned Orders scenario*

In the scenario the system watches for an event that indicates the customer has made a change to the current order. Possible events include:

- Item added to order
- Item quantity changed
- Item removed from the order
- Order submitted
- Orders merged

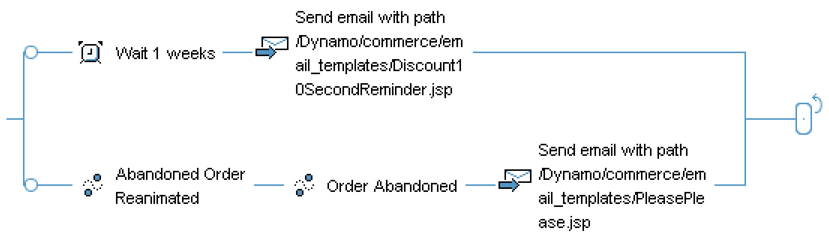
When one of the above events occurs, the system updates the order to reflect the date and time it was last updated. It then identifies the order as reanimated or converted, as appropriate. If the order is converted, it also records the order's promotion-related information for reporting purposes.

As previously mentioned, the Abandoned Orders scenario is provided as a convenience for you. As such, your remaining task is to design and create scenarios that watch for abandoned and lost orders and encourage customers to reanimate and convert them. The following *hypothetical* scenario is provided as an example:


Order Abandoned
Order Abandoned
Give Promotion promotions: 10% off order
Send email with path /Dynamo/commerce/email_templates/Discount10Reminder.jsp

In the first part of the scenario, the system watches for an order to be identified as abandoned. When this event occurs, the system grants the customer a promotion that offers a 10% discount if an order is placed and notifies the customer of this promotion via an email, `Discount10Reminder.jsp`.

The remainder of the scenario looks as follows:


Wait 1 weeks
Send email with path /Dynamo/commerce/email_templates/Discount10SecondReminder.jsp
Abandoned Order Reanimated
Order Abandoned
Send email with path /Dynamo/commerce/email_templates/PleasePlease.jsp

In the remainder of this scenario, one of two branches can succeed:

- If the abandoned order remains idle for one week, a second email message is sent to remind the customer of the promotion, `Discount10SecondReminder.jsp`.
- If the customer reanimates the order and then re-abandons it, a different email message is sent, `PleasePlease.jsp`.

Because the scenario is configured such that one only branch can succeed, the customer is sent no more than two messages for any one abandoned order. (Recall that this scenario is hypothetical and is not included in your installation.)

Finally, it's important to note that users' profiles have two abandonment-related properties that you may want to utilize in scenarios:

- `abandonedOrderCount`, which stores the number of abandoned orders currently associated with the user.
- `abandonedOrders`, which stores the list of `abandonedOrder` items currently associated with the user. An item of this type stores the ID of the abandoned order.

These properties enable you to create an even richer set of abandonment-related scenarios. For example, you could create a scenario that watches for a user to log in and then, if the user's `abandonmentCount` profile property is equal to or greater than 1, grants a "Free Shipping on Orders Purchased Today" promotion to the user.

## Testing Scenarios that Respond to Abandonment Activity

As you create scenarios that respond to abandonment activity, you can test them via the *Abandoned Order Messages* page of the Commerce Administration UI. The interface enables you to manually assign a specific abandonment state to one or more orders, which causes the system to fire the appropriate scenario event (Order Abandoned, Abandoned Order Reanimated, and so on). As such, you can force orders through the various segments of your scenarios in order to test them.

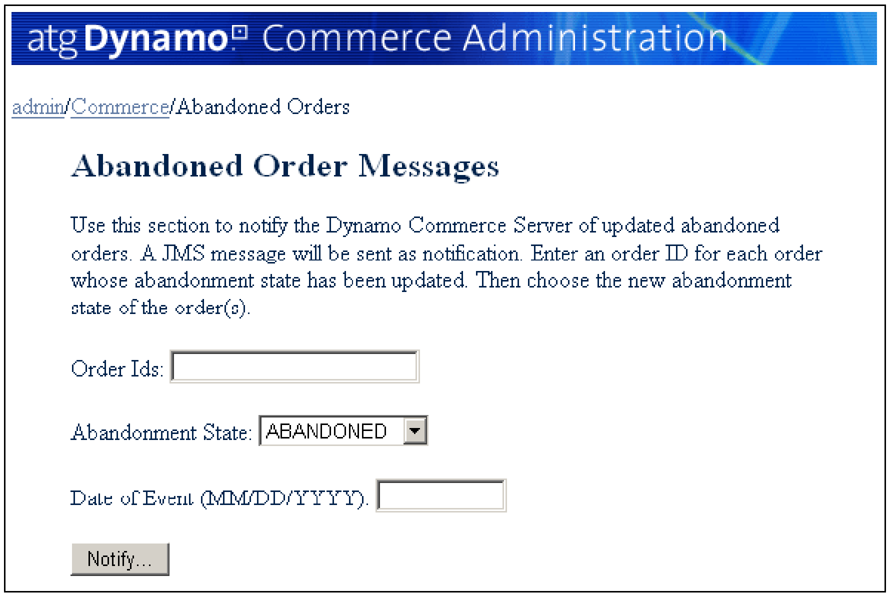
To test your abandonment scenarios on one or more orders, do the following:

1. Create one or more orders to use as testing data.
2. Access the ATG Dynamo Server Administration UI by pointing your browser to the link appropriate for your application server. For example, JBoss users use this URL by default:

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

   **Note**: During application assembly, you must specify the Dynamo Administration UI module in order to access this UI. See the *ATG Programming Guide*.
3. Log into the UI with your username and password. The default username/password is `admin/admin`.
4. Click the **Commerce Administration** link on the main *Dynamo Administration* page.
5. Click the **Abandoned Order Administration** link on the *Dynamo Commerce Administration* page.

   The system displays the *Abandoned Order Messages* page, which is shown in the following figure:



admin/Commerce/Abandoned Orders

Abandoned Order Messages

Use this section to notify the Dynamo Commerce Server of updated abandoned orders. A JMS message will be sent as notification. Enter an order ID for each order whose abandonment state has been updated. Then choose the new abandonment state of the order(s).

Order Ids:

Abandonment State: ABANDONED

Date of Event (MM/DD/YYYY):

*Abandoned Order Messages page of the Commerce Administration UI*

**6.** Enter the appropriate information:

- In the Order Ids field, enter the IDs of the orders whose abandonment states you want to change.
- In the Abandonment State drop-down list, select the abandonment state.
- In the Date of Event field, enter the date. Typically, the date to use is the current date. However, if the scenario utilizes a time element (for example, “wait 2 weeks”), you can enter a future date to advance the scenario beyond that element.
- Click the **Notify…** button.

## Scenario Event Elements

The following event elements can be used in scenarios that must watch for abandoned order activity:

**Order Abandoned**

**Abandoned Order Converted**

**Abandoned Order Lost**

**Abandoned Order Reanimated**

### *Order Abandoned*

The system watches for an order to be identified as abandoned.

While you can use the optional parameters within this element to further define the type of order to watch for, typically no optional parameters are required.

### *Abandoned Order Reanimated*

The system watches for an order to be identified as reanimated.

While you can use the optional parameters within this element to further define the type of order to watch for, typically no optional parameters are required.

### *Abandoned Order Converted*

The system watches for an order to be identified as converted.

While you can use the optional parameters within this element to further define the type of order to watch for, typically no optional parameters are required.

### *Abandoned Order Lost*

The system watches for an order to be identified as lost.

While you can use the optional parameters within this element to further define the type of order to watch for, typically no optional parameters are required.

## Scenario Action Elements

The following action elements can be used in scenarios that must respond to abandoned order activity:

**Set Order's Last Updated Date**

**Reanimate Abandoned Order**

**Convert Abandoned Order**

**Log Promotion Information**

### *Set Order's Last Updated Date*

Updates an order to reflect the date and time it was most recently modified by the owner.

Use this action element to update the order to reflect the fact that user activity has occurred. User activity can include adding items, removing items, changing item quantities, merging orders, and checking out orders.

Examples:

```
Set Order's Last Updated date orderId: Event's Order's id

Set Order's Last Updated date orderId: Event's DestinationOrder's
id
```

Use the second example when the action is invoked as a result of the merging of two orders. In this case, the action should update the date and time of the destination order, not the source order.

### *Reanimate Abandoned Order*

Identifies an abandoned order as reanimated.

Use this action element when an abandoned order has been modified and should, therefore, be reanimated. Modifications can include adding items, removing items, changing item quantities, merging orders, and checking out orders.

Examples:

```
Reanimate Abandoned Order orderId: Event's Order's id

Reanimate Abandoned Order orderId: Event's DestinationOrder's id
```

Use the second example when the action is invoked as a result of the merging of two orders. In this case, the action should reanimate the destination order, not the source order.

### *Convert Abandoned Order*

Identifies a previously abandoned order as converted.

Use this action element when a previously abandoned order has been checked out by the owner and, therefore, should be identified as converted.

Example:

```
Convert Abandoned Order orderId: Event's Order's id
```

### *Log Promotion Information*

Calculates and records the number and total value of the promotions applied to a converted order.

Use this action element when a previously abandoned order has been converted, and you want to log its promotion-related information.

Example:

```
Log Promotion Information orderId: Event's Order's id
```

# 9 Catalog Navigation and Searching

A commerce site must supply mechanisms for customers to navigate through the site and find the products they want to buy. ATG Commerce includes services that you can use to implement navigational and searching mechanisms on your sites.

This chapter includes the following sections:

**Using the parentCategory Property**

**Displaying Catalog Items**

**Catalog Navigation**

**Catalog Searching**

This chapter covers only components specific to the ATG Commerce native search capabilities, which are based on the `atg.commerce.catalog.SearchFormHandler`; this is not the same as the ATG Search product. For general information on components used in searching, see the *Using Search Forms* chapter in the *ATG Page Developer's Guide*. For information on ATG Search, including how to use it as part of your ATG Commerce site, see the *ATG Search Installation and Configuration Guide* and the *ATG Search Administration Guide*.

## Using the parentCategory Property

A category or product can be the child of more than one category. Specifying multiple parent categories makes the catalog more flexible, but can complicate navigation. This is especially true if the customer accesses a category or product through a search facility rather than by traversing the catalog hierarchy; if the customer then wants to move up the hierarchy, you need to determine which parent category to move to. You can use the `parentCategory` property of the category and product items to specify a default parent category for this purpose. A product can have different `parentCategories` for each catalog in which the product appears.

For example, suppose you have a link on each page that takes the customer up one level in the catalog hierarchy. If customer views a product that has multiple parent categories, your sites can track which parent category the customer accessed the product from, and make this link point back to that category. But if the customer finds the product by searching rather than navigating through the catalog hierarchy, you can have the link point to the category specified by the `parentCategory` property.

If your catalog uses a different name for the `parentCategory` property, set the `parentCategoryPropertyName` property in the

`/atg/commerce/catalog/CatalogTools.properties` file to the actual name of the property. For example:

```
parentCategoryPropertyName=higherCategory
```

For more information on properties of catalogs, categories, and products, and on how to extend the product catalog, see the *ATG Commerce Programming Guide*.

## Displaying Catalog Items

The components described in this section allow you to find items in the product catalog and display them for your users.

### Looking Up Items in the Catalog

Use the `atg.commerce.catalog.custom.CatalogItemLookupDroplet` class to locate and display items in a repository. This ATG servlet bean takes input parameters specifying the repository, repository ID, site ID, site scope, catalog, and item type, and renders the specified item on the page.

Rather than specifying the repository and item type through input parameters, you can set these through the servlet bean's properties file. ATG Commerce includes several lookup components that are configured to use the product catalog as the default repository and find a specific item type. The following three components are all based on the `CatalogItemLookupDroplet` and are found in `/atg/commerce/catalog/`:

- `CategoryLookup`
- `ProductLookup`
- `SKULookup`

If your sites include more than one catalog, or if your catalog uses item types not found in the catalog, you may want to create additional `CatalogItemLookupDroplet` components. For more information about the `CatalogItemLookupDroplet` servlet bean, see CatalogItemLookupDroplet in Appendix: ATG Commerce Servlet Beans.

The following additional lookup components in `/atg/commerce/catalog/` are based on the `ItemLookupDroplet`, which does not include catalog or site filtering capabilities.

- `MediaLookup`
- `CatalogLookup`

See the *ATG Page Developer's Guide* for information on the `ItemLookupDroplet`. This droplet is also used by the gift list feature; see *Setting Up Gift Lists and Wish Lists* in the *ATG Commerce Programming Guide*.

The following example shows a portion of a JSP that uses the `ProductLookup` component to display the current product. The product's repository ID is passed to this page (via the `itemId` parameter) from the page that links to it:

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

<dsp:oparam name="output">
<p><b><dsp:valueof param="element.displayName"/></b>
<p><dsp:getvalueof id="img13" param="element.largeImage.url"
        idtype="java.lang.String">
<dsp:img src="<%=img13%>"/>
</dsp:getvalueof>
<dsp:valueof param="element.longDescription"/>
</dsp:oparam>
</dsp:droplet>
```

## ForEachItemInCatalog Servlet Bean

This servlet bean is identical to the commonly used `ForEach` bean, except that it only iterates over items that exist in the user's current catalog (for information on the `ForEach` servlet bean, see the *ATG Page Developer's Guide*). It includes an additional optional parameter, `profile`. The user's profile stores information on which catalog that user can view. If you do not supply a profile, the current session-scoped profile is used instead.

Use this servlet bean for lists of items that might not necessarily be in the current catalog, such as `relatedProducts`, `relatedCategories`, and `replacementSkus`.

## Sending Messages When Items are Viewed

You can use the `atg.userprofiling.ViewItemEventSender` servlet bean to send JMS messages when the customer views items in the catalog. `ViewItemEventSender` sends a JMS object message of class `atg.userprofiling.dms.ViewItemMessage`. The `ViewItemMessage` object identifies the repository item being viewed and the item's location. These messages can be used to trigger actions. For example, a message listener could be configured to store information in the customer's profile about the products viewed.

The `ViewItemEventSender` takes a single input parameter called `eventobject` that specifies the item viewed. There are no output parameters or open parameters.

ATG Commerce includes two `ViewItemEventSender` components in `/atg/commerce/catalog`: `ProductBrowsed` and `CategoryBrowsed`. Depending on which one you use, the `eventobject` passed in is either the product or category repository item.

The following example shows a portion of a JSP that uses the `ProductBrowsed` component to send a message when a product is viewed. The product's repository ID is passed to this page (via the `itemId` parameter) from the page that links to it:

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

```
<dsp:oparam name="output">
  <dsp:droplet name="/atg/commerce/catalog/ProductBrowsed">
    <dsp:param param="element" name="eventobject"/>
  </dsp:droplet>
</dsp:oparam>
</dsp:droplet>
```

## Catalog Navigation

If your catalog has a hierarchical structure, you can set up your sites so that customers can navigate to products by traversing the catalog hierarchy. The structure of the catalog hierarchy is determined by the child categories and child products of each category. For example, in a grocery store site, the user might get to a product called Oranges by first selecting the Fruit category, then selecting the Citrus Fruit category (which is a child category of Fruit), and then selecting the Oranges product (which is a child product of Citrus Fruit).

### Displaying Root Categories

Typically, a catalog home page displays a list of root categories. Unlike other categories, root categories cannot be found through the `childCategories` property of other categories. Root categories are those that appear in the `allRootCategories` property of the user's catalog.

The following example uses a `ForEach` servlet bean with the `allRootCategories` property to find the root categories.

```
<HTML> <HEAD>
<TITLE>Home Page</TITLE>
</HEAD>

<BODY BGCOLOR="#FFFFFF">
<H1>Home Page</H1>
 <dsp:droplet name="/atg/dynamo/droplet/ForEach">
   <dsp:param bean="/atg/userprofiling/Profile.catalog.allRootCategories"
        name="array"/>
  <dsp:oparam name="output">
    <tr>
      <td>
      <dsp:getvalueof id="a26" param="element.template.url"
           idtype="java.lang.String">
<dsp:a href="<%=a26%>">
</dsp:getvalueof>
      <dsp:param param="element.repositoryId" name="id"/>
      <dsp:param value="pop" name="navAction"/>
      <dsp:param param="element" name="Item"/>
      <dsp:valueof param="element.displayName"/></dsp:a></td>
    </tr>
  </dsp:oparam>

  <dsp:oparam name="empty">
    <p>No root categories found.
   </dsp:oparam>
 </dsp:droplet>

</BODY> </HTML>
```

## Displaying Child Categories and Products

You can use servlet beans in your catalog pages to display a list of all of the child categories and child products of a category. For example, the grocery store’s Fruit page might list many different fruit products (Apples, Pears), as well as child categories (Citrus Fruit). When a user clicks on the name of a product on this page, your sites display that product. When a user clicks on the name of a category, your sites display a list of that category’s child categories and child products.

The following example shows a portion of a JSP that renders the `displayName` property of each child category and child product of the current category. Each of these values is rendered as a link to the corresponding item. The current category’s repository ID is passed to this page (via the `itemId` parameter) from the page that links to it:

```
<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.childCategories" name="array"/>
     <dsp:oparam name="outputStart">
     <p><b>Child Categories:</b>
     <ul>
     </dsp:oparam>
     <dsp:oparam name="output">
     <li><dsp:getvalueof id="a24" param="element.template.url"
              idtype="java.lang.String">
<dsp:a href="<%=a24%>">
     <dsp:valueof param="element.displayName"/>
     <dsp:param param="element.repositoryId" name="itemId"/>
     </dsp:a></dsp:getvalueof></li>
     </dsp:oparam>
     <dsp:oparam name="outputEnd">
     </ul>
     </dsp:oparam>
   </dsp:droplet>

   <dsp:droplet name="/atg/commerce/catalog/ForEachItemInCatalog">
     <dsp:param param="element.childProducts" name="array"/>
     <dsp:oparam name="outputStart">
     <p><b>Child Products:</b>
     <ul>
     </dsp:oparam>
     <dsp:oparam name="output">
     <li><dsp:getvalueof id="a61" param="element.template.url"
              idtype="java.lang.String">
<dsp:a href="<%=a61%>">
     <dsp:valueof param="element.displayName"/>
     <dsp:param param="element.repositoryId" name="itemId"/>
     </dsp:a></dsp:getvalueof></li>
     </dsp:oparam>
     <dsp:oparam name="outputEnd">
     </ul>
     </dsp:oparam>
   </dsp:droplet>
</dsp:oparam>
</dsp:droplet>
```

## Historical Navigation

Although a catalog is usually structured hierarchically, the hierarchical structure is not rigid. A category or product can be the child of more than one category, so there can be multiple paths to any catalog item. In addition, your pages may have links to related products that enable users to jump from one part of the hierarchy to another, and other navigational aids such as search facilities.

ATG Commerce provides components that you can use to keep track of the path a customer follows when moving around your sites. This tracking method, called historical navigation or "breadcrumbs," enables you to construct a list of the items the customer has visited, and create and display links to these items, so the customer can easily get back to them.

The `atg.commerce.catalog.CatalogNavHistory` class tracks the customer's path through the catalog. `CatalogNavHistory` is a subclass of `atg.repository.servlet.NavHistory`, which can be used to track navigation history over any set of pages that display repository content. ATG Commerce includes a session-scoped `CatalogNavHistory` component at `/atg/commerce/catalog/CatalogNavHistory`.

`CatalogNavHistory` maintains a stack of locations the customer has visited, consisting of the actual repository items the customer viewed. It stores these locations (items) in the `navHistory` property.

`CatalogNavHistory` tracks the customer's path across all possible actions:

- When the customer navigates down the hierarchy by clicking a navigational link, an item is pushed onto the stack.
- When the customer navigates up the hierarchy by clicking a navigational link or the Back button, the appropriate item is removed from the stack, and the new item is pushed onto the stack.

- When the customer jumps to an unrelated area of the site, whether by a global navigation or some other link, the path jumps to that area. If there is a default navigation path to the new page, as in a hierarchical structure, `CatalogNavHistory` uses that default path.

To add and remove items in the `navHistory` array, use the `CatalogNavHistoryCollector` servlet bean, which is of class `atg.repository.servlet.NavHistoryCollector`. This servlet bean takes the following input parameters:

- `item` - The repository item currently being viewed.
- `navAction` - The operation to be performed on the stack. Options are push, pop, and jump. A blank `navAction` is treated as push.
- `navCount` - Used to detect use of the Back button. The `navCount` parameter should be passed in with any link that leads to a page that uses breadcrumbs. It must be embedded in a link to enable the target page to detect Back button usage. For example:

```
<dsp:getvalueof id="templateUrl" idtype="String"
     param="element.template.url">
  <dsp:a page="<%=templateUrl%>">
    <dsp:param name="id" param="element.repositoryId"/>
    <dsp:param name="navAction" value="push"/>
    <dsp:param name="navCount"
         bean="/atg/commerce/catalog/CatalogNavHistory.navCount"/>
    <dsp:valueof param="element.displayName"/>
  </dsp:a>
</dsp:getvalueof>
```

  The `navCount` parameter can be used to prevent errors in the `navHistory` caused by use of the Back button. The `navCount` parameter can be used to detect when the Back button is used and the `CatalogNavHistoryCollector` resets the stack appropriately.

  For example, if you open a page that displays links to categories, each category link should have an embedded `navCount` parameter with `navCount=1`. If you browse the catalog for a while and then use the Back button to return to the original page, you are given the page that was cached by your browser. The page isn't rendered again by ATG Commerce. The category links in that page still have `navCount=1`. If you click one of those links, the `CatalogNavHistoryCollector` in the target page compares the stale `navCount` page parameter that was passed with the link to the `currentvalue` of `/atg/commerce/catalog/CatalogNavHistory.navCount`. These values won't match because the parameter from the link is stale, so the `CatalogNavHistoryCollector` knows that the Back button was used and rebuilds the `navHistory`.

### *Collecting the Customer's History*

Use the `navAction` push operation to collect a user's navigation history. In the first part of our example, below, the JSP code adds an item to the `navHistory` stack when the customer clicks the link to `CategoryPage.jsp`, permitting you to track their path.

```
<dsp:droplet name="/atg/dynamo/droplet/ForEach">
   <dsp:param bean="/atg/userprofiling/Profile.catalog.allRootCategories"
        name="array"/>
  <dsp:oparam name="output">
  <p>
  <!
    Here we set up the navCount and navAction parameters
    which will indicate to the next page how it should update its
    CatalogNavHistory collector. In particular, to set the navCount,
    we query the CatalogNavHistory component, asking it for a number
    that identifies its current state.
  !>
  %>
  <dsp:a href="CategoryPage.jsp">
   <dsp:valueof param="element.displayName"/>
   <dsp:param param="element.repositoryId" name="itemId"/>
   <dsp:param bean="/atg/commerce/catalog/CatalogNavHistory.navCount"
        name="navCount"/>
   <dsp:param value="push" name="navAction"/>
  </dsp:a>
 </dsp:oparam>
</dsp:droplet>
```

### *Rendering the Customer's Path*

This second half of our example shows how to render a list of the locations the customer has visited. First, update `CatalogNavHistory` with the current location, using the `category`, `navAction`, and `navCount` parameters passed to the page that embeds this fragment, along with the `CatalogNavHistoryCollector` component, which updates `CatalogNavHistory`.

If the Back button is detected, `CatalogNavHistory` clears the stack and generates a new one for the current page, using the default parent property of the catalog.

```
<dsp:droplet name="/atg/commerce/catalog/CatalogNavHistoryCollector">
  <dsp:param param="categoryObj" name="item"/>
  <dsp:param param="navAction" name="navAction"/>
  <dsp:param param="navCount" name="navCount"/>
</dsp:droplet>
```

Then use a page fragment such as the following to display the user's trail. The servlet bean lists the names of each category held by the `CatalogNavHistory` component and creates a link to the Category page that displays them. The following parameters are passed:

- `itemId` - holds the category `repositoryId`
- `navAction` - set with the value of pop, indicating that the `CatalogNavHistory` should clear its stack and rebuild it using the default parent category property

- `navCount` - holds the index for each category element from the `CatalogNavHistory.navHistory` array

The `pageType` parameter indicates what type of page is calling the servlet bean; if a category page, the last category entry in `CatelogNavHistory` is not made into a link. If the servlet bean is called from a product page, the last entry is made into a link, because it represents the product's parent category.

```
<dsp:droplet name="/atg/dynamo/droplet/ForEach">
 <dsp:param bean="/atg/commerce/catalog/CatalogNavHistory.navHistory"
           name="array"/>
 <dsp:oparam name="output">
  <dsp:droplet name="/atg/dynamo/droplet/Switch">
   <dsp:param param="count" name="value"/>
   <dsp:getvalueof id="nameval1" param="size" idtype="java.lang.String">
   <dsp:oparam name="<%=nameval1%>">
       <dsp:droplet name="/atg/dynamo/droplet/Switch">
        <dsp:param param="pageType" name="value"/>
        <dsp:oparam name="product">
         <dsp:a href="CategoryPage.jsp">
          <dsp:param param="element.repositoryId" name="itemId"/>
          <dsp:param value="pop" name="navAction"/>
          <dsp:param param="index" name="navCount"/>
          <dsp:valueof param="element.displayName"/>
         </dsp:a>
        </dsp:oparam>
        <dsp:oparam name="default">
         <dsp:valueof param="element.displayName"/>
        </dsp:oparam>
       </dsp:droplet>
      </dsp:oparam>
   </dsp:getvalueof>
   <dsp:oparam name="default">
    <dsp:a href="CategoryPage.jsp">
     <dsp:param param="element.repositoryId" name="itemId"/>
     <dsp:param value="pop" name="navAction"/>
     <dsp:param param="index" name="navCount"/>
     <dsp:valueof param="element.displayName"/>
    </dsp:a>
    &nbsp;&gt;&nbsp;
   </dsp:oparam>
  </dsp:droplet>
 </dsp:oparam>
</dsp:droplet>
```

### *Tracking Location Jumps*

This example shows how to track customer navigation outside of normal history collection, such as with a hard-coded link that deviates from hierarchical navigation.

```
<a href="param:featuredProduct.template.url">
  <param name="id" value="param:featuredProduct.repositoryId">
  <param name="navAction" value="jump">
  <param name="navCount"
    value="property:/atg/commerce/catalog/CatalogNavHistory.navCount">
     link text
</a>
```

## Catalog Searching

You can provide searching mechanisms to enable customers to find products that satisfy a set of criteria. This section includes the following information related to searching catalogs:

Overview of Catalog Searching

Preconfigured Catalog Search Components

Configuring the Search Form Handler

Configuring Catalog Search Types

Combining Catalog Search Types

Processing Searches

Displaying Search Results

Searching Catalogs in Preview Mode

Using Search Form Handlers with Internationalized Catalogs

For additional search information, see the *Using Search Forms* chapter in the *ATG Page Developer's Guide*.

### Overview of Catalog Searching

ATG Commerce provides the form handler class `atg.commerce.catalog.custom.CatalogSearchFormHandler` to search the catalog repository for items such as products and categories. You can use this form handler to construct searches for one or more catalog item types, and you can specify which properties to examine when searching.

For example, you can search for products that contain a specified substring in their names, or for categories that are tagged with a specified keyword, or for both products and categories that have a specified set of property values.

Configuration settings in the form handler's properties file specify the kinds of elements to search for, the properties of those elements to consider when searching, and additional configuration details for each type of search. Typically, customers specify target values to search for through form fields in a JSP.

The `CatalogSearchFormHandler` class should provide sufficient search functionality and flexibility for most Commerce applications. ATG Commerce includes several `CatalogSearchFormHandler` components in `/atg/commerce/catalog`, each one configured for a different set of search options (see Preconfigured Catalog Search Components). You can create additional `CatalogSearchFormHandler` components and configure them through their properties files or through the Component Editor in the ACC. If your store requires custom search capabilities, you can extend `CatalogSearchFormHandler` or write another form handler.

`CatalogSearchFormHandler` can be configured to perform four types of searching:

**Keyword Searches**
Keyword searches use keyword property names and input search strings to search product and category keywords. The customer enters values that are used for keyword matching. An example of a keyword search is "Show me all products and categories with the keyword shoe."

**Text Searches**
Full-text searches use text property names and input search strings to perform text pattern matching on properties. An example of a full-text search is "Show me all products whose `longDescription` property contains the word `wool`." Note that your database must be configured properly to support full-text searches. For more information, see the discussion on databases and database access in the *ATG Installation and Configuration Guide*.

**Hierarchical Searches**
Hierarchical searches look in a subset of categories, starting from a given category, and including that category's child categories, the children of those children, and so on. The given category is indicated by the repository ID in the `hierarchicalCategoryId` property. To perform hierarchical searches, you must generate the `ancestorCategories` property for each product and category item, as described in the *ATG Commerce Programming Guide*.

**Advanced Searches**
Advanced searches provide possible search options for each property specified in the form handler's `advancedSearchPropertyNames` property. For example, enumerated types are defined in the repository with a set number of values. Advanced searches retrieve these values from the definition to display in a selection box. The advanced query is built from options selected by the customer to further refine the catalog search. For example, advanced searches allow a customer to search on a description, manufacturer, or price. An example of an advanced search is "Show me all products with the keyword shoe where price range is expensive."

## Preconfigured Catalog Search Components

ATG Commerce includes five preconfigured instances of `CatalogSearchFormHandler` in `/atg/commerce/catalog/custom`:

- `CatalogSearch` searches keywords, descriptions, and display names, and finds matching products and categories.
- `CategorySearch` searches keywords and descriptions and finds matching categories only.
- `ProductSearch` searches keywords and descriptions and finds matching products only.

- `ProductTextSearch` searches only description fields, not keywords, and finds matching products only.
- `AdvProductSearch` combines keyword and text searching with hierarchical and advanced searching, and finds products that match all search criteria.

To see how these components are configured, open them in the ACC Component Editor and view their property settings.

**Note:** When running on a preview server, mode, the five preconfigured search components are instances of `FilteringSearchFormHandler` in `atg/commerce/catalog/custom`. For more information, see Searching Catalogs in Preview Mode.

## Configuring the Search Form Handler

`CatalogSearchFormHandler` can search for any type of repository item in the catalog repository. You specify the item types to find by setting the `itemTypes` property in the form handler's properties file to a list of strings, each naming one item type.

Item types typically include `category` or `product`, but you can configure search form handlers to search for SKUs or for custom category or product subtypes you have created. You can also create multiple instances of `CatalogSearchFormHandler` and configure them to search for different kinds of objects. For example, you could have one form handler that searches only for clothing products and another that searches for all products and categories; a clothing search page might use the first form handler, while a more general search page would use the second.

In addition to specifying the item types to search for, you must also set the property `catalogTools` so that it refers to a `CatalogTools` object that provides access to categories, products, SKUs, and other catalog information. Unless you have implemented your own catalog management system, you should use the default `CatalogTools` component at `/atg/commerce/catalog/CatalogTools`.

The following is an example of a properties file for a `CatalogSearchFormHandler` component that can perform all four types of searching:

```
$class=atg.commerce.catalog.custom.CatalogSearchFormHandler
$scope=session

doKeywordSearch=true
keywordsPropertyNames=keywords

doTextSearch=true
textSearchPropertyNames=description,displayName

doHierarchicalSearch=true
ancestorCategoriesPropertyName=ancestorCategories

doAdvancedSearch=true
advancedSearchPropertyNames=weightRange,manufacturer,childSKUs
```

```
catalog^=/atg/commerce/catalog/custom/CatalogTools.catalog
itemTypes^=/atg/commerce/customCatalogTools.productItemTypes
```

Because there are so many possible combinations of search options, it is convenient to have several `CatalogSearchFormHandler` components, each configured differently. ATG Commerce includes five different `CatalogSearchFormHandler` components (see Preconfigured Catalog Search Components), and you can create additional instances. You can also change the behavior of a `CatalogSearchFormHandler` component for an individual page by setting hidden input fields. For example, suppose your Commerce application has a `CatalogSearchFormHandler` component named `KeywordSearch` that is configured only for keyword searching, but you have one page where you want to enable text searching as well. You could use this component and just enable keyword searching on the page by including this tag:

```
<dsp:input type="hidden" bean="KeywordSearch.doTextSearch" value="true">
```

## Configuring Catalog Search Types

As mentioned previously, `CatalogSearchFormHandler` provides four different types of searching:

- keyword searches
- text searches
- hierarchical searches
- advanced searches

This section describes how to configure a `SearchFormHandler` component for each of these types of searching.

### *Keyword Searches*

To enable keyword searches in the form handler, set the property `doKeywordSearch` to true. You can specify the target values to search for by setting the `keywords` property of the search form handler to an array of strings, or by setting the `searchInput` property to a string containing one or more words separated by spaces. These values are typically specified by the customer through a form input field. (You can change the separator character used to parse `searchInput` by setting the form handler's `keywordInputSeparator` property.)

You can force the search form handler to convert all keyword inputs to uppercase before searching by setting the `toUpperCaseKeywords` property to `true`. Similarly, you can force the form handler to convert keyword inputs to lowercase by setting `toLowerCaseKeywords` to `true` instead.

By default, keyword searches look at the `keywords` property of each catalog item. You can override the default behavior by setting the `keywordsPropertyNames` property of the form handler in its properties file. You can specify one or more properties to consider in keyword searches, and each of these properties can be either single-valued or multi-valued.

Keyword searches treat single-valued and multi-valued properties differently. If a property specified in `keywordsPropertyNames` is single-valued (e.g., the property is of type String), the keyword search algorithm uses the `QueryBuilder.CONTAINS` query to examine each target value and see if it appears anywhere within the property's current value. For example, if you have a `keywordString` property whose type is String, and you search for the values `red`, `green`, and `blue` using keyword search, the resulting query is:

```
keywordString CONTAINS "red"
OR keywordString CONTAINS "green"
OR keywordString CONTAINS "blue"
```

Since CONTAINS performs a substring match, this query returns `true` for an item whose `keywordString` has the value `reduced calorie`, because `reduced` contains the string `red` within it.

However, if a property specified in `keywordsPropertyNames` is multi-valued (for example, the property is of type `String[]`), the keyword search algorithm uses the `QueryBuilder.INCLUDESANY` query to perform a single search for an exact match between any value in the property and any value in the set of search criteria. For example, if you have a `keywords` property whose type is `String[]`, and you search for the values `red`, `green`, and `blue` using keyword search, the resulting query is:

```
keywords INCLUDES ANY ["red","green","blue"]
```

Since INCLUDES ANY searches for an exact match, this query returns `false` for an item whose keywords are `diet` and `reduced calorie`, because `red` is not an exact match for `reduced calorie`.

If you specify multiple properties in `keywordsPropertyNames`, the keyword search generates a query for each property, then combines these queries using the OR operator. This means that if any one of the queries returns `true`, the item is returned by the search operation.

### *Text Searches*

To enable text searches in the form handler, set the property `doTextSearch` to true. The target search string is specified by setting the form handler's `searchInput` property, typically by the customer entering the value in a form input field. Specify which properties to examine by setting the `textSearchPropertyNames` property of the form handler. If this property is not set, text searches use a default set of properties that is defined by the repository.

The implementation of text searching is RDBMS-specific and uses the database's text searching facility, if there is one. If your database supports a full-text search engine, you must configure the search engine properly, and set the repository component's `simulateTextSearchQueries` property to `false`. If a full-text search engine is not available (either because your RDBMS does not support one, or because you do not have a license for the one your RDBMS supports), the SQL repository can simulate full-text searching by using the LIKE operator to determine whether the target value is a substring of any of the text properties being examined. To enable this feature, set the repository component's `simulateTextSearchQueries` property to `true`. Note that although simulated full-text searching is useful for development purposes, performance is unlikely to be adequate for a production server.

By default, the product catalog's `simulateTextSearchQueries` is set to `true` to support full-text searching on the SOLID database that is included with ATG products. For more information about configuring your catalog for full-text searching, see the *ATG Commerce Programming Guide*

### *Hierarchical Searches*

To enable hierarchical searches in the form handler, set the `doHierarchicalSearch` property to `true`. You specify the category from which to start the search by setting the form handler's `hierarchicalCategoryId` property to the repository ID of the category whose descendants you want to find. This property is typically set through a hidden input tag, or specified by the customer through a form input field.

Hierarchical searching requires that each category and product item have a multi-valued property whose value is a complete list of all of its ancestor categories. Hierarchical searching restricts the search results to items whose ancestor categories include the category specified through `hierarchicalCategoryId`.

Specify the name of the ancestor category property by setting the `ancestorCategoriesPropertyName` property of the form handler. Each category and product item has an `ancestorCategories` property whose value is a Set of that item's ancestor categories. The values of this property can be generated automatically using the component `/atg/commerce/catalog/custom/AncestorGeneratorService`. See *Using the Catalog Maintenance System* in the *ATG Commerce Programming Guide* for more information on `AncestorGeneratorService`.

### *Advanced Searches*

To enable advanced searches, set the form handler's `doAdvancedSearch` property to `true`. You then specify the set of properties to search by setting the `advancedSearchPropertyNames` property. Advanced searches are limited to the set of properties you name here.

Target values are specified for one or more of these properties by adding values to the `propertyValues` property of the form handler, typically through form input fields. This property is a Dictionary to which you add one key/value pair for each property you want to search. The key is the property name, and the value to search for. For example, to look for items whose `color` property is set to `red`, set `CatalogSearchFormHandler.propertyValues.color` to `red`. Setting a value to an empty string omits it from the search, and therefore specifies that the property matches any value.

For each property specified in `propertyValues`, a query is generated based on whether the property is single-valued or multi-valued. For single-valued properties a simple equality test is used. For multi-valued properties an INCLUDES test is generated, so that the query succeeds if any of the property's values match the target value. If you specify multiple properties, the queries are combined using the AND operator, so all properties must match for the catalog item to be selected.

For example, searching `color` for a value of `red` and `availableSizes` for a value of `medium`, where `color` is a single String and `availableSizes` is an array of Strings, results in the following query:

```
(color = red) AND (availableSizes INCLUDES medium)
```

`CatalogSearchFormHandler` has a property called `propertyValuesByType,` which is a Dictionary containing one key/value pair for each property named in `advancedSearchPropertyValues` whose type is either `enumerated` or `RepositoryItem`. The key is the name of the property and the value is a Collection of the possible values. The `propertyValuesByType` property is useful for building forms that allow customers to select, for example, the size of an item, where size is an enumerated property with a set of predefined values like small, medium, and large. The following example illustrates this:

```
<!-- Create a select field to choose size and a default option -->
Size: <dsp:select bean="SearchHandler.propertyValues.size">

<!-- Create a default choice -->
<dsp:option value="" selected="true"/>Any size

<!-- Now create an option for each defined value for size -->
<dsp:droplet name="ForEach">
 <dsp:param value="SearchHandler.propertyValuesByType.size" name="array"/>
 <dsp:oparam name="output">
 <dsp:getvalueof id="option11" param="element" idtype="java.lang.String">

<dsp:option value="<%=option11><dsp:valueof param="element">Unknown
size</dsp:valueof>

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

</dsp:select>
```

Another approach to determining the possible values of properties whose type is `enumerated` or `RepositoryItem` is to use the servlet bean located at `/atg/commerce/catalog/RepositoryValues`. This component is an instance of the `atg.repository.servlet.PossibleValues` class. See `PossibleValues` in the *ATG Page Developer's Guide* for a detailed description of this servlet bean.

**Note:** When running in preview mode, the `/atg/commerce/catalog/RepositoryValues` component becomes an instance of `atg.commerce.catalog.FilteringCatalogPossibleValues`. For more information, see Searching Catalogs in Preview Mode.

The `RepositoryValues` servlet bean returns essentially the same information as the `propertyValuesByType` property of the `CatalogSeachFormHandler` class. However, there are some important differences:

- `RepositoryValues` determines possible values for only a single item type at a time, while the `propertyValuesByType` property works with multiple item types at the same time.
- `RepositoryValues` can look up values for any property of a repository item, while `propertyValuesByType` works with only the properties specified in the form handler's `advancedSearchPropertyNames` property.
- `RepositoryValues` works anywhere in a JSP, while the `propertyValuesByType` property is only available within search forms you construct using the `SearchFormHandler` class.

## Combining Catalog Search Types

The `CatalogSearchFormHandler` class allows you to specify multiple search types in a single request. For example, you can search on both keywords and text, or you can combine advanced searching with hierarchical searching to find only items in a particular category. In fact, you can use any combination of search types.

Search types are combined according to the following rules:

- Text and keyword searches are combined using the OR operator, so that a match on either set of criteria selects an item for inclusion in the search results.
- Hierarchical and advanced searches are combined using the AND operator, limiting the scope of the search to items that satisfy the hierarchical or advanced search requirements in addition to any specified text or keyword search criteria.

The query is of the form:

```
(KeywordConditions OR TextConditions) AND HierarchicalConditions AND
AdvancedSearchConditions
```

For example, suppose you have a catalog of movies, and you configure a search form handler to allow all four types of searches. The customer enters the following search criteria:

```
keywords=comedy
textSearchPropertyNames=description
searchInput=Steve Martin
hierarchicalCategoryId=BudgetMovies
propertyValues.format=DVD
```

The search will locate all comedies *plus* all movies whose description mentions Steve Martin, but will return only the subset of those movies that are found in the `BudgetMovies` category and are available on DVD.

## Processing Searches

To implement search in your pages, ensure that your JSP includes the search servlet bean. This example uses `CatalogSearch`, but you can use any of the other servlet beans for more specific searches.

```
<IMPORTBEAN BEAN="/atg/commerce/catalog/CatalogSearch">
```

`SearchFormHandler` executes its search query when the `handleSearch` method is called. Typically, you associate the form handler's `search` property with a submit button, as in this example:

```
<dsp:input bean="/myCatalog/SearchForm.search" value="Go" type="submit"/>
```

You can limit searches to a specific catalog or set of catalogs. The `CatalogSearchFormHandler` and `FilteringSearchFormHandler` both include `catalogs` and `queryByCatalog` properties. The `catalogs` property contains an array of catalog IDs. If this property is populated, then only items that have membership in at least one of the specified catalogs are returned. If the catalogs property is null, but the `queryByCatalog` property is set to `true`, the query is limited to the user's current catalog. If `catalogs` is null and `queryByCatalog` is `false`, the query searches all catalogs.

If you are using the multisite feature, you can also limit searches to a specific site or sites, using the following parameters:

- `siteIds`—Only items that belong to the specified sites are returned.
- `siteScope`—Specifies the site scope to use when searching for items. Possible values include:
  - `current`—Returns matching repository items from the current site only. This is the default.
  - `all`—Returns all matching repository items and does not filter based on site context.
  - any—Returns matching repository items that belong to any site.
  - none—Returns matching repository items that don't have any site affiliations.
  - *`shareable_type_ID`*—Returns matching repository items that belong to any sites that are in a sharing group with the current site, as defined by the `ShareableType` component ID. For example, you can return items that belong to sites that share a shopping cart with the current site.

The following example shows a product search form that uses the `siteIds` property to filter search results by site:

```
<dsp:importbean bean="/atg/dynamo/droplet/ForEach" />
<dsp:importbean bean="/atg/multisite/Site"/>
<dsp:importbean bean="/atg/commerce/catalog/ProductSearch"/>
<dsp:getvalueof id="contextroot" idtype="java.lang.String" bean="/Originating
Request.contextPath"/>

<dsp:form action="${contextroot}/search/searchResults.jsp" method="post" id=
"simpleSearch" formid="simplesearchform">

    <%-- Search input control --%>
    <dsp:input bean="ProductSearch.searchInput" type="text" value="" />

    <%-- Get the list of sites that share a shopping cart with the
    current site. --%>
    <dsp:droplet name="SharingSitesDroplet">
      <dsp:param name="shareableTypeId" value="atg.ShoppingCart"/>
      <dsp:param name="excludeInputSite" value="true"/>

      <%-- Loop through the sites that share a shopping cart and render labels
      and checkboxes for them. --%>
      <dsp:oparam name="output">
        <dsp:droplet name="ForEach">
          <dsp:param name="array" param="sites"/>
          <dsp:setvalue param="site" paramvalue="element"/>

            <%-- Display a checkbox and name for the current site first. --%>
            <dsp:oparam name="outputStart">
              <dsp:input bean="ProductSearch.siteIds" type="checkbox"
                         beanvalue="Site.id" checked="true" id="currentStore"/>
              <label for="currentStore">
                <dsp:valueof bean="Site.name"/>
              </label>
            </dsp:oparam>

<%-- Display the other sites that share shopping cart with the current
            site. --%>
            <dsp:oparam name="output">
                <dsp:input bean="ProductSearch.siteIds" type="checkbox"
                     paramvalue="site.id" id="otherStore" checked="false"/>
                <label for="otherStore">
                  <dsp:valueof param="site.name"/>
                </label>
            </dsp:oparam>
        </dsp:droplet>
      </dsp:oparam>

      <%-- If there are no shared sites, include the current site only and
      don't display checkboxes. --%>
      <dsp:oparam name="empty">
        <dsp:input bean="ProductSearch.siteIds" type="hidden"
                   beanvalue="Site.id"/>
      </dsp:oparam>
    </dsp:droplet>

    <%-- Display the search form's submit button. --%>
    <dsp:input bean="ProductSearch.search" type="submit"  value="Search"/>

</dsp:form>
```

## Displaying Search Results

After executing the query, `SearchFormHandler` makes the search results available in two different properties, which contain the same information but organize it differently:

- The `searchResults` property is a Collection of all catalog items that satisfied the search criteria. If you search for multiple item types (such as categories and products) all items returned by the search appear in the list regardless of their type.
- The `searchResultsByItemType` property is a HashMap containing one key/value pair for each item type you searched for. The key is the item type name (the value specified in the form handler's `itemTypes` property), and the value is a Collection of items of that type that satisfied the search criteria.

For example, if you search for categories and products in the catalog schema, the `searchResultsByItemType` property will have a key called `category` whose value is a Collection of matching categories, and another key called `product` whose value is a Collection of matching products. The `searchResults` property will have a Collection in which some of the items are categories and some of the items are products.

Within each Collection, the items are not sorted, but reflect the order they were retrieved from the database. You can use the sorting capabilities of a servlet bean (such as `ForEach`) to control the order in which the items are displayed.

The following example uses `ForEach` with `searchResultsByItemType` to display only the products returned by the search, sorted by display name:

```
Your search returned the following products:

<dsp:droplet name="ForEach">
 <dsp:param value="CatalogSearch.searchResultsByItemType.product" name="array"/>
 <dsp:param value="+displayName" name="sortProperties"/>

<dsp:oparam name="outputStart">
  <ul>
</ dsp:oparam>

 <dsp:oparam name="output">
   <li><dsp:valueof param="element.displayName">Unknown product</dsp:valueof></li>
 </dsp:oparam>

<dsp:oparam name="outputEnd">
  </ul>
</dsp:oparam>

 <dsp:oparam name="empty">
   <p>No matching products were found.
 </dsp:oparam>
</dsp:droplet>
```

The following example is longer and includes more of the available options:

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

<%--
---------------------------------------------------------------
This JSP bean displays the contents of a search
that potentially returns both category and product repository items.
The one paramater, ResultArray, accepts a HashMap that contains
elements with the keys "category" and "product".  The values of these
keys are collections of category or product repository items found in
the search.
--------------------------------------------------------------
--%>

<dsp:importbean bean="/atg/dynamo/droplet/Switch"/>
```

```
<dsp:importbean bean="/atg/dynamo/droplet/IsEmpty"/>
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:importbean bean="/atg/dynamo/droplet/RQLQueryForEach"/>

<dsp:droplet name="ForEach">
  <dsp:param param="ResultArray" name="array"/>

<%--Each item in this array is a Collection of Categories or
          Products...--%>
  <dsp:param value="ResultCollection" name="elementName"/>

  <dsp:oparam name="output">
    <dsp:droplet name="Switch">

<%--The key tells us if this is a Collection of Products
              or Categories:--%>
      <dsp:param param="key" name="value"/>

<%--For the list of CATEGORIES: --%>
      <dsp:oparam name="category">

        <blockquote>

        <dsp:droplet name="Switch">
          <dsp:param param="ResultCollection" name="value"/>
          <dsp:oparam name="default">
            <p>

<%--For each Category in the Collection: --%>
            <dsp:droplet name="ForEach">
              <dsp:param param="ResultCollection" name="array"/>
              <dsp:param value="+displayName" name="sortProperties"/>
              <dsp:param value="Category" name="elementName"/>
              <dsp:oparam name="outputStart">
                <b>We found these categories matching your search</b>
                <p>
              </dsp:oparam>
              <dsp:oparam name="output">


<%-- Display a link to the Category: --%>
                <dsp:getvalueof id="a78" param="Category.template.url"
                                        idtype="java.lang.String">
<dsp:a href="<%=a78%>">
                  <dsp:param param="Category.repositoryId" name="id"/>
                  <dsp:param value="jump" name="navAction"/>
                  <dsp:param param="Category" name="Item"/>
                  <dsp:valueof param="Category.displayName">No
                       name</dsp:valueof></dsp:a></dsp:getvalueof>
                <br>
              </dsp:oparam>
              <dsp:oparam name="empty">
                <b>There are no categories matching your search</b>
                <p>
              </dsp:oparam>
            </dsp:droplet>
          </dsp:oparam>

<%--If NO Categories returned by the search: --%>
          <dsp:oparam name="unset">
            No category items in the catalog could be found that match your query
          </dsp:oparam>
        </dsp:droplet>
<%--ForEach Category--%>

        </blockquote>
        <P>
      </dsp:oparam>

<%--For the list of PRODUCTS: --%>
      <dsp:oparam name="product">
        <blockquote><p>

        <dsp:droplet name="Switch">
          <dsp:param param="ResultCollection" name="value"/>

          <dsp:oparam name="default">


<%--For each Product in the Collection: --%>
            <dsp:droplet name="ForEach">
              <dsp:param param="ResultCollection" name="array"/>
              <dsp:param value="+displayName" name="sortProperties"/>
              <dsp:param value="Product" name="elementName"/>
              <dsp:oparam name="outputStart">
                <p>
                <b>We found these products matching your search</b>
                <p>
              </dsp:oparam>
              <dsp:oparam name="output">

<%-- Display a link to the Product: --%>
                <dsp:getvalueof id="a173" param="Product.template.url"
                            idtype="java.lang.String">
<dsp:a href="<%=a173%>">
                  <dsp:param param="Product.repositoryId" name="id"/>
                  <dsp:param value="jump" name="navAction"/>
                  <dsp:param param="Product" name="Item"/>
                  <dsp:valueof param="Product.displayName">No name</dsp:valueof>
                   &nbsp;-&nbsp;<dsp:valueof param="Product.description"/>
                </dsp:a></dsp:getvalueof>

                <br>
              </dsp:oparam>
              <dsp:oparam name="empty">
                <b>There are no products matching your search</b>
                <p>
              </dsp:oparam>

            </dsp:droplet>
<%--ForEach Product--%>

          </dsp:oparam>


<%--If NO Products returned by the search:--%>
          <dsp:oparam name="unset">
           No product items in the catalog could be found that match your query<p>
          </dsp:oparam>

        </dsp:droplet>
        </blockquote><P>
      </dsp:oparam>
    </dsp:droplet>

  </dsp:oparam>

</dsp:droplet>
<%--ForEach Item returned by Search --%>

</dsp:droplet>

</dsp:page>
```

## Searching Catalogs in Preview Mode

If you are using Content Administration to develop catalogs, be aware that searches against catalogs that have not been deployed to production (preview mode) are performed slightly differently than searches against deployed catalogs. The two main differences are that in preview mode the following hold true:

- The five preconfigured search components become instances of `atg.commerce.catalog.FilteringSearchFormHandler`
- The `/atg/commerce/catalog/RepositoryValues` component becomes an instance of `atg.commerce.catalog.FilteringCatalogPossibleValues`.

The following example demonstrates the different ways the two modes retrieve the same results.

- **In production mode:** By default, the `SearchFormHandler` and `CatalogPossibleValues` classes restrict the search results to the user's current catalog (see Processing Searches). If a user does a keyword search for "red" in production mode, the query includes a "where `keywordString` CONTAINS 'red'" clause and the `SearchFormHandler` adds another clause to make the query "where `keywordString` CONTAINS 'red' and `catalogs` CONTAINS (the user's current catalog)". Therefore, the repository would only return items that exist in the user's current catalog.

- **In preview mode:** The `catalogs` property of categories, products, and SKUs is derived and non-queryable. So the `SearchFormHandler` cannot add the extra clause to narrow the search to only the user's current catalog. Instead, the query is run without the extra clause and items not in the user's current catalog may be returned. Before returning the result set, `FilteringCatalogSearchFormHandler` then iterates through the results, and checks the `catalogs` property of each. Those items that do not contain the user's current catalog among their `catalogs` are then removed from the list, and the "filtered" result set containing only items in the user's current catalog is returned.

## Using Search Form Handlers with Internationalized Catalogs

Some businesses require internationalized catalogs, which can display product information in different languages, or display different sets of products to customers in different countries.

The `CatalogTools` component includes a property called `alternateRepositories` that lets you specify a mapping between symbolic names (called *repository keys*) and alternative repositories to use as the product catalog. You can then use customer locale as the repository key to determine which version of the catalog to display.

When the customer searches the catalog on an internationalized site, you want to make sure they search the catalog specific to their language or locale. You can do this by setting the search form handler's `repositoryKey` property to the name that identifies the repository you want to search. The search form handler uses `repositoryKey` to retrieve the appropriate catalog from the `CatalogTools` component. If you don't set a `repositoryKey`, the catalog repository is used.

The `repositoryKey` property is typically set through a hidden input field in the search form, as in this example:

```
<dsp:input value='<dsp:valueof bean="Profile.locale"/>' type="hidden"
bean="MySearchFormHandler.repositoryKey">
```

Using `repositoryKey` in conjunction with the `alternateRepositories` property of the `CatalogTools` component lets you ensure that customers see only the appropriate products when searching the catalog.

# 10 Implementing Product Comparison

Commerce sites often require the ability for a site user to compare items in the product catalog. A simple site may offer the user a single product comparison list and enable the user to add and remove products from the list, as well as compare the properties of the products in the list. A more complex site may offer the user multiple comparison lists to compare different types of items (for example, one list to compare cameras, a different list to compare televisions, and so on). ATG Commerce provides a product comparison system that enables you to meet both these simple and complex requirements.

The default implementation of the ATG Commerce 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. (Note that your application developers may have extended the system to include additional information.) Additionally, it enables the page developer to display product comparison information as a table, which the user can manipulate to change the sort criteria for the displayed information.

This chapter describes how to implement a product comparison system and includes the following sections:

**Understanding the ProductList Component**

**Querying the Product Comparison List**

**Managing Product Comparison Lists**

**Examples of Product Comparison Pages**

## Understanding the ProductList Component

By default, ATG Commerce includes a session-scoped instance of `ProductComparisonList`, located in Nucleus at `/atg/commerce/catalog/comparison/ProductList`. However, your application developers may have configured additional instances of `ProductComparisonList` to manage multiple comparison lists (for example, a list to compare cameras, a different list to compare televisions, and so on).

The `items` property of the `ProductList` component stores the list of `Entry` objects that represent each product in the product comparison list. Each `Entry` object combines category, product, SKU, and inventory information in a single object and, by default, exposes the properties described in the table below. (Note that your developers may have extended the product comparison system and added additional 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. |
| `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. |

| Property Name | Property Type | Description |
|---|---|---|
| `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>`. However, you can change the format by setting the `ProductComparisonList.productLinkFormat` property.<br><br>**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:<br><br>`columns=\`<br>`  Product Name=productLink,\`<br>`  Price=sku.listPrice,\`<br>`  …`<br><br>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:<br><br>`columns=\`<br>`    Product Name=productLink;`<br>`product.displayName,\`<br>`    Price=sku.listPrice,\`<br>`    …`<br><br>For more information on the `TableInfo` component, see the *Implementing Sortable Tables* chapter in the *ATG Page Developer's Guide*. |
| `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.<br><br>**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. |

| Property Name | Property Type | Description |
|---|---|---|
| `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 the Java code or by using `<dsp:valueof bean="ProductList.entries[id]"/>` in the JSP. You can also use this property to delete specific entries, for example, with a form handler. |

You can refer to the properties of entries in the product comparison list (that is, 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>
```

## Querying the Product Comparison List

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

By default, ATG Commerce includes a globally-scoped instance of `ProductListContains`, located in Nucleus at `/atg/commerce/catalog/comparison/ProductListContains`.

The `ProductListContains` servlet bean accepts the following input parameters:

- `productList` (Required)
  The `ProductComparisonList` object to examine.
- `productID` (Required)
  The repository ID of the product to look for in `productList`.
- `categoryID`
  The repository ID of the category to look for in `productList`.

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.

- `skuID`
  The repository ID of the SKU to look for in `productList`.

  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.

- `repositoryKey`
  The key to pass to `CatalogTools` to select a product catalog repository in which to look for the item. The key-to-catalog mapping is defined in `CatalogTools`. If this parameter is unset, the default product catalog repository is used.

  This optional parameter is useful for localization, which often requires the use of alternate product catalogs for different locales.

`ProductListContains` doesn't set any output parameters. However, it renders one of the following open parameters:

- `true`
  Rendered if the product comparison list contains the specific product, category, and SKU.

- `false`
  Rendered if the product comparison list doesn't contain the specified product, category, and SKU.

For JSP examples of the `ProductListContains` servlet bean, refer to Examples of Product Comparison Pages in this chapter.

## Managing Product Comparison Lists

The `ProductListHandler` form handler manages product comparison lists.

By default, ATG Commerce includes a request-scoped instance of `ProductListHandler`, which is located in Nucleus at `/atg/commerce/catalog/comparison/`. `ProductListHandler` is configured to operate on the product comparison list that is managed by the `ProductList` component located at `/atg/commerce/catalog/comparison/`. That is, the `ProductList` component (class `atg.commerce.catalog.comparison.ProductComparisonList`) is specified in `ProductListHandler.productList`.

If your application uses multiple instances of `ProductComparisonList` to manage multiple product comparison lists (for example, a list to compare cameras, a different list to compare televisions, and so on), 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 and SKU information if supplied in `categoryID` and `skuID`. |
| `handleAddProductAllSkus` | Adds all of the SKUs for the product specified by `productID` to the product comparison list, applying optional category information if supplied in `categoryID`. |
| `handleAddProductList` | Adds all of the products specified by `productIDList` to the product comparison list, applying optional category information if supplied in `categoryID` 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 information if supplied in `categoryID`. |
| `handleCancel` | Resets the form handler by setting `productID`, `categoryID`, and `skuID` to null. |
| `handleClearList` | Clears the product comparison list and redirects the user to the `clearListSuccessURL` on success. |
| `handleRefreshInventoryData` | Updates the inventory information in the product comparison list.<br><br>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` from the product comparison list. |
| `handleRemoveEntries` | Removes the list entries whose ids are specified in `entryIds` from the product comparison list. |

| Handle Method | Description |
|---|---|
| `handleRemoveProduct` | Removes the product specified by `productID` from the product comparison list, applying the optional category and SKU information if supplied in `categoryID` and `skuID`, respectively. |
| `handleRemoveProductAllSkus` | Removes all entries for the product specified by `productID` from the product comparison list. |
| `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`. |

Note that all of `ProductListHander`'s handle methods manage optional category and SKU information 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 API Reference*. For JSP examples of `ProductListHandler`, refer to the next section, Examples of Product Comparison Pages.

## Examples of Product Comparison Pages

This section provides JSP examples that illustrate the following:

- Displaying a Product Comparison Table
- Adding or Removing a Product from a Product Comparison List
- Adding Multiple Products to a Product Comparison List
- Removing Specific Entries from a Product Comparison List

### Displaying a Product Comparison Table

This JSP example shows how to display a simple product comparison table using the products in `ProductList.items` and the table information in `ProductList.tableInfo` (`ProductList`'s referenced `TableInfo` object).

Note that `ProductComparisonList` (of which the `ProductList` component is an instance) provides some convenience methods like `sortProperties` and `tableColumns` that call through to the referenced `TableInfo` object. Consequently, the expression `ProductList.tableColumns` is equivalent to the expression `ProductList.tableInfo.tableColumns`.

**Note:** The use and behavior of the `TableInfo` component is described in detail in the *Implementing Sortable Tables* chapter in the *ATG Page Developer's Guide*. Please refer to this manual for additional information on `TableInfo`.

```
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:importbean bean="/atg/dynamo/droplet/BeanProperty"/>
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductList"/>

<dsp:droplet name="ForEach">
  <dsp:param bean="ProductList.items" name="array"/>
  <dsp:param value="+product.displayName" name="sortProperties"/>

  <!-- Display table headings using TableInfo class -->
  <dsp:oparam name="outputStart">
    <table border="1" cellpadding="5" cellspacing="1">
    <dsp:droplet name="ForEach">
      <dsp:param bean="ProductList.tableColumns" name="array"/>
      <dsp:param value="" name="sortProperties"/>
      <dsp:oparam name="output">
        <dsp:valueof param="element.heading"/>
      </dsp:oparam>
    </dsp:droplet>
  </dsp:oparam>

  <!-- Display one table row for each item -->
  <dsp:oparam name="output">
    <dsp:setvalue paramvalue="element" param="currentProduct"/>
    <tr>
    <dsp:droplet name="ForEach">
      <dsp:param bean="ProductList.tableColumns" name="array"/>
      <dsp:param value="" name="sortProperties"/>
      <dsp:oparam name="output">
        <td>

        <dsp:droplet name="BeanProperty">
          <dsp:param param="currentProduct" name="bean"/>
          <dsp:param param="element.property" name="propertyName"/>
          <dsp:oparam name="output">
            <dsp:valueof valueishtml="<%=true%>" param="propertyValue"/>
          </dsp:oparam>
          </dsp:droplet>

          </td>
        </dsp:oparam>
     </dsp:droplet>
     </tr>
  </dsp:oparam>

  <!-- Close the table -->
  <dsp:oparam name="outputEnd">
    </table>
  </dsp:oparam>
</dsp:droplet>
```

## Adding or Removing a Product from a Product Comparison List

This JSP fragment shows how to add or remove a single product from a product comparison list. The example assumes that the `ProductListContains` servlet bean is embedded in a product display page using the following:

```
<dsp:include page="example.jsp"><dsp:param name="product"
     value="current product"/></dsp:include>
```

where *current product* is an expression that provides access to the product displayed on the page.

The given product is passed into the servlet bean in the `productId` input parameter. The `ProductListContains` servlet bean then checks whether it is stored in the product comparison list in `ProductList`. If the product is in the product comparison list, then the servlet bean renders the `true` open parameter on the product display page, and the user can click the "Remove from comparison list" submit button to remove the product from the list. If the product isn't in the product comparison list, then the servlet bean renders the `false` open parameter on the product display page, and the user can click the "Add to comparison list" submit button to add the product to the list.

```
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductList"/>
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductListContains"/>
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductListHandler"/>

<dsp:form action="product.jsp" method="POST">
<dsp:droplet name="ProductListContains">
  <dsp:param bean="ProductList" name="productList"/>
  <dsp:param param="product.repositoryId" name="productID"/>

  <dsp:oparam name="true">
    <dsp:input bean="ProductListHandler.productID" paramvalue="productID"
         type="hidden"/>
    <dsp:input bean="ProductListHandler.removeProduct" value="Remove from
         comparison list" type="submit"/>
  </dsp:oparam>

  <dsp:oparam name="false">
    <dsp:input bean="ProductListHandler.productID" paramvalue="productID"
         type="hidden"/>
    <dsp:input bean="ProductListHandler.addProduct" value="Add to
      comparison list" type="submit"/>
  </dsp:oparam>

</dsp:droplet>
</dsp:form>
```

## Adding Multiple Products to a Product Comparison List

This JSP example shows how to build a form from a search results list and let the user check off multiple products on the form and add them to a product comparison list.

Each product the user checks off on the form is added to `productIdList`, which stores the list of repository IDs for the products to add to the comparison list when calling either `ProductListHandler`'s `handleAddProductList` method or `handleAddProductListAllSkus` method. In this example, the `handleAddProductList` method is called. (For more information on `ProductListHandler`'s handle methods, refer to Managing Product Comparison Lists.)

```
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductListHandler"/>
<dsp:importbean bean="/atg/commerce/catalog/SearchFormHandler"/>

<dsp:form action="compare.jsp" method="POST">
<dsp:droplet name="ForEach">
  <dsp:param bean="SearchFormHandler.searchResults" name="array"/>
  <dsp:param value="+displayName" name="sortProperties"/>

  <dsp:oparam name="outputStart">
    <table border=0 cellpadding=0 cellspacing=0>
  </dsp:oparam>

  <dsp:oparam name="output">
    <tr>
    <td>
      <dsp:input bean="ProductListHandler.productIdList"
           paramvalue="element.repositoryId" type="checkbox"/>
      <dsp:valueof param="element.displayName"/> - <dsp:valueof
           param="element.description"/>
    </td>
    </tr>
  </dsp:oparam>

  <dsp:oparam name="outputEnd">
    </table></br>
    <dsp:input bean="ProductListHandler.addProductList" value="Add to list"
         type="submit"/>
  </dsp:oparam>
```

```
</dsp:droplet>
</dsp:form>
```

## Removing Specific Entries from a Product Comparison List

This JSP example shows how to build a form that lets the user either check off multiple entries to delete from a product comparison list or delete all of the entries from the list. The `ProductListHandler` identifies each entry by its unique ID, which is stored in the `id` property.

Each product the user checks off on the form is added to `entryIds`, which stores the list of entry IDs for the products to remove from the product comparison list when calling `ProductListHandler`'s `handleRemoveEntries` method.

To remove all of the entries from the product comparison list, the user can simply click the "Remove all" submit button, which calls `ProductListHandler`'s `handleClearAll` method. (For more information on `ProductListHandler`'s handle methods, refer to Managing Product Comparison Lists.)

```
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductList"/>
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductListHandler"/>

<dsp:form action="delete.jsp" method="POST">
<dsp:droplet name="ForEach">
  <dsp:param bean="ProductList.items" name="array"/>
  <dsp:param value="+product.displayName" name="sortProperties"/>

  <dsp:oparam name="empty">
    There are no items in your comparison list.
  </dsp:oparam>

  <dsp:oparam name="outputStart">
    <b>Remove items from comparison list</b>
    <blockquote>
  </dsp:oparam>

  <dsp:oparam name="output">
    <dsp:input bean="ProductListHandler.entryIds" paramvalue="element.id"
               type="checkbox"/>
    <dsp:valueof valueishtml="<%=true%>"
         param="element.product.displayName"/></br>
  </dsp:oparam>

  <dsp:oparam name="outputEnd">
    </blockquote>
    <br>
    <dsp:input bean="ProductListHandler.clearListSuccessURL" value="compare.jsp"
               type="hidden"/>
    <dsp:input bean="ProductListHandler.clearList" value="Remove
      all" type="submit"/>
    <dsp:input bean="ProductListHandler.removeProductSuccessURL"
              value="compare.jsp" type="hidden"/>
    <dsp:input bean="ProductListHandler.removeEntries" value="Remove selected
        items" type="submit"/>
  </dsp:oparam>

</dsp:droplet>
</dsp:form>
```

## Using Product Comparison Lists in a Multisite Environment

If you are using ATG'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.

**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.

# 11 Implementing Shopping Carts

In ATG Commerce, a shopping cart is an `Order` in an INCOMPLETE state. The shopping cart stores the information about the items a given customer wants to order and their associated quantities and prices. In addition, it stores the shipping and payment information for the order.

This chapter provides page developers with information on how to manage shopping carts on a commerce site. It includes the following sections:

**Understanding the ShoppingCart Component**
Provides information on the `ShoppingCart` component, which stores a customer's current and saved shopping carts.

**Managing Shopping Carts**
Provides information on retrieving, creating, modifying, and saving shopping carts.

Many of the code examples provided in this chapter are taken from the JSPs in the commerce sample catalog. For more information on the sample catalog and how to run it, see About the ATG Commerce Sample Catalog section.

For additional examples of how to manage shopping carts, you can refer to the Motorprise store reference application; see the *ATG Business Commerce Reference Application Guide*.

If you are using ATG's multisite feature, note that the shopping 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. Cart configuration for multisite is done through Site Administration; see the *ATG Multisite Administration Guide* for information.

**Note:** Because a shopping cart is, by definition, an `Order` in an INCOMPLETE state, the terms "shopping cart" and "order" are used interchangeably in this chapter.

## Understanding the ShoppingCart Component

The `ShoppingCart` component is responsible for storing and managing a customer's shopping carts. It maintains the customer's current shopping cart that is used during the purchase process, and it stores any other shopping carts that have been persisted by that customer. These shopping carts are represented as `atg.commerce.order.Order` objects in the ATG Commerce object model, and represented as order items in the Order Repository.

By default, the `/atg/commerce/ShoppingCart` component is a session-scoped instance of `atg.commerce.order.OrderHolder`. The following table describes its important properties:

| Property Name | Property Type | Description |
|---|---|---|
| `current` | `Order` | The current `Order` object. If null, then a new `Order` is automatically created. |
| `currentEmpty` | `boolean` | True indicates the current `Order` is null or includes no `CommerceItems`. |
| `currentExists` | `boolean` | True indicates the current `Order` exists. |
| `currentTransient` | `boolean` | True indicates the current `Order` is null or transient. |
| `empty` | `boolean` | True indicates both the current order and the collection of saved orders are empty. |
| `failoverRecoveryPricingOperation` | `String` | The operation to perform in case of failover. The default setting is ORDER_TOTAL. |
| `handlerOrderId` | `String` | Identifies the `Order`. |
| `last` | `Order` | The last completed `Order`. When an `Order` is submitted for checkout, the `Order` in `ShoppingCart.current` is moved to `ShoppingCart.last`, and the `ShoppingCart.current` property is reinitialized. |
| `orderType` | `String` | The type of order to create when constructing a new `Order`.<br><br>By default, this property is set to `/atg/commerce/order/OrderTools.defaultOrderType`. |
| `persistOrders` | `boolean` | True indicates the `Order` is persisted. |
| `repriceAfterFailoverRecovery` | `boolean` | True indicates that an `Order` will be repriced after failover recovery. |
| `restorableOrders` | `RestorableOrders` | The set of orders that can be restored through session backup. |

| | | |
|---|---|---|
| `saved` | `Collection` | A `Collection` of the user's saved shopping carts.<br><br>Applications should use the handle methods provided in `atg.commerce.order.OrderHolder` to move any cart from `ShoppingCart.saved` to `ShoppingCart.current`. See the *ATG API Reference* for more information. |
| `savedEmpty` | `boolean` | True indicates the `Collection` of saved shopping carts is null or empty. |

# Managing Shopping Carts

This section provides information on the following shopping cart tasks:

- Creating and Retrieving Shopping Carts
- Adding Items to Shopping Carts
- Adding Shipping Information to Shopping Carts
- Adding Payment Information to Shopping Carts
- Repricing Shopping Carts
- Saving Shopping Carts

## Creating and Retrieving Shopping Carts

As previously mentioned in this chapter, the `ShoppingCart` component stores a user's current and saved shopping carts. You can use the `ShoppingCart` component's properties and handle methods to create a new shopping cart or retrieve one of the user's saved shopping carts and make it the user's current shopping cart.

The following JSP example illustrates how to create and retrieve shopping carts. In the example, the `ShoppingCart.savedEmpty` property is checked to determine whether the current user has any saved shopping carts. If the user doesn't have any saved shopping carts, the user is given the option to create one. If the user has saved shopping carts, the user is given the option to select one of the saved shopping carts to either delete or make the current shopping cart, to delete all of the saved shopping carts, or to create a new shopping cart.

```
<dsp:importbean bean="/atg/commerce/ShoppingCart"/>
```

```
<dsp:form action="shoppingcart.jsp" method="post">
 <dsp:droplet name="/atg/dynamo/droplet/Switch">
   <dsp:param bean="ShoppingCart.savedEmpty" name="value"/>

   <dsp:oparam name="true">
     <!-- since there are no saved carts, we cannot switch to another so
          we only give them the option to create a new cart -->
     <dsp:input bean="ShoppingCart.create" value="Create" type="submit"/>
another shopping cart
   </dsp:oparam>

   <dsp:oparam name="false">
     <!-- We have other shopping carts, so let them do everything -->
     Shopping Cart <dsp:select bean="ShoppingCart.handlerOrderId">
     <dsp:droplet name="ForEach">
       <dsp:param bean="ShoppingCart.saved" name="array"/>
       <dsp:param value="savedcart" name="elementName"/>
       <dsp:oparam name="output">
         <dsp:getvalueof id="option26" param="savedcart.id"
idtype="java.lang.String">
<dsp:option value="<%=option26%>"/>
</dsp:getvalueof><dsp:valueof param="savedcart.id"/>
       </dsp:oparam>
     </dsp:droplet>
     </dsp:select>:
  <dsp:input bean="ShoppingCart.switch" value="Switch" type="submit"/> to,
  <dsp:input bean="ShoppingCart.delete" value="Delete" type="submit"/> or
  <dsp:input bean="ShoppingCart.create" value="Create" type="submit"/>
another shopping cart.<BR>
     <dsp:input bean="ShoppingCart.deleteAll"
value="Delete All Shopping Carts" type="submit"/>
   </dsp:oparam>

 </dsp:droplet>
</dsp:form>
```

### *Implementing Order Retrieval*

You can also use the `OrderLookup` servlet bean to retrieve a user's incomplete orders (that is, shopping carts). `OrderLookup` enables you to retrieve a single order, all orders assigned to a particular cost center (ATG Business Commerce only), all orders placed by a particular user, or all orders placed by a particular user that are in specific state, such as INCOMPLETE.

Once the desired shopping cart is moved to `ShoppingCart.current`, you can use a `ForEach` servlet bean to iterate over the commerce items in the cart and display them. The following JSP code example illustrates how to do this. In the example, a checkbox is rendered beside each item to allow the user to remove any item from the cart; likewise, a textbox is rendered beside each item to allow the user to modify the quantity of any item.

```
<droplet bean="ForEach">
  <param name="array" value="bean:ShoppingCart.current.commerceItems">
  <param name="elementName" value="item">
  <oparam name="output">
<tr valign=top>
<td>
  <input type="checkbox" unchecked
    bean="CartModifierFormHandler.removalCatalogRefIds"
    value="param:item.catalogRefId">
</td>
<td>
  <input size=4 name="param:item.catalogRefId"
value="param:item.quantity">
</td>
<td>
  <droplet src="product_fragment.jsp">
  <param name="childProduct" value="param:item.auxiliaryData.productRef">
  <%@ include file="product_fragment.jsp"%>
  </droplet>
  </oparam>
</droplet>
```

`cart.jsp` in the Motorprise reference application illustrates similar functionality. If you've installed ATG Business Commerce, you can access `cart.jsp` at `<ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\web-app\en\catalog\`. You can also open the page in the ACC's Document Editor via the Pages and Components>J2EE Pages task area.

## Adding Items to Shopping Carts

This section describes how to use the `CartModifierFormHandler` to add items to the current shopping cart. It includes the following sections:

- Adding One Item at a Time
- Adding Multiple Items at Once
- Overriding the Default Commerce Item Type
- Handling Custom Commerce Item Properties

### *Adding One Item at a Time*

The simplest way to add items to the current shopping cart is to add them one at a time. The following JSP code example serves as an illustration.

In the example, the user can select which SKU of the current product to add to the cart from a drop-down list. A `ForEach` servlet bean is used to iterate over the SKUs of the product and populate the drop-down list, which is associated with the `catalogRefIds` property of the `CartModifierFormHandler`. Additionally, the user can specify in a textbox a quantity of the selected SKU to add to the cart, which is associated with the `quantity` property of the `CartModifierFormHandler`.

When the user clicks the Add To Cart submit button, the form is processed, the properties of `CartModifierFormHandler` are set, and the `handleAddItemToOrder` method of `CartModifierFormHandler` is invoked. The `handleAddItemToOrder` method adds the quantity (specified in `CartModifierFormHandler.quantity`) of the selected SKU (specified in `CartModifierFormHandler.catalogRefIds` and identified by repository ID) to the current `Order` and then reprices the `Order`.

```
<dsp:importbean
bean="/atg/commerce/order/purchase/CartModifierFormHandler"/>
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>

<%--Create a form for user to select a SKU and add it to his/her cart:--%>
<dsp:getvalueof id="form10" bean="/OriginatingRequest.requestURI"
idtype="java.lang.String">
<dsp:form action="<%=form10%>" method="post">

<%--Store this product's ID in the Form Handler: --%>
<dsp:input bean="CartModifierFormHandler.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">

 <table cellpadding=0  cellspacing=0 border=0>
   <tr><td class=box-top-store>Add to Cart</td></tr>
   <tr><td class=box>

<%--Display any errors that have been generated during Cart
        operations:--%>
       <dsp:include
page="../../common/DisplayCartModifierFormHandlerErrors.jsp"></dsp:include>

       Add

<%--Textbox with QTY the user wants to order: --%>
       <dsp:input bean="CartModifierFormHandler.quantity" size="4"
value="1" type="text"/>

<%--Create a dropdown list with all SKUs in the Product.
        Store the selected SKU's id in the form handler: --%>
       <dsp:select bean="CartModifierFormHandler.catalogRefIds">

<%--For each of the SKUs in this Product, add the SKU to the
        dropdown list:--%>
         <dsp:droplet name="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="option73" param="Sku.repositoryID"
idtype="java.lang.String">
<dsp:option value="<%=option73%>"/>
</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>
       <br>


<%-- ADD TO CART BUTTON: Adds this SKU to the Order--%>
       <dsp:input bean="CartModifierFormHandler.addItemToOrder"
value="Add to Cart" type="submit"/>

<%-- Go to this URL if NO errors are found during the ADD TO
CART button processing:--%>
       <dsp:input bean="CartModifierFormHandler.addItemToOrderSuccessURL"
value="/checkout/cart.jsp" type="hidden"/>
     </td>
   </tr>
 </table>
</dsp:form></dsp:getvalueof>
```

For detailed information about `CartModifierFormHandler` and its handle methods, you can refer to the *Modifying Orders* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

Additionally, if you've installed ATG Business Commerce, you can also refer to `AddToCart.jsp` in the Motorprise reference application for a similar JSP code example. `AddToCart.jsp` is embedded into `product.jsp` to enable the user to add a quantity of the displayed SKU to the user's shopping cart. You can access both `AddToCart.jsp` and `product.jsp` at `<ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\web-app\en\catalog\`. You can also open these pages in the ACC's Document Editor via the Pages and Components>J2EE Pages task area.

### *Adding Multiple Items at Once*

You can create pages that allow users to add multiple items to the current shopping cart in a single form submission. The items can refer to different products, different SKUs, and have different quantities. The `CartModifierFormHandler` contains an `items` property that allows you to set per-item property values.

The following JSP code example illustrates adding multiple items for more than one SKU for a single product. In the example, the user can specify in a textbox a different quantity for each SKU to add to the cart. There are hidden input fields for the product ID and SKUs. Each product ID, SKU, and quantity textbox is associated with a subproperty of one element in the `CartModifierFormHandler.items` array.

When the user clicks the Add To Cart submit button, the form is processed, the properties of `CartModifierFormHandler` are set, and the `handleAddItemToOrder` method of `CartModifierFormHandler` is invoked. The `handleAddItemToOrder` method iterates through the `CartModifierFormHandler.items` elements and adds an item for each element with a non-zero `quantity`, using that element's `productId` and `catalogRefId` for the new item.

```
<dsp:importbean
bean="/atg/commerce/order/purchase/CartModifierFormHandler"/>
<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:form action="display_product.jsp" method="post">
<input name="id" type="hidden" value='<dsp:valueof
param="product.repositoryId"/>'>
<dsp:input bean="CartModifierFormHandler.addItemToOrderSuccessURL"
type="hidden" value="shoppingcart.jsp"/>
<table border=1>
<tr>
<td>SKU</td>
<td>Quantity</td>
</tr>
<dsp:droplet name="ForEach">
  <dsp:param name="array" param="product.childSKUs"/>
  <dsp:param name="elementName" value="sku"/>
  <dsp:param name="indexName" value="skuIndex"/>
  <dsp:oparam name="outputStart">
    <dsp:input bean="CartModifierFormHandler.addItemCount"
paramvalue="size" type="hidden"/>
  </dsp:oparam>
  <dsp:oparam name="output">
    <tr>
    <td><dsp:valueof param="sku.displayName"/></td>
    <td>
      <dsp:input
bean="CartModifierFormHandler.items[param:skuIndex].quantity" size="4"
type="text" value="0"/>
      <dsp:input
bean="CartModifierFormHandler.items[param:skuIndex].catalogRefId"
paramvalue="sku.repositoryId" type="hidden"/>
      <dsp:input
bean="CartModifierFormHandler.items[param:skuIndex].productId"
paramvalue="product.repositoryId" type="hidden"/>
    </td>
    </tr>
  </dsp:oparam>
```

```
</dsp:droplet>
</table>
<BR>
<dsp:input bean="CartModifierFormHandler.addItemToOrder" type="submit"
value="Add To Cart"/>
</dsp:form>
```

The `CartModifierFormHandler` must be told how many elements to allocate for the `items` array. This is done by setting the `CartModifierFormHandler.addItemCount` property. In the preceding example, `addItemCount` is set to the number of SKUs defined for the product in a hidden input field . This technique works in the example because all of the `CartModifierFormHandler.items` input fields have explicit `value` or `paramvalue` attributes.

The next code fragment illustrates a more complex technique for setting `CartModifierFormHandler.addItemCount`. This technique is appropriate if you want to preserve a user's input when a page is redisplayed because of a form submission error. The `dsp:setvalue` tag is not executed if the page is redisplayed.

```
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param name="value" bean="CartModifierFormHandler.addItemCount"/>
  <dsp:oparam name="0">
    <dsp:setvalue bean="CartModifierFormHandler.addItemCount" value="5"/>
  </dsp:oparam>
</dsp:droplet>
<dsp:input bean="CartModifierFormHandler.addItemCount" value="5"
type="hidden"/>
```

### Overriding the Default Commerce Item Type

By default, ATG Commerce supports three types of commerce items. One corresponds to regular SKUs. The other two correspond to configurable SKUs and their subproperties (see Creating Configurable SKUs in the *ATG Commerce Catalog Administration* chapter). Your sites may support additional custom commerce item types (see *Extending the Purchase Process* in the *Customizing the Purchase Process Externals* chapter in the *ATG Commerce Programming Guide*).

The `commerceItemType` property of `CartModifierFormHandler` determines what type of commerce item is created by `addItemToOrder`. Normally, `commerceItemType` is set to "default." You can specify a different commerce item type in the `CartModifierFormHandler` configuration file or in a form input field. If you add multiple items in a single form submission, you can override the `CartModifierFormHandler.commerceItemType` setting for an individual item by including a hidden input field to set `CartModifierFormHandler.items[n].commerceItemType`.

Values for `commerceItemType` must match keys in `/atg/commerce/order/OrderTools.commerceItemTypeClassMap`.

### *Handling Custom Commerce Item Properties*

The CartModifierFormHandler.addItemToOrder method has built-in support for some types of commerce item extensions. If your sites have extended commerce items with primitive data type properties, you can supply values for those properties by associating form fields with the CartModifierFormHandler.value Dictionary.

For example, suppose your sites have extended commerce items to support monograms, and the custom properties are named monogram and style. The following JSP code example illustrates how to handle user input for monograms without making any CartModifierFormHandler changes.

```
<b>Monogram Options</b><br>
Initials / Text: <dsp:input bean="CartModifierFormHandler.value.monogram"
size="20" type="text"/><br>
Style: <dsp:select bean="CartModifierFormHandler.value.style">
    <dsp:option value="Block"/>Block
    <dsp:option value="Diamond"/>Diamond
    <dsp:option value="Panel"/>Panel
    <dsp:option value="Stagger"/>Stagger
    <dsp:option value="Script"/>Script
 </dsp:select><br>
```

If you add multiple items in a single form submission, you can supply different custom property values for each item via the value Dictionary in each CartModifierFormHandler.items array element. Continuing with the preceding example, you could specify a single item's monogram text as:

```
Initials / Text: <dsp:input
bean="CartModifierFormHandler.items[0].value.monogram" size="20"
type="text"/><br>
```

If you add multiple items in a single form submission and you want to supply the same custom property values for all or most of the items, you can use CartModifierFormHandler.value for the common values and CartModifierFormHandler.items[n].value for special cases. If a property name appears in both the common Dictionary and an individual item's Dictionary, addItemToOrder uses the individual item's value.

Note that the value Dictionary cannot be used for standard commerce item properties, such as quantity and catalogRefId.

## Adding Shipping Information to Shopping Carts

Adding shipping information to shopping carts involves the following subprocesses:

- Creating a list of shipping groups for potential use in the current order. The user can select from among these shipping groups when checking out the order. See Creating Potential Shipping Groups.

- Specifying the shipping groups to use with the current order. See Adding Shipping Groups to an Order.
- Selecting the shipping methods, such as Ground or Next Day, for the order's shipping groups. See Selecting Shipping Methods.

**Note 1:** Be aware that the `ShippingGroupFormHandler` form handler and `AvailableShippingMethods` servlet bean, which are described in the subsections below, do **not** reprice a given `Order`. Consequently, if you enable customers to make order changes that affect order price through either mechanism, you should reprice the given `Order` using the `RepriceOrderDroplet` servlet bean before displaying its price to the customer. For more information on `RepriceOrderDroplet`, see Repricing Shopping Carts in this chapter.

**Note 2:** If an order contains any gift items, `ShippingGroupDroplet` and `ShippingGroupFormHandler` treat those items and their shipping information differently from other items. See the *ATG Commerce Programming Guide* for details.

### *Creating Potential Shipping Groups*

You can create a list of shipping groups for potential use in an `Order` in one of two ways:

- from information gathered from the user via forms
- from information stored in the user's profile

To create shipping groups from information obtained from the user via forms, use `CreateHardgoodShippingGroupFormHandler` and `CreateElectronicShippingGroupFormHandler`. These form handlers create, respectively, hard good and electronic shipping groups. Additionally, if the `addToContainer` property of the form handlers is set to true (which it is by default), they add the new shipping group to the `ShippingGroupMapContainer` and make it the default shipping group in the container. The `ShippingGroupMapContainer` stores the shipping groups available for use in the current order. Once the shipping group is added to the `ShippingGroupMapContainer`, the user can use it when checking out the current order. See Adding Shipping Groups to An Order.

The following JSP code example from `hardgood_sg.jsp` in the commerce sample catalog illustrates the use of `CreateHardgoodShippingGroupFormHandler`.

```
<dsp:importbean
bean="/atg/commerce/order/purchase/CreateHardgoodShippingGroupFormHandler"
/>
<dsp:importbean bean="/atg/userprofiling/Profile"/>

<hr>Enter new shipping address for HardgoodShippingGroup

<dsp:form action="hardgood_sg.jsp" method="post">

ShippingGroup NickName:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.hardgoodShippingGroupName"
size="30" type="text" value=""/>
```

```
<br>First:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.firstName" beanvalue="Profile.firstName" size="30" type="text"/>

Middle:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.middleName" beanvalue="Profile.middleName" size="30"
type="text"/>

Last:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.lastName" beanvalue="Profile.lastName" size="30" type="text"/>

<br>Address:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.address1" beanvalue="Profile.defaultShippingAddress.address1"
size="30" type="text"/>

Address (line 2):<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.address2" beanvalue="Profile.defaultShippingAddress.address2"
size="30" type="text"/>

<br>City:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.city" beanvalue="Profile.defaultShippingAddress.city" size="30"
type="text" required="<%=true%>"/>

State:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.state" maxsize="2"
beanvalue="Profile.defaultShippingAddress.state" size="2" type="text"
required="<%=true%>"/>

Postal Code:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.postalCode" beanvalue="Profile.defaultShippingAddress.postalCode"
size="10" type="text" required="<%=true%>"/>

Country:<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.HardgoodShippingGroup.Shippin
gAddress.country" beanvalue="Profile.defaultShippingAddress.country"
size="10" type="text"/>

<br>
<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.newHardgoodShippingGroupSucce
ssURL" type="hidden" value="shipping.jsp?init=false"/>

<dsp:input
```

```
bean="CreateHardgoodShippingGroupFormHandler.newHardgoodShippingGroupError
URL" type="hidden" value="shipping.jsp?init=false"/>

<dsp:input
bean="CreateHardgoodShippingGroupFormHandler.newHardgoodShippingGroup"
priority="<%=(int)-10%>" type="submit"
value="Create HardgoodShippingGroup"/>

</dsp:form>
```

The following JSP code example from `electronic_sg.jsp` in the commerce sample catalog illustrates the use of `CreateElectronicShippingGroupFormHandler`.

```
<dsp:importbean
bean="/atg/commerce/order/purchase/CreateElectronicShippingGroupFormHandle
r"/>
<dsp:importbean bean="/atg/userprofiling/Profile"/>

<hr>Enter new e-mail address for ElectronicShippingGroup

<dsp:form action="electronic_sg.jsp" method="post">

ShippingGroup NickName:<dsp:input
bean="CreateElectronicShippingGroupFormHandler.electronicShippingGroupNam
e" size="30" type="text"/>
<br>E-mail Address:<dsp:input
bean="CreateElectronicShippingGroupFormHandler.emailAddress"
beanvalue="Profile.email" size="30" type="text"/>

<br>
<dsp:input
bean="CreateElectronicShippingGroupFormHandler.newElectronicShippingGroupS
uccessURL" type="hidden" value="shipping.jsp?init=false"/>

<dsp:input
bean="CreateElectronicShippingGroupFormHandler.newElectronicShippingGroupE
rrorURL" type="hidden" value="shipping.jsp?init=false"/>

<dsp:input
bean="CreateElectronicShippingGroupFormHandler.newElectronicShippingGroup"
priority="<%=(int)-10%>" type="submit"
value="Create ElectronicShippingGroup"/>

</dsp:form>
```

In the commerce sample catalog, both `hardgood_sg.jsp` and `electronic_sg.jsp` are embedded into `shipping.jsp`, which itself manages the shipping information for the user's current order. (Note that `electronic_sg.jsp` is commented out.)

For more information on both `CreateHardgoodShippingGroupFormHandler` and `CreateElectronicShippingGroupFormHandler`, see the *Checking Out Orders* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

In contrast to creating shipping groups from information gathered from the user via forms, you can also create shipping groups from information that is stored in the user's profile using the `ShippingGroupDroplet` servlet bean. `ShippingGroupDroplet` uses the information obtained from the current user's profile to initialize `ShippingGroups` and add them to the `ShippingGroupMapContainer`. 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. If `HardgoodShippingGroups` are created, `ShippingGroupDroplet` also creates a default `HardgoodShippingGroup` for the container using the default shipping address in the user's profile. Once the shipping groups are added to the `ShippingGroupMapContainer`, the user can select from among them when checking out the current `Order`. See Adding Shipping Groups to An Order.

Additionally, `ShippingGroupDroplet` uses the information in the user's current `Order` to initialize `CommerceItemShippingInfo` objects for the `CommerceItem` objects in the `Order`. A `CommerceItemShippingInfo` object is a helper object that represents the relationship between a `CommerceItem` and a `ShippingGroup`; it contains properties that allow the total quantity in the `CommerceItem` to be spread across multiple shipping groups. The `CommerceItemShippingInfoContainer` stores the `CommerceItemShippingInfo` objects created for the current `Order`.

Once a set of `CommerceItemShippingInfo` objects is initialized for an order, you can present your customer with a form that allows the customer to:

- Specify a different `ShippingGroup` for each `CommerceItemShippingInfo` object.
- Update the `SplitQuantity` and `SplitShippingGroupName` property values of the `CommerceItemShippingInfo` objects and submit the changes by calling the `ShippingGroupFormHandler.splitShippingInfos` method. In this way, `CommerceItem` objects can be associated with additional `ShippingGroups` than those provided by the `ShippingGroupDroplet` initialization. The changes are stored in additional `CommerceItemShippingInfo` objects.

When the customer is satisfied with the `ShippingGroup` to `CommerceItem` associations, he or she clicks a button to proceed with the purchase process. Behind the scenes, this button invokes the `ShippingGroupFormHandler.applyShippingGroups` handler. The handler collects the information in the `CommerceItemShippingInfo` helper objects and adds corresponding `ShippingGroupCommerceItemRelationship` objects to the order. A `ShippingGroupCommerceItemRelationship` creates an association between a `CommerceItem` and a `ShippingGroup` and represents the quantity of the item in the `CommerceItem` that will be shipped using the information in the `ShippingGroup`.

While they are closely related, `CommerceItemShippingInfo` and `ShippingGroupCommerceItemRelationship` objects serve slightly different purposes. A `CommerceItemShippingInfo` object is external to the order and provides a means for defining commerce item-to-shipping group relationships. These changes do not affect the actual order until they are applied, allowing the order to remain in a stable state until the `ShippingGroupFormHandler.applyShippingGroups` handler is invoked. Once the information in the

CommerceItemShippingInfo objects is applied to the order, the relationships are stored in the order as ShippingGroupCommerceItemRelationship objects.

The input parameters passed into ShippingGroupDroplet determine how the droplet creates and initializes CommerceItemShippingInfo objects for each CommerceItem and whether the CommerceItemShippingInfoContainer is cleared before they are created. Three parameters control CommerceItemShippingInfo object creation: initShippingInfos, createOneInfoPerUnit, and initBasedOnOrder. These parameters, along with the other ShippingGroupDroplet input parameters, are described in the table below.

**Note:** For more information on CommerceItemShippingInfo objects, see *Shipping to Multiple Addresses* section in the *Processing Orders* chapter of the *ATG Business Commerce Reference Application Guide*. For more information on ShippingGroupCommerceItemRelationship objects, see the *Using Relationship Objects* section in the *Working with Purchase Process Objects* chapter of the *ATG Commerce Programming Guide*.

ShippingGroupDroplet takes the following input parameters:

| Parameter | Description |
|---|---|
| Clear | When set to True, ShippingGroupDroplet clears both the user's CommerceItemShippingInfoContainer and ShippingGroupMapContainer. |
| clearShippingGroups | When set to True, ShippingGroupDroplet clears the user's ShippingGroupMapContainer. |
| clearShippingInfos | When set to True, ShippingGroupDroplet clears the user's CommerceItemShippingInfoContainer.<br><br>This should be done at least once per Order to create fresh CommerceItemShippingInfo objects that refer to the unique CommerceItem objects in each Order. |
| createOneInfoPerUnit | When set to True, ShippingGroupDroplet creates a CommerceItemShippingInfo object for each individual unit contained in each CommerceItem. For example, a CommerceItem with a quantity of five will have five CommerceItemShippingInfo objects created for it. If a user has a default ShippingGroup in his or her profile, each CommerceItemShippingInfo object is initialized with that ShippingGroup. Set to False by default. |

| | |
|---|---|
| initBasedOnOrder | When set to True, ShippingGroupDroplet creates a CommerceItemShippingInfo object for each ShippingGroupCommerceItemRelationship object in the Order. The CommerceItemShippingInfo is initialized with the ShippingGroup that exists in the ShippingCommerceItemRelationship. This option is provided for the scenario where a customer has already gone part way through the checkout process and the order already contains some ShippingGroupCommerceItemRelationship objects. Set to False by default. |
| initShippingGroups | When set to True, the ShippingGroup types supplied in the shippingGroupTypes input parameter will be initialized. |
| initShippingInfos | When set to True, ShippingGroupDroplet creates a CommerceItemShippingInfo object for each CommerceItem in the Order. If a user has a default ShippingGroup in his or her profile, the CommerceItemShippingInfo object is initialized with that ShippingGroup. Set to True by default. |
| order | Used to override the component's default setting for the user's order. |
| shippingGroupTypes | A comma-separated list of ShippingGroup types, such as hardgoodShippingGroup or electronicShippingGroup, that is used to determine what types of ShippingGroups to initialize. |

ShippingGroupDroplet sets the following output parameters:

| Parameter | Description |
|---|---|
| shippingGroups | The Map referenced by the ShippingGroupMapContainer. |
| order | The Order object that represents the user's order. |

ShippingGroupDroplet renders one open parameter named output.

The following code example illustrates the use of ShippingGroupDroplet. The example creates HardgoodShippingGroup objects for the current user based on the availability of shipping address information in the user's profile. ShippingGroupDroplet also creates CommerceItemShippingInfo objects for the items in the user's current order.

```
<dsp:droplet name="ShippingGroupDroplet">
  <dsp:param value="true" name="clear"/>
  <dsp:param value="hardgoodShippingGroup" name="shippingGroupTypes"/>
  <dsp:param value="true" name="initShippingGroups"/>
  <dsp:param value="true" name="initShippingInfos"/>
  <dsp:oparam name="output"> Manipulation of objects here...
  </dsp:output>
</dsp:droplet>
```

You can refer to `shipping.jsp` in the commerce sample catalog for another JSP code example that illustrates the use of `ShippingGroupDroplet`.

### *Adding Shipping Groups to an Order*

Use `ShippingGroupFormHandler` to add shipping groups to an `Order` once the shipping information for the `Order` has been gathered from the following two processes (described in detail in the previous section):

- The shipping groups for potential use in the `Order` have been created via `CreateHardgoodShippingGroupFormHandler`, `CreateElectronicShippingGroupFormHandler`, and/or `ShippingGroupDroplet`. The shipping groups have been added to the `ShippingGroupMapContainer`.
- The `CommerceItemShippingInfo` objects for each `CommerceItem` in the `Order` have been created via `ShippingGroupDroplet`. The `CommerceItemShippingInfo` objects have been added to the `CommerceItemShippingInfoContainer`.

As an example, consider the following code segment from `complex_shipping.jsp` in the commerce sample catalog.

**Note:** In the code segment below, you can assume that each referenced component has been imported into the page via a `dsp:importbean` tag. See the actual JSP for these import statements.

```
<dsp:droplet name="ShippingGroupDroplet">
  <dsp:param name="clearShippingGroups" value="false"/>
  <dsp:param name="initShippingGroups" value="false"/>
  <dsp:param name="initShippingInfos" param="init"/>
  <dsp:oparam name="output">
  <!-- begin output -->

<table border=0 cellpadding=0 cellspacing=0 width=800>

  <tr>
    <td width=55></td>
    <td valign="top" width=745>
  <table border=0 cellpadding=4 width=80%>
    <tr><td></td></tr>
    <tr><td></td></tr>
    <tr valign=top>
      <td>
<%-- table with multiple rows with eleven cells  --%>
```

```
        <table border=0 cellpadding=4 cellspacing=1 width=100%>
          <tr>
            <td colspan=12><span class=help>To ship a line item to another
address, select the address and click the "Save" button. To ship only some
of the items to another address, change the quantity and select the
address. You must save changes individually before continuing.
            </span></td>
          </tr>
          <tr bgcolor="#666666" valign=bottom>
            <td colspan=2><span class=smallbw>Part #</span></td>
            <td colspan=2><span class=smallbw>Name</span></td>
            <td colspan=2 align=middle><span class=smallbw>Qty</span></td>
            <td colspan=2 align=middle><span class=smallbw>Qty to
move</span></td>
            <td colspan=2 align=middle><span class=smallbw>Shipping
address</span></td>
            <td colspan=2><span class=smallbw>Save changes</span></td>
          </tr>

<%-- get the real shopping cart items  --%>
          <dsp:droplet name="ForEach">
            <dsp:param name="array" param="order.commerceItems"/>
            <dsp:oparam name="output">
              <dsp:setvalue paramvalue="element" param="commerceItem"/>
              <dsp:setvalue bean="ShippingGroupFormHandler.listId"
paramvalue="commerceItem.id"/>
              <dsp:droplet name="ForEach">
                <dsp:param bean="ShippingGroupFormHandler.currentList"
name="array"/>
                <dsp:oparam name="output">
                  <!-- begin line item -->
                  <dsp:setvalue paramvalue="element" param="cisiItem"/>
                  <dsp:form action="complex_shipping.jsp" method="post">
                  <tr valign=top>
                   <td><nobr><dsp:valueof
param="commerceItem.auxiliaryData.catalogRef.manufacturer_part_number"/>
</nobr></td>
                   <td></td>
                   <td><dsp:valueof
param="commerceItem.auxiliaryData.catalogRef.displayName"/></td>
                   <td></td>
                   <td align=right>
<dsp:valueof param="element.quantity"/></td>
                   <td>&nbsp;</td>
                   <td>
                   <dsp:input
bean="ShippingGroupFormHandler.currentList[param:index].splitQuantity"
paramvalue="element.quantity" size="4" type="text"/></td>
                   <td>&nbsp;</td>
                   <td>
```

```
                     <dsp:select
bean="ShippingGroupFormHandler.currentList[param:index].splitShippingGroup
Name">
                     <dsp:droplet name="ForEach">
                       <dsp:param name="array" param="shippingGroups"/>
                       <dsp:oparam name="output">
                         <dsp:droplet name="Switch">
                           <dsp:param name="value" param="key"/>
                           <dsp:getvalueof id="nameval4"
param="cisiItem.shippingGroupName" idtype="java.lang.String">
<dsp:oparam name="<%=nameval4%>">
                             <dsp:getvalueof id="option305" param="key"
idtype="java.lang.String">
<dsp:option selected="<%=true%>" value="<%=option305%>"/>
</dsp:getvalueof><dsp:valueof param="key"/>
                           </dsp:oparam>
</dsp:getvalueof>
                           <dsp:oparam name="default">
                             <dsp:getvalueof id="option313" param="key"
idtype="java.lang.String">
<dsp:option selected="<%=false%>" value="<%=option313%>"/>
</dsp:getvalueof><dsp:valueof param="key"/>
                           </dsp:oparam>
                         </dsp:droplet>
                       </dsp:oparam>
                     </dsp:droplet>
                     </dsp:select>
                   </td>
                   <td></td>
                   <td>
                     <dsp:input
bean="ShippingGroupFormHandler.splitShippingInfosSuccessURL" type="hidden"
value="complex_shipping.jsp?init=false"/>
                     <dsp:input bean="ShippingGroupFormHandler.ListId"
paramvalue="commerceItem.id" priority="<%=(int)9%>" type="hidden"/>
                     <dsp:input
bean="ShippingGroupFormHandler.splitShippingInfos" type="submit"
value=" Save "/>
                   </td>
                  </tr>
                  </dsp:form>
                  <!-- end line item -->
                </dsp:oparam>
              </dsp:droplet><!-- end inner ForEach -->
            </dsp:oparam>
          </dsp:droplet><!-- end outer ForEach -->

        <tr>
          <td colspan=12>
<%-- table with one row with one cell  --%>
```

```
          <table border=0 cellpadding=0 cellspacing=0 width=100%>
            <tr bgcolor="#666666">
              <td></td>
            </tr>
          </table>
          </td>
        </tr>
      </table>
      </td>
    </tr>
    <tr>
      <td>
        <dsp:form action="complex_shipping.jsp" method="post">
        <dsp:input
bean="ShippingGroupFormHandler.applyShippingGroupsSuccessURL"
type="hidden" value="billing.jsp?init=true"/>
        <dsp:input bean="ShippingGroupFormHandler.applyShippingGroups"
type="submit" value="Continue"/>
        </dsp:form>
     </td>
   </tr>
 </table>
 </td>
</tr>
</table>

  <!-- end output -->
  </dsp:oparam>
</dsp:droplet><!-- end ShippingGroupDroplet -->
```

Note the following sections of `complex_shipping.jsp`:

1. When the page is rendered, `ShippingGroupDroplet` is used to initialize `CommerceItemShippingInfo` objects for the items in the user's current order and add them to the `CommerceItemShippingInfoContainer`. Recall from the previous section that, by default, `ShippingGroupDroplet` associates each `CommerceItemShippingInfo` object with the default shipping group in the `ShippingGroupMapContainer`.
2. The remainder of the code renders an interface that enables the user to assign quantities of the items in the order to different shipping groups. This is achieved through the use of nested `ForEach` servlet beans:
   - The outer `ForEach` servlet bean receives the array of items in the `Order` as an input parameter. It renders its `output` open parameter once for each `CommerceItem` in the `Order`. In the `oparam`, the `ShippingGroupFormHandler.listId` property is set to the ID of the current `CommerceItem`. The current item's ID is the key to its `List` of `CommerceItemShippingInfo` objects, which are now exposed via the `ShippingGroupFormHandler.currentList` property. (A hidden input tag farther down the page also sets this property on a subsequent request.)

- The inner `ForEach` servlet bean receives as an input parameter the array of `CommerceItemShippingInfo` objects for the current `CommerceItem` in the outer `ForEach` iteration. It renders its `output oparam` once for each `CommerceItemShippingInfo` object in the array. Essentially, the output rendered is a form that displays the following: the part # and name of the associated `CommerceItem`, the quantity in the current `CommerceItemShippingInfo` object, a drop-down list with which to change the shipping group for a specified quantity in the `CommerceItemShippingInfo` object, a textbox with which to specify the quantity in the `CommerceItemShippingInfo` to assign to the selected shipping group, and a Save submit button with which to make these new associations.

  The shipping group drop-down list is populated with the shipping groups in the `ShippingGroupMapContainer`. These are exposed by the `ShippingGroupDroplet` through a `shippingGroups` convenience parameter, and a third nested `ForEach` servlet bean is used to iterate over the shipping groups and populate the drop-down list.

  Note that the shipping group drop-down list is associated with the `splitShippingGroupName` property of the current `CommerceItemShippingInfo` object. Similarly, the quantity textbox is associated with the `splitQuantity` property of the current `CommerceItemShippingInfo` object.

  Finally, note that the Save submit button invokes the `handleSplitShippingInfos` method of `ShippingGroupFormHandler`. The `handleSplitShippingInfos` method uses the values in the `quantity` and `splitQuantity` properties of the `CommerceItemShippingInfo` object to determine if the object should be split into two objects. If that is necessary, it then uses these properties and the shipping group specified in the `splitShippingGroupName` property to construct a second `CommerceItemShippingInfo` object. The method then sets the properties of both the old and new objects accordingly and adds the new object to the `CommerceItemShippingInfoContainer`. Once the form is processed, the page is rendered again and reflects the changes the user has made.

**3.** The Continue submit button at the bottom of the page enables the user to apply the current shipping associations to the order and proceed to specifying billing information. The submit button invokes the `handleApplyShippingGroups` method of `ShippingGroupFormHandler`, which adds the shipping groups that the user has selected to the `Order`. It does this by iterating over the `CommerceItemShippingInfo` **objects** in the `CommerceItemShippingInfoContainer`. For each `CommerceItemShippingInfo` object in the container, the associated shipping group is retrieved from the `ShippingGroupMapContainer` and added to the `Order`, and the appropriate quantity of the associated `CommerceItem` is added to that `ShippingGroup`.

**Note:** For detailed information on all the handle methods of `ShippingGroupFormHandler`, see the *Preparing a Complex Order for Checkout* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*. For more information on adding items to shipping groups, see the

*Assigning Items to Shipping Groups* section of the *Working With Purchase Process Objects* chapter in the *ATG Commerce Programming Guide*.

If you've installed ATG Business Commerce, you can also refer to `checkout/shipping.jsp` and `checkout/ship_to_multiple.jsp` in the Motorprise reference application for JSP code examples that illustrate the use of `ShippingGroupFormHandler`. You can access these pages at `<ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\web-app\en\checkout\`. You can also open these pages in the ACC's Document Editor via the Pages and Components>J2EE Pages task area. For more details, see the *Shipping Information* section of the *Processing Orders* chapter in the *ATG Business Commerce Reference Application Guide*.

### *Selecting Shipping Methods*

You can use the `atg/commerce/pricing/AvailableShippingMethods` servlet bean to provide the user with a list of available shipping methods (such as Ground or Next Day) for a particular shipping group. Given a shipping group, `AvailableShippingMethods` queries the `ShippingPricingEngine` and returns a list of the shipping methods available for the type of the given shipping group.

`AvailableShippingMethods` requires only one input parameter, named `shippingGroup`, which is the specific shipping group to be shipped. (For a complete list of the input parameters of `AvailableShippingMethods`, see *The Pricing Servlet Beans* section of the *Using and Extending Pricing Services* chapter of the *ATG Commerce Programming Guide*.) It sets one output parameter named `availableShippingMethods`, which is a list of Strings representing the shipping methods that can be used to set a `shippingMethod` value in a `HardgoodShippingGroup`. Additionally, it renders one open parameter named `output`.

The following JSP code segment from `/checkout/shipping_method.jsp` in the Motorprise reference application illustrates the use of `AvailableShippingMethods`. In the example, the `AvailableShippingMethods` servlet bean is used to populate a select list from which to choose a shipping method for the current shipping group.

```
<tr valign=top>
 <td align=right width=25%><span class=smallb>Shipping method</span></td>
 <td align=left>
  <%-- The AvailableShippingMethods servlet bean permits the user to
     select a shipping method that is applied to the current
     ShippingGroup. --%>
  <dsp:droplet name="AvailableShippingMethods">
   <dsp:param name="shippingGroup" param="sGroup">
   <dsp:param bean="UserPricingModels.shippingPricingModels"
    name="pricingModels">
   <dsp:param bean="Profile" name="profile">

   <dsp:oparam name="output">
     <dsp:select
bean="ShoppingCart.current.ShippingGroups[param:index].shippingMethod">
      <dsp:droplet name="ForEach">
       <dsp:param name="array" param="availableShippingMethods">
       <dsp:param name="elementName" value="method">
       <dsp:oparam name="output">
        <dsp:getvalueof id="methodname" idtype="String" param="method">
        <dsp:option value="<%=methodname%>"><dsp:valueof
param="method"></dsp:getvalueof>
       </dsp:oparam>
      </dsp:droplet>
     </dsp:select>
    </dsp:oparam>

  </dsp:droplet>
 </td>
</tr>
```

If you've installed ATG Business Commerce, you can access `shipping_method.jsp` at `<ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\web-app\en\checkout\`. You can also open the page in the ACC's Document Editor via the Pages and Components>J2EE Pages task area. For more information on `shipping_method.jsp`, see the *Shipping Information* section of the *Processing Orders* chapter in the *ATG Business Commerce Reference Application Guide*.

## Adding Payment Information to Shopping Carts

Adding payment information to shopping carts involves the following subprocesses:

- Creating a list of payment groups for potential use in the current order. The user can select from among these payment groups when checking out the order. See Creating Potential Payment Groups.
- Specifying the payment groups to use with current order. See Adding Payment Groups to an Order.

### *Creating Potential Payment Groups*

You can create a list of payment groups for potential use in an `Order` in one of two ways:

- from information gathered from the user via forms
- from information stored in the user's profile

To create payment groups from information obtained from the user via forms, use `CreateCreditCardFormHandler` and `CreateInvoiceRequestFormHandler`. (**Note:** The `CreateInvoiceRequestFormHandler` is provided with ATG Business Commerce only.) These form handlers create, respectively, `CreditCard` and `InvoiceRequest` payment groups. Additionally, if the `addToContainer` property of the form handlers is true (which it is by default), they add the new payment group to the `PaymentGroupMapContainer` and make it the default payment group in the container. The `PaymentGroupMapContainer` stores the payment groups available for use in the current order. Once the payment group is added to the `PaymentGroupMapContainer`, the user can use it when checking out the current order. See Adding Payment Groups to an Order.

The following JSP code example from `credit_card.jsp` in the commerce sample catalog illustrates the use of `CreateCreditCardFormHandler`.

```
<dsp:importbean
bean="/atg/commerce/order/purchase/CreateCreditCardFormHandler"/>
<dsp:importbean bean="/atg/userprofiling/Profile"/>

<hr><p>Enter new CreditCard information

<dsp:form action="credit_card.jsp" method="post">

<br>CreditCard NickName:<dsp:input
bean="CreateCreditCardFormHandler.creditCardName" size="30" type="text"
value=""/>

<br>CreditCardNumber:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.CreditCardNumber"
maxsize="20" size="20" type="text" value="4111111111111111"/>

<br>CreditCardType:
<dsp:select bean="CreateCreditCardFormHandler.creditCard.creditCardType"
required="<%=true%>">
<dsp:option value="Visa"/>Visa
<dsp:option value="MasterCard"/>Master Card
<dsp:option value="American Express"/>American Express
</dsp:select>

<br>ExpirationMonth: <dsp:select
bean="CreateCreditCardFormHandler.creditCard.ExpirationMonth">
<dsp:option value="1"/>January
<dsp:option value="2"/>February
<dsp:option value="3"/>March
<dsp:option value="4"/>April
<dsp:option value="5"/>May
<dsp:option value="6"/>June
<dsp:option value="7"/>July
<dsp:option value="8"/>August
<dsp:option value="9"/>September
<dsp:option value="10"/>October
<dsp:option value="11"/>November
<dsp:option value="12"/>December
</dsp:select>

<br>expirationYear:Year: <dsp:select
bean="CreateCreditCardFormHandler.creditCard.expirationYear">
<dsp:option value="2002"/>2002
<dsp:option value="2003"/>2003
<dsp:option value="2004"/>2004
<dsp:option value="2005"/>2005
<dsp:option value="2006"/>2006
<dsp:option value="2007"/>2007
<dsp:option value="2008"/>2008
```

```
<dsp:option value="2009"/>2009
<dsp:option value="2010"/>2010
</dsp:select>

<br>FirstName:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.firstName"
beanvalue="Profile.firstName" size="30" type="text"/>
<br>MiddleName:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.middleName"
beanvalue="Profile.middleName" size="30" type="text"/>
<br>LastName:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.lastName"
beanvalue="Profile.lastName" size="30" type="text"/>
<br>EmailAddress:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.email"
beanvalue="Profile.email" size="30" type="text"/>
<br>PhoneNumber:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.phoneNumber"
beanvalue="Profile.daytimeTelephoneNumber" size="30" type="text"/>
<br>Address:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.address1"
beanvalue="Profile.defaultBillingAddress.address1" size="30" type="text"/>
<br>Address (line 2):<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.address2"
beanvalue="Profile.defaultBillingAddress.address2" size="30" type="text"/>
<br>City:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.city"
beanvalue="Profile.defaultBillingAddress.city" size="30" type="text"/>
<br>State:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.state"
beanvalue="Profile.defaultBillingAddress.state" size="30" type="text"/>
<br>PostalCode:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.postalCode"
beanvalue="Profile.defaultBillingAddress.postalCode" size="30"
type="text"/>
<br>Country:<dsp:input
bean="CreateCreditCardFormHandler.creditCard.billingAddress.country"
beanvalue="Profile.defaultBillingAddress.country" size="30" type="text"/>
<dsp:input bean="CreateCreditCardFormHandler.copyToProfile" type="hidden"
value="false"/>

<dsp:input bean="CreateCreditCardFormHandler.newCreditCardSuccessURL"
type="hidden" value="billing.jsp?init=false"/>
<dsp:input bean="CreateCreditCardFormHandler.newCreditCardErrorURL"
type="hidden" value="credit_card.jsp"/>
<dsp:input bean="CreateCreditCardFormHandler.newCreditCard" type="submit"
value="Enter Credit Card"/>

</dsp:form>
```

In the commerce sample catalog, `credit_card.jsp` is embedded into `billing.jsp`, which itself manages the payment information for the user's current order.

You use `CreateInvoiceRequestFormHandler` in the same way that you use `CreateCreditCardFormHandler`, although note that by default the `CreateInvoiceRequestFormHandler` requires that a `poNumber` is specified for an invoice payment group. For more information on both `CreateCreditCardFormHandler` and `CreateInvoiceRequestFormHandler`, see the *Checking Out Orders* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

In contrast to creating payment groups from information gathered from the user via forms, you can also create payment groups from information that is stored in the user's profile using the `PaymentGroupDroplet` servlet bean. `PaymentGroupDroplet` uses the information obtained from the current user's profile to initialize payment groups and add them to the `PaymentGroupMapContainer`. The input parameters passed into `PaymentGroupDroplet` determine what types of `PaymentGroups` are created (such as, credit card, store credit, and/or gift certificate), and whether the `PaymentGroupMapContainer` is cleared before they are created. Once the payment groups are added to the `PaymentGroupMapContainer`, the user can select from among them when checking out the current `Order`. See Adding Payment Groups to an Order.

Additionally, `PaymentGroupDroplet` uses the information in the user's current `Order` to initialize `CommerceIdentifierPaymentInfo` objects for the `Order`. A `CommerceIdentifierPaymentInfo` object is a helper object that represents the relationship between an `Order` and its components (commerce items, shipping, and taxes) and payment information; it contains properties that allow the cost of a given item, as well as the order's shipping costs and tax, to be spread across multiple payment groups. The `CommerceIdentifierPaymentInfoContainer` stores the `CommerceIdentifierPaymentInfo` objects created for use in the current `Order`. `CommerceIdentifierPaymentInfo` objects are further subclassed into the following types:

- `OrderPaymentInfo` objects store payment information for the entire order, including all commerce items, shipping, and taxes.
- `ItemPaymentInfo` objects store payment information for individual items.
- `ShippingPaymentInfo` objects store payment information for shipping costs.
- `TaxPaymentInfo` objects store payment information for taxes.

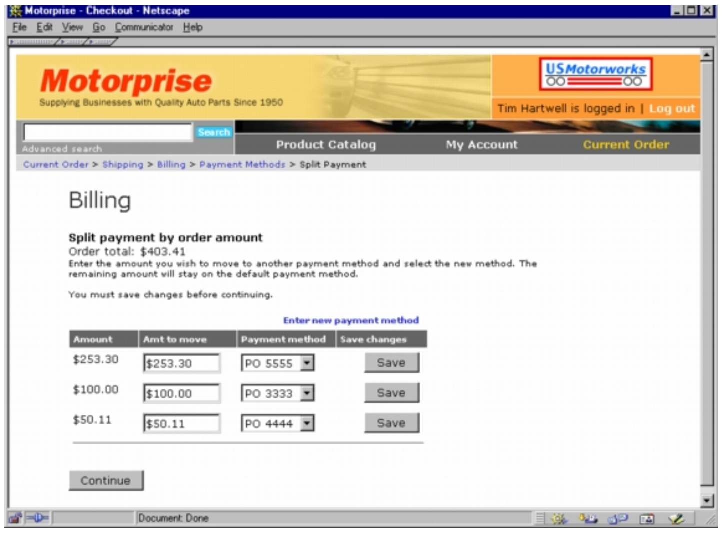
The input parameters you provide to `PaymentGroupDroplet` determine which of the four `CommerceIdentifierPaymentInfo` object types are initialized for the `Order`. In general, you will opt to either initialize an `OrderPaymentInfo` object for the entire order, or a combination of `ItemPaymentInfo`, `ShippingPaymentInfo`, and `TaxPaymentInfo` objects that account for all of the order's components.

Once a set of `CommerceIdentifierPaymentInfo` objects is initialized for an order, you can present your customer with a form that allows the customer to:

- Specify a different `PaymentGroup` for each `CommerceIdentifierPaymentInfo` object.
- Update the `SplitPaymentMethod`, `SplitAmount`, and `SplitQuantity` property values of the `CommerceIdentifierPaymentInfo` objects and submit the changes by calling the `PaymentGroupFormHandler.handleSplitPaymentInfos` method. In

this way, CommerceIdentifier objects can be associated with additional PaymentGroups other than those provided by the PaymentGroupDroplet initialization. The changes are stored in additional CommerceIdentifierPaymentInfo objects.

The illustration below shows a sample user interface that allows the user to split payment amounts among PaymentGroups.


Motorprise - Checkout - Netscape
File Edit View Go Communicator Help
Motorprise
Supplying Businesses with Quality Auto Parts Since 1950
USMotorworks
Tim Hartwell is logged in | Log out
Search
Advanced search
Product Catalog
My Account
Current Order
Current Order > Shipping > Billing > Payment Methods > Split Payment
Document: Done

Billing

**Split payment by order amount**
Order total: $403.41
Enter the amount you wish to move to another payment method and select the new method. The remaining amount will stay on the default payment method.

You must save changes before continuing.

Enter new payment method

| Amount | Amt to move | Payment method | Save changes |
|---|---|---|---|
| $253.30 | $253.30 | PO 5555 | Save |
| $100.00 | $100.00 | PO 3333 | Save |
| $50.11 | $50.11 | PO 4444 | Save |

Continue

When the customer is satisfied with the PaymentGroup to CommerceIdentifier associations, he or she clicks a button to proceed with the purchase process. Behind the scenes, this button invokes the PaymentGroupFormHandler.handleApplyPaymentGroups method. This method collects the information in the CommerceIdentifierPaymentInfo helper objects and adds corresponding PaymentGroup relationship objects to the order. A PaymentGroup relationship object creates an association between a CommerceIdentifier and a PaymentGroup and represents the amount of the cost in the CommerceIdentifier that will be paid for using the information in the PaymentGroup. PaymentGroup relationship objects can be of several types:

- A PaymentGroupOrderRelationship represents a relationship between an Order and a PaymentGroup. This relationship object also stores tax payment information.
- A PaymentGroupCommerceItemRelationship represents a relationship between a CommerceItem and a PaymentGroup.
- A PaymentGroupShippingGroupRelationship represents a relationship between a ShippingGroup and a PaymentGroup.

While they are closely related, CommerceIdentifierPaymentInfo and PaymentGroup relationship objects serve slightly different purposes. A CommerceIdentifierPaymentInfo object is external to the order and provides a means for defining commerce identifier-to-payment group relationships. These changes do not affect the actual order until they are applied, allowing the order to remain in a stable state until the PaymentGroupFormHandler.handleApplyPaymentGroups method is invoked. Once the information in the CommerceIdentifierPaymentInfo objects is applied to the order, the relationships are stored in the order as PaymentGroup relationship objects.

**Note:** For more detailed information on PaymentGroup relationship objects, see the *Using Relationship Objects* section in the *Working with Purchase Process Objects* chapter of the *ATG Commerce Programming Guide*.

The input parameters passed into PaymentGroupDroplet determine how the droplet creates and initializes CommerceIdentifierPaymentInfo objects for the order and whether the CommerceIdentifierPaymentInfoContainer is cleared before they are created. Parameters control CommerceIdentifierPaymentInfo object creation: initOrderPayment, initItemPayment, initShippingPayment, initTaxPayment, and initBasedOnOrder. These parameters, along with the other PaymentGroupDroplet input parameters, are described in the table below.

When you use PaymentGroupDroplet, you should consider the following:

- Whether the user is paying for the total cost of the Order with one or more payment groups **or** the component costs of the Order (commerce item costs, shipping costs, and tax) with one or more payment groups.
- The types of payment groups the user can use to pay for the Order. By default, ATG Commerce supports the following types of PaymentGroups: giftCertificate, storeCredit, and creditCard. ATG Business Commerce also supports invoiceRequest PaymentGroups.

These factors determine what kind of PaymentGroup objects you use PaymentGroupDroplet to initialize for potential use in the order.

PaymentGroupDroplet takes the following input parameters:

| Parameter | Description |
|---|---|
| clear | Boolean. When set to True, PaymentGroupDroplet clears both the user's CommerceIdentifierPaymentInfoContainer and PaymentGroupMapContainer. |
| cleaPaymentGroups | Boolean. When set to True, PaymentGroupDroplet clears the user's PaymentGroupMapContainer. |
| clearPaymentInfos | Boolean. When set to True, PaymentGroupDroplet clears the user's CommerceIdentifierPaymentInfoContainer. This should be done at least once per new order; the default value is false. |

| | |
|---|---|
| `createAllPaymentInfos` | Boolean. When set to `True`, the `PaymentGroupDroplet` creates an `OrderPaymentInfo` for all possible payment types stored in the user's profile (credit cards, store credit, gift certificate, or invoice request). Initially, the entire order's cost is assigned to the `OrderPaymentInfo` that is associated with the default `PaymentGroup` and all other `OrderPaymentInfo` objects are set to 0. This option supports a different type of user interface than that which is described in the sections above. Use `createAllPaymentInfos` to initialize the objects needed to support a user interface where the user manually specifies the amount of the order to be paid by each payment type in her profile. See Using createAllPaymentInfos for more information.<br><br>This option is `False` by default. |
| `initBasedOnOrder` | Boolean. When set to `True`, `PaymentGroupDroplet` creates a `CommerceIdentifierPaymentInfo` object for each `PaymentGroup` relationship object in the `Order`. This option is provided for the scenario where a customer has already gone part way through the checkout process and the order already contains some `PaymentGroup` relationship objects.<br><br>The types of `CommerceIdentifierPaymentInfo` objects that are created correspond to the `PaymentGroup` relationship types. For example, if a `PaymentGroupCommerceItemRelationship` exists in the `Order`, `PaymentGroupDroplet` creates a corresponding `CommerceItemPaymentInfo` object and adds it to the `CommerceIdentifierPaymentInfoContainer`. Each `CommerceIdentifierPaymentInfo` object is initialized with the `PaymentGroup` that exists in its corresponding `PaymentGroup` relationship object.<br><br>Set to `False` by default. |
| `initItemPayment` | Boolean. When set to `True`, `PaymentGroupDroplet` creates a `CommerceItemPaymentInfo` object for each `CommerceItem` in the order and adds them to the `CommerceIdentifierPaymentInfoContainer`. If a user has a default `PaymentGroup` in his or her profile, the `CommerceItemPaymentInfo` object is initialized with that `PaymentGroup`. Set to `False` by default.<br><br>**Note:** A `CommerceItemPaymentInfo` object is a `CommerceIdentifierPaymentInfo` object whose `CommerceIdentifier` is a `CommerceItem`; it is used for `CommerceItem` payment information. |

| | |
|---|---|
| `initOrderPayment` | Boolean. When set to `True`, `PaymentGroupDroplet` creates an `OrderPaymentInfo` object and adds it to the `CommerceIdentifierPaymentInfoContainer`. If a user has a default `PaymentGroup` in his or her profile, the `OrderPaymentInfo` object is initialized with that `PaymentGroup`. Set to `True` by default.<br><br>**Note:** An `OrderPaymentInfo` object is a `CommerceIdentifierPaymentInfo` object whose `CommerceIdentifier` is an `Order`; it is used for `Order` payment information. |
| `initPaymentGroups` | Boolean. When set to `True`, the `PaymentGroup` types supplied in the `paymentGroupTypes` input parameter are initialized. |
| `initShippingPayment` | Boolean. When set to `True`, `PaymentGroupDroplet` creates a `ShippingGroupPaymentInfo` object for each `ShippingGroup` in the order and adds them to the `CommerceIdentifierPaymentInfoContainer`. If a user has a default `PaymentGroup` in his or her profile, the `ShippingGroupPaymentInfo` object is initialized with that `PaymentGroup`. Set to `False` by default.<br><br>**Note:** A `ShippingGroupPaymentInfo` object is a `CommerceIdentifierPaymentInfo` object whose `CommerceIdentifier` is a `ShippingGroup`; it is used for `ShippingGroup` payment information. |
| `initTaxPayment` | Boolean. When set to `True`, `PaymentGroupDroplet` creates a `TaxPaymentInfo` object and adds it to the `CommerceIdentifierPaymentInfoContainer`.<br><br>A `TaxPaymentInfo` object is a `CommerceIdentifierPaymentInfo` object whose `CommerceIdentifier` is an `Order`; it is used for tax payment information. |
| `order` | The user's `Order`. You can use this parameter to override the default setting for `PaymentGroupDroplet.order`. |
| `paymentGroupTypes` | A comma-separated list of `PaymentGroup` types (such as "creditCard," "storeCredit," "giftCertificate") that is used to determine which `PaymentGroupInitializer` components are executed. |

`PaymentGroupDroplet` sets the following output parameters:

| Parameter | Description |
|---|---|
| `cipiMap` | The `Map` referenced by the `CommerceIdentifierPaymentInfoContainer`. |
| `order` | The `Order` object that represents the user's order. |
| `paymentGroups` | The `Map` referenced by the `PaymentGroupMapContainer`. |

`PaymentGroupDroplet` renders one open parameter named `output`.

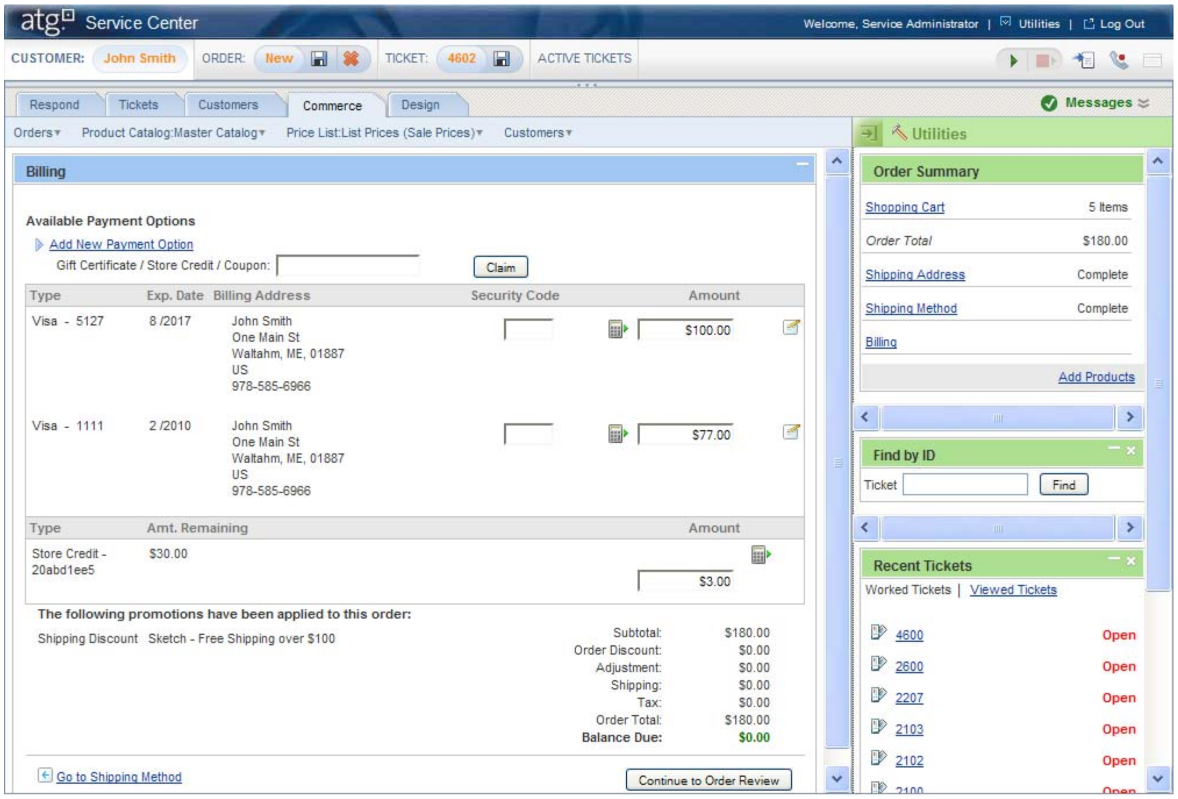
The following code example illustrates the use of `PaymentGroupDroplet`. The example creates `CreditCard`, `StoreCredit`, and `GiftCertificate` `PaymentGroup` objects based on their availability for the current user. Additionally, it creates `CommerceItemPaymentInfo` objects, `ShippingGroupPaymentInfo` objects, and a `TaxPaymentInfo` object. The example enables the user to pay for the `CommerceIdentifiers` in the `Order` at the line item level with any of the available `PaymentGroup` objects.

```
<dsp:droplet name="PaymentGroupDroplet">
  <dsp:param value="true" name="clear"/>
  <dsp:param value="giftCertificates, storeCredit, creditCard"
       name="paymentGroupTypes"/>
  <dsp:param value="true" name="initPaymentGroups"/>
  <dsp:param value="true" name="initItemPayment"/>
  <dsp:param value="true" name="initTaxPayment"/>
  <dsp:param value="true" name="initShippingPayment"/>
  <dsp:oparam name="output">Manipulation of objects here...
  </output>
</dsp:droplet>
```

You can refer to `billing.jsp` and `invoice_request.jsp` in the commerce sample catalog for additional JSP code examples that illustrate the use of `PaymentGroupDroplet`.

### *Using createAllPaymentInfos*

You can use the `createAllPaymentInfos` input parameter to initialize the objects needed to support a user interface where the user manually specifies the amount of the order to be paid by each payment type in her profile, for example:



**Billing**

Available Payment Options

Add New Payment Option

Gift Certificate / Store Credit / Coupon: Claim

| Type | Exp. Date | Billing Address | Security Code | Amount |
|---|---|---|---|---|
| Visa - 5127 | 8 /2017 | John Smith<br>One Main St<br>Waltahm, ME, 01887<br>US<br>978-585-6966 | | $100.00 |
| Visa - 1111 | 2 /2010 | John Smith<br>One Main St<br>Waltahm, ME, 01887<br>US<br>978-585-6966 | | $77.00 |

| Type | Amt. Remaining | Amount |
|---|---|---|
| Store Credit - 20abd1ee5 | $30.00 | $3.00 |

The following promotions have been applied to this order:

Shipping Discount Sketch - Free Shipping over $100

| | |
|---|---|
| Subtotal: | $180.00 |
| Order Discount: | $0.00 |
| Adjustment: | $0.00 |
| Shipping: | $0.00 |
| Tax: | $0.00 |
| Order Total: | $180.00 |
| **Balance Due:** | **$0.00** |

Go to Shipping Method

Continue to Order Review

**Order Summary**

| | |
|---|---|
| Shopping Cart | 5 Items |
| Order Total | $180.00 |
| Shipping Address | Complete |
| Shipping Method | Complete |
| Billing | |

Add Products

**Find by ID**

Ticket Find

**Recent Tickets**

Worked Tickets | Viewed Tickets

| | |
|---|---|
| 4600 | Open |
| 2600 | Open |
| 2207 | Open |
| 2103 | Open |
| 2102 | Open |
| 2100 | Open |

When the `createAllPaymentInfos` parameter is set to `True`, the `PaymentGroupDroplet` creates an `OrderPaymentInfo` for all possible payment types stored in the user's profile (credit cards, store credit, gift certificate, or invoice request). Initially, the entire order's cost is assigned to the `OrderPaymentInfo` that is associated with the default `PaymentGroup` and all other `OrderPaymentInfo` objects are set to 0. For example, consider the following scenario:

- The total cost of the order is $100.
- The user's profile has two credit cards stored in it, Credit Card A and Credit Card B as well as a store credit.
- Credit Card A is the default payment method for the user.

In this scenario, `PaymentGroupDroplet` creates three `OrderPaymentInfo` objects, one for each credit card and another for the store credit. The `amount` property for the `OrderPaymentInfo` associated with Credit Card A is set to 100. The amount properties for the `OrderPaymentInfo` objects associated with Credit Card B and the store credit are set to 0.

In the application, the user is presented with a list of payment options generated by iterating over the list of `OrderPaymentInfo` objects. The user provides the amount to be paid by each payment option directly in the form, effectively setting the `amount` property for the `OrderPaymentInfo` object associated with each `PaymentGroup`. If the user adds additional `PaymentGroups` during the checkout process, you should call the `PaymentGroupDroplet` again to create `OrderPaymentInfo` objects for the newly added `PaymentGroups`.

### *Adding Payment Groups to an Order*

Use `PaymentGroupFormHandler` to add payment groups to an `Order` once the payment information for the `Order` has been gathered from the following two processes (described in detail in the previous section):

- The payment groups for potential use in the `Order` have been created via `CreateCreditCardFormHandler`, `CreateInvoiceRequestFormHandler`, and/or `PaymentGroupDroplet`. The payment groups have been added to the `PaymentGroupMapContainer`.
- The `CommerceIdentifierPaymentInfo` objects for the `CommerceIdentifiers` in the `Order` have been created via `PaymentGroupDroplet`. The `CommerceIdentifierPaymentInfo` objects have been added to the `CommerceIdentifierPaymentInfoContainer`.

As an example, consider the following code segment from `complex_billing.jsp` in the commerce sample catalog. This page permits the user to divide the total cost of the `Order` across multiple `CreditCard` payment groups.

**Note:** In the code segment below, you can assume that each referenced component has been imported into the page via a `dsp:importbean` tag. See the actual JSP for these import statements.

```
<dsp:droplet name="PaymentGroupDroplet">
  <dsp:param name="initOrderPayment" param="init"/>
  <dsp:param name="clearPaymentInfos" param="init"/>
  <dsp:oparam name="output">
  <dsp:setvalue bean="PaymentGroupFormHandler.listId"
paramvalue="order.id"/>
  <!-- begin output -->
     <table border=0 cellpadding=0 cellspacing=0 width=800>
      <tr>
      </tr>
      <tr>
        <td width=55></td>
        <td valign="top" width=745>
        <table border=0 cellpadding=4 width=80%>
          <tr><td></td></tr>
          <tr>
            <td colspan=2><span class="big">Billing</span></td>
          </tr>
          <tr><td></td></tr>
          <tr>
            <td colspan=2><b>Split payment by order amount</b><br>
            Order total: <dsp:valueof converter="currency"
param="order.priceInfo.total"/><br>
            <span class=help>Enter the amount you wish to move to another
payment method and select the new method. The remaining amount will stay
on the default payment method. <P>You must save changes before
continuing.</span></td>
```

```
          </tr>
          <tr valign=top>
            <td>
            <table border=0 cellpadding=4 cellspacing=1>
              <tr valign=top>
                <td colspan=9 align=right>
                </td>
              </tr>
              <tr valign=bottom bgcolor="#666666">
                <td colspan=2><span class=smallbw>Amount</span></td>
                <td colspan=2><span class=smallbw>Amt to move
&nbsp;</span></td>
                <td colspan=2><span class=smallbw>Payment
method</span></td>
                <td colspan=3><span class=smallbw>Save changes</span></td>
              </tr>
                  <dsp:droplet name="ForEach">
                    <dsp:param bean="PaymentGroupFormHandler.currentList"
name="array"/>
                    <dsp:oparam name="output">
                      <!-- begin order line item -->
                      <dsp:form action="complex_billing.jsp"
method="post">
                      <tr valign=top>
                        <td><dsp:valueof converter="currency"
param="element.amount"/></td>
                        <td>&nbsp;</td>
                        <td>
                        $<dsp:input
bean="PaymentGroupFormHandler.currentList[param:index].splitAmount"
size="6" value="0.00" type="text"/></td>
                        <td>&nbsp;</td>
                        <td>
                          <dsp:select
bean="PaymentGroupFormHandler.currentList[param:index].splitPaymentMe
thod">
                          <dsp:droplet name="ForEach">
                            <dsp:param name="array"
param="paymentGroups"/>
                            <dsp:oparam name="output">
                              <dsp:droplet name="Switch">
                                <dsp:param name="value" param="key"/>
                                <dsp:getvalueof id="nameval3"
param="...element.paymentMethod" idtype="java.lang.String">
<dsp:oparam name="<%=nameval3%>">
                                  <dsp:getvalueof id="option264"
param="key" idtype="java.lang.String">
<dsp:option selected="<%=true%>" value="<%=option264%>"/>
</dsp:getvalueof><dsp:valueof param="key"/>
                                </dsp:oparam>
```

```
</dsp:getvalueof>
                                <dsp:oparam name="default">
                                  <dsp:getvalueof id="option272"
param="key" idtype="java.lang.String">
<dsp:option selected="<%=false%>" value="<%=option272%>"/>
</dsp:getvalueof><dsp:valueof param="key"/>
                                </dsp:oparam>
                              </dsp:droplet>
                            </dsp:oparam>
                          </dsp:droplet>
                          </dsp:select>
                        </td>
                        <td>&nbsp;</td>
                        <td>&nbsp;</td>
                        <td>&nbsp;</td>
                        <td>
                         <dsp:input bean="PaymentGroupFormHandler.ListId"
paramvalue="order.id" priority="<%=(int)9%>" type="hidden"/>
                         <dsp:input
bean="PaymentGroupFormHandler.splitPaymentInfosSuccessURL" type="hidden"
value="complex_billing.jsp?init=false"/>
                         <dsp:input
bean="PaymentGroupFormHandler.splitPaymentInfos" type="submit"
value=" Save "/>
                        </td>
                      </tr>
                      </dsp:form>
                      <!-- end order line item -->
                    </dsp:oparam>
                  </dsp:droplet>

      <td colspan=9>
<%-- table with one row with one cell  --%>
      <table border=0 cellpadding=0 cellspacing=0 width=100%>
        <tr bgcolor="#666666">
          <td></td>
        </tr>
      </table>
      </td>
    </tr>
           </table>
          </td>
        </tr>
        <tr>
          <td><br>
            <dsp:form action="complex_billing.jsp" method="post">
            <dsp:input
bean="PaymentGroupFormHandler.applyPaymentGroupsSuccessURL" type="hidden"
value="order_confirmation.jsp"/>
            <dsp:input bean="PaymentGroupFormHandler.applyPaymentGroups"
```

```
type="submit" value="Continue"/>
            </dsp:form>
          </td>
       </tr>
     </table>
     </td>
    </tr>
    </table>
    </div>

  <!-- end output -->
  </dsp:oparam>
</dsp:droplet> <!-- end PaymentGroupDroplet -->
```

Note the following sections of `complex_billing.jsp`:

1. When the page is rendered, `PaymentGroupDroplet` is used to initialize an `OrderPaymentInfo` object for the user's current order and add it to the `CommerceIdentifierPaymentInfoContainer`. Recall from the previous section that by default `PaymentGroupDroplet` associates each `CommerceIdentifierPaymentInfo` object with the default payment group in the `PaymentGroupMapContainer`. (Note that the available `CreditCard` payment groups have been initialized and added to the `PaymentGroupMapContainer` on a previous page, `billing.jsp`.)

2. The `PaymentGroupFormHandler.listId` property is set to the ID of the `Order` object set in the `order` output parameter of `PaymentGroupDroplet`. The order's ID is the key to its `List` of `CommerceIdentifierPaymentInfo` objects, which are now exposed via the `PaymentGroupFormHandler.currentList` property. (A hidden input tag farther down the page also sets this property on a subsequent request.)

3. The remainder of the code renders an interface that enables the user to assign specific amounts of the order's total cost to different payment groups. This is achieved through the use of nested `ForEach` servlet beans:

   - The outer `ForEach` servlet bean receives as an input parameter the array of `CommerceIdentifierPaymentInfo` objects for the current `Order`. It renders its `output oparam` once for each `CommerceIdentifierPaymentInfo` object in the array. Essentially, the output rendered is a form that displays the following: the amount associated with the current `CommerceIdentifierPaymentInfo` object, a drop-down list with which to change the payment group for a specified amount in the `CommerceIdentifierPaymentInfo` object, a textbox with which to specify the amount in the `CommerceIdentifierPaymentInfo` to associate with the selected payment group, and a Save submit button with which to make these new associations.

     The payment group drop-down list is populated with the payment groups in the `PaymentGroupMapContainer`. These are exposed by the `PaymentGroupDroplet` through a `paymentGroups` convenience parameter, and a second nested `ForEach` servlet bean is used to iterate over the payment groups and populate the drop-down list.

Note that the payment group drop-down list is associated with the `splitPaymentMethod` property of the current `CommerceIdentifierPaymentInfo` object. Similarly, the amount textbox is associated with the `splitAmount` property of the current `CommerceIdentifierPaymentInfo` object.

Finally, note that the Save submit button invokes the `handleSplitPaymentInfos` method of `PaymentGroupFormHandler`. The `handleSplitPaymentInfos` method uses the values in the `amount` and `splitAmount` properties of the `CommercIdentifierPaymentInfo` object to determine if the object should be split into two objects. If that is necessary, it then uses these properties and the payment group specified in the `splitPaymentMethod` property to construct a second `CommerceIdentifierPaymentInfo` object. The method then sets the properties of both the old and new objects accordingly and adds the new object to the `CommerceIdentifierPaymentInfoContainer`. Once the form is processed, the page is rendered again and reflects the changes the user has made.

4. The Continue submit button at the bottom of the page enables the user to apply the current payment associations to the order and proceed to order confirmation. The submit button invokes the `handleApplyPaymentGroups` method of `PaymentGroupFormHandler`, which adds the payment groups that the user has selected to the `Order`. It does this by iterating over the `CommerceIdentifierPaymentInfo` objects in the `CommerceIdentifierPaymentInfoContainer`. For each `CommerceIdentifierPaymentInfo` object in the container, the associated payment group is retrieved from the `PaymentGroupMapContainer` and added to the `Order`, and the appropriate amount of the associated `CommerceIdentifier` is added to that `PaymentGroup`.

**Note:** For detailed information on all the handle methods of `PaymentGroupFormHandler`, see the *Preparing a Complex Order for Checkout* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*. For more information on adding costs to payment groups, see the *Assigning Costs to Payment Groups* section of the *Working With Purchase Process Objects* chapter in the *ATG Commerce Programming Guide*.

If you've installed ATG Business Commerce, you can refer to `checkout/payment_methods.jsp`, `checkout/SplitPaymentOrderDetails.jsp`, and `checkout/SplitPaymentDetails.jsp` in the Motorprise reference application for additional JSP code examples that illustrate the use of `PaymentGroupFormHandler`. In particular, note that `/checkout/SplitPaymentDetails.jsp` illustrates how to enable the user to split the component costs or "line items" in the order (that is, the item costs, shipping costs, and tax) across multiple payment groups. You can access these Motorprise pages at `<ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\web-app\en\checkout\`. You can also open these pages in the ACC's Document Editor via the Pages and Components>J2EE Pages task area. For more details, see the *Payment Information* section of the *Processing Orders* chapter in the *ATG Business Commerce Reference Application Guide*.

## Repricing Shopping Carts

As described earlier in this chapter in Adding Items to Shopping Carts, `CartModifierFormHandler` automatically reprices a shopping cart when it is used to add items to the cart. (Note that it also reprices the cart when it is used to *remove* items from the cart.) However, you'll need to reprice shopping carts via some other mechanism if customers can make changes that affect order price through other form handlers that do not reprice shopping carts (for example, by making shipping changes via the form handlers that create and manage shipping groups), or if the shopping carts 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 a shopping cart, but you cannot do so through a form action and corresponding handle method, use the `RepriceOrderDroplet` servlet bean. In fact, you can use the `RepriceOrderDroplet` servlet bean to reprice a customer's shopping cart every time the customer accesses a shopping cart page. This ensures that the customer always views accurate pricing information as he or she makes changes to cart.

The `RepriceOrderDroplet` servlet bean takes one required input parameter, `pricingOp`, that should be set to the pricing operation to execute. The possible pricing operations are defined in `atg.commerce.pricing.PricingConstants`, and they include the following:

ORDER_TOTAL
ORDER_SUBTOTAL
ORDER_SUBTOTAL_SHIPPING
ORDER_SUBTOTAL_TAX
ITEMS
SHIPPING
ORDER
TAX
NO_REPRICE

Typically, the `pricingOp` input parameter is the only parameter you need to specify when using `RepriceOrderDroplet`. For a detailed list of all the parameters of `RepriceOrderDroplet`, see RepriceOrder in *Appendix: ATG Commerce Servlet Beans*.

To use `RepriceOrderDroplet` in a page on your sites, simply import the servlet bean into the page using the following import statement:

```
<dsp:importbean
bean="/atg/commerce/order/purchase/RepriceOrderDroplet">
```

Then add JSP code similar to the following before displaying any pricing information for the current shopping cart.

```
<dsp:droplet name="RepriceOrderDroplet">
  <dsp:param value="ORDER_SUBTOTAL" name="pricingOp"/>
</dsp:droplet>-->
```

For additional information on the `RepriceOrderDroplet` servlet bean, see the *Repricing Orders* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

## Saving Shopping Carts

You use the `SaveOrderFormHandler` to save the user's current shopping cart, add the shopping cart to the user's list of saved carts in `ShoppingCart.saved`, and construct a new, empty cart as the user's current shopping cart in `ShoppingCart.current`.

Additionally, you can use the `description` property of `SaveOrderFormHandler` to set the `description` property of the `Order`; this enables users to name their shopping carts with meaningful names. If the user doesn't provide a descriptive name, then the `SaveOrderFormHandler` automatically creates one using the user's locale and the current date and time.

The following JSP code illustrates the use of `SaveOrderFormHandler`.

```
<dsp:importbean bean="/atg/commerce/order/purchase/SaveOrderFormHandler"/>

Order # <dsp:valueof bean="ShoppingCart.current.id"/>
  <dsp:form action="saved_orders.jsp">
    Enter a name to identify this order:<p>
    <dsp:input bean="SaveOrderFormHandler.description" type="text"/>
    <dsp:input bean="SaveOrderFormHandler.saveOrder" value="Save order"
         type="submit"/>
    <dsp:input bean="SaveOrderFormHandler.saveOrderSuccessURL"
         value="../user/saved_orders.jsp" type="hidden"/>
    <dsp:input bean="SaveOrderFormHandler.saveOrderErrorURL"
         value="../user/save_order.jsp" type="hidden"/>
  </dsp:form>
```

For descriptions of the handle methods of `SaveOrderFormHandler` and detailed information on how an `Order` is saved to the Order Repository, see the *Saving Orders* section of the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*. For a JSP code example illustrating the use of `SaveOrderFormHandler`, see the *Saving Orders* section of the *My Account* chapter in the *ATG Business Commerce Reference Application Guide*.

The `OrderLookup` servlet bean retrieves one or more `Order` objects, depending on the supplied input parameters. You can use this servlet bean to retrieve:

- A single order
- All orders placed by a particular user
- All orders placed by a particular user that are in a specific state
- All orders assigned to a particular cost center (ATG Business Commerce only)

The `OrderLookup` servlet bean that is provided with ATG Commerce is an instance of class `atg.commerce.order.OrderLookup` if you are using ATG Consumer Commerce, or `atg.b2bcommerce.order.B2BOrderLookup` if you are using ATG Business Commerce.

The behavior of the servlet bean is governed by several properties that are set in the `OrderLookup` component. You can use the Pages and Components > Components by Path area of the ATG Control

Center to view and modify the `OrderLookup` component, which is located at `/atg/commerce/order/`. The following properties are important when configuring the `OrderLookup` servlet bean.

| Property | Description |
|---|---|
| `enableSecurity` | If this property is set to false, any user can view any order.<br><br>If this property is set to true, users are restricted to viewing only their own orders. A user is considered to own an order if the profile ID associated with the order matches the profile ID of that user.<br><br>By default, this property is set to true. |
| `orderManager` | The path of the global `OrderManager` component that is responsible for all direct interactions with `Order` objects. For most stores, this is set to `/atg/commerce/order/OrderManager`. |
| `profilePath` | The path to the `Profile` object of the user that is currently logged in. For most stores, this is set to `/atg/userprofiling/Profile`. |
| `openStates` | This property contains a list of order states that indicate that an order is open. Because several order states can indicate that an order is open, you can use this property to specify those states. Consequently, when you query for all "open" orders, you retrieve all orders with states specified in this property. By default, this property is set to the following:<br><br>-- `submitted`<br>-- `processing`<br>-- `pending_merchant_action`<br>-- `pending_customer_action`<br><br>You can override this list of "open" states by using the optional `openStates` input parameter, which is described later in this section. |
| `closedStates` | This property contains a list of order states that indicate that an order is closed. Because several order states can indicate that an order is closed, you can use this property to specify those states. Consequently, when you query for all "closed" orders, you retrieve all orders with states specified in this property. By default, this property is set to the following:<br><br>-- `no_pending_action`<br><br>You can override this list of "closed" states by using the optional `closedStates` input parameter, which is described later in this section. |
| `useRequestLocale` | Error messages generated by this servlet bean can be localized. If this boolean property is set to `true`, the messages are localized to the locale requested by the user. |

| | |
|---|---|
| `defaultLocale` | This property specifies the default locale to use to localize error messages. The `defaultLocale` is used to localize the messages if `useRequestLocale` is set to false or if the user does not request a locale. |
| `queryTotal` | If this property is set to false, the servlet bean does not set the `totalCount` and `total_count` output parameters to the total number of orders. (The `totalCount` and `total_count` output parameters are described later in this section.) |

The `OrderLookup` servlet bean takes the following input parameters:

| Parameter | Description |
|---|---|
| `orderId` | The ID of the order to retrieve.<br><br>**Note:** If you are using ATG Consumer Commerce, either this or `userId` is required. If you are using ATG Business Commerce, either this, `userId`, or `costCenterId` is required. |
| `userId` | The ID of the user profile whose orders will be retrieved.<br><br>**Note:** If you are using ATG Consumer Commerce, either this or `orderId` is required. If you are using ATG Business Commerce, either this, `orderId`, or `costCenterId` is required. |
| `costCenterId`<br><br>(ATG Business Commerce only) | The ID of the cost center whose orders will be retrieved.<br><br>**Note:** This parameter applies to ATG Business Commerce only. Either this, `orderId`, or `userId` is required. |
| `state` | The desired state of the orders to retrieve.<br><br>This parameter can be used in conjunction with `userId`. You can specify one of the following:<br><br>-- any one of the states defined in `atg.commerce.states.OrderStates` (if you are using ATG Consumer Commerce) or `atg.b2bcommerce.states.B2BorderStates` (if you are using ATG Business Commerce)<br><br>-- open<br><br>-- closed<br><br>If you specify “open,” then all orders whose states are specified in the `openStates` property of the `OrderLookup` component are returned. If you specify “closed,” then all orders whose states are specified in the `closedStates` property of the `OrderLookup` component are returned. You can override either list of states by using the optional `openStates` or `closedStates` input parameter (see below). |

| | |
|---|---|
| `openStates` | A comma-separated list of states that correspond to "open" state.<br><br>This parameter can be used in conjunction with the `state` input parameter when the `state` input parameter is set to "open." Use this optional parameter when you want to override the configured list of states in the `openStates` property of the `OrderLookup` component. |
| `closedStates` | A comma-separated list of states that correspond to "closed" state.<br><br>This parameter can be used in conjunction with the `state` input parameter when the `state` input parameter is set to "closed." Use this optional parameter when you want to override the configured list of states in the `closedStates` property of the `OrderLookup` component. |
| `sortBy` | A string that specifies an `Order` property by which to sort the orders.<br><br>This parameter can be used in conjunction with `userId`. When using this parameter, you can specify the name of any Order Repository property (that is, the name of any property defined in `orderrepository.xml`), such as `id`, `state`, or `submittedDate`. |
| `sortAscending` | True or false. This parameter is used in conjunction with the `sortBy` input parameter. If set to true, the `Order` objects in the resulting array are sorted in ascending order by the property specified in the `sortBy` input parameter. The default value is false. |
| `numOrders` | The number of orders to return for the given query. |
| `startIndex` | The index of the first order in the result set. This parameter is useful for cycling through a large number of orders. |
| `queryTotal` | Indicates whether the number of retrieved orders will be calculated into a total that's accessible through the `totalCount` and `total_count` output parameters. Setting this property to `false` prevents the total count from being generated, regardless of the value specified in the `queryTotal` property. Omitting this parameter causes the default value, `true`, to be used. Use this parameter to ensure that queries to the database are made only when necessary. |
| `queryTotalOnly` | Indicates whether the total number of orders and the orders themselves are produced from the servlet bean. Setting this parameter to `true` makes the total number of retrieved orders available through the `totalCount` and `total_count` output parameters. The orders themselves are not retrieved or accessible. Use this parameter to ensure that queries to the database are made only when necessary.<br><br>Omitting this parameter, which is the same as setting it to `false`, saves a list of order objects to the `output` open parameter as well as the total number of orders to the `totalCount` and `total_count` output parameters.<br><br>If `queryTotal=false` (orders, no total) and `queryTotalOnly=true` (total, no orders), a total is generated only as specified in the `queryTotalOnly` parameter. |

The OrderLookup servlet bean sets the following output parameters:

| Parameter | Description |
|---|---|
| `result` | The array of `Order` objects. If the `orderId` input parameter was used, then this parameter contains a single `Order` object. |
| `errorMsg` | If an error occurred, this is the detailed error message for the user. |
| `count` | The size of the array of `Order` objects. |
| `totalCount` | If the `queryTotal` property is set to true, this parameter indicates the total number of orders that meet the criteria for the order lookup. |
| `total_count` | Identical to the `totalCount` output parameter. |
| `startRange` | The index number that marks the beginning of a range of orders. For example, if 5 orders were returned from a given `OrderLookup` query, the `startRange` is set to 1. |
| `endRange` | The index number that marks the end of a range of orders. For example, if 5 orders were returned from a given `OrderLookup` query with a `startRange` of 6, the `endRange` is set to 10. |
| `nextIndex` | The index of the first order in the next set of results. If the `startIndex` or `numOrders` input parameter was null, then this parameter will be null. |
| `previousIndex` | The index of the first order in the previous set of results. If the `startIndex` or `numOrders` input parameter was null, then this parameter will be null. |

The OrderLookup servlet bean renders the following open parameters:

| Parameter | Description |
|---|---|
| `output` | The `oparam` rendered if the orders are successfully retrieved. |
| `empty` | The `oparam` rendered if there are no orders to return. |
| `error` | The `oparam` rendered if an error occurs. |

The following example describes how to use the OrderLookup servlet bean to retrieve all open orders for the current user and to display their IDs:

```
<dsp:droplet name="/atg/commerce/order/OrderLookup">
 <dsp:param bean="/atg/userprofiling/Profile.repositoryId" name="userId"/>
 <dsp:param value="open" name="state"/>
 <dsp:oparam name="output">
    <dsp:droplet name="/atg/dynamo/droplet/ForEach">
       <dsp:param param="result" name="array"/>
       <dsp:oparam name="outputStart">
            <OL>
       </dsp:oparam>
       <dsp:oparam name="outputEnd">
            </OL>
       </dsp:oparam>
       <dsp:oparam name="empty">
             No open orders.
       </dsp:oparam>
       <dsp:oparam name="output">
            <LI> <dsp:valueof param="element.id">no order number</dsp:valueof>
       </dsp:oparam>
     </dsp:droplet>
 </dsp:oparam>
 <dsp:oparam name="error">
        <span class=profilebig>ERROR:
          <dsp:valueof param="errorMsg">no error message</dsp:valueof>
        </span>
 </dsp:oparam>
</dsp:droplet>
```

The following example describes how to use the `OrderLookup` servlet bean to display information about a particular order with the ID of 123:

```
<dsp:droplet name="/atg/commerce/order/OrderLookup">
  <dsp:param value="123" name="orderId"/>
  <dsp:oparam name="error">
    <p>
    ERROR:
    <dsp:valueof param="errorMsg">no error message</dsp:valueof>
    <p>
  </dsp:oparam>
  <dsp:oparam name="output">
    <p>
    order #<dsp:valueof param="result.id">no order id</dsp:valueof>
    <p>
This order is in state:
<dsp:valueof param="result.stateAsString"/>
    <P>
    This order was placed on
      <dsp:valueof date="MMMMM d, yyyy" param="result.submittedDate"/>.
    <P>
  </dsp:oparam>
</dsp:droplet>
```

# 12 Implementing an Order Approval Process

B2B applications often require that customers' orders be reviewed by authorized persons who can approve or reject them. If your application implements an order approval process, you need to display to a given approver the orders that require his or her approval, and you need to provide some means to process the approver's decisions (of approval and/or rejection). `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.

This chapter provides information on how you can use these three components in the JSPs of your application to support the following parts of the order approval process:

**Displaying Orders Requiring Approval**

**Processing Approvals and Rejections**

**Displaying a History of Approved and Rejected Orders**

## Displaying Orders Requiring Approval

Use the `ApprovalRequiredDroplet` servlet bean 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`.

See the ApprovalRequiredDroplet entry in Appendix: ATG Commerce Servlet Beans for additional information.

## Processing Approvals and Rejections

Use the `ApprovalFormHandler` form handler (class `atg.b2bcommerce.approval.ApprovalFormHandler`) to process 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:

```
<dsp:input bean="ApprovalFormHandler.approveOrder" value=" Approve Order"
           type="submit"/>
<dsp:input bean="ApprovalFormHandler.rejectOrder" value=" Reject Order"
           type="submit"/>
```

If the `handleApproveOrder` method is called for `ApprovalFormHandler.approveOrder`, the `handleApproveOrder` method processes the approver's *approval* of the order. In contrast, if the `handleRejectOrder` method is called for `ApprovalFormHandler.rejectOrder`, the `handleRejectOrder` method processes the approver's *rejection* of the order.

The following JSP example illustrates how to use `ApprovalFormHandler` with a single order. In this example, the approver enters an order id and a comment (or message) to be associated with his approval or rejection of the order. He then approves or rejects the order by clicking the Approve Order submit button or the Reject Order submit button, respectively.

```
<dsp:importbean bean="/atg/commerce/approval/ApprovalFormHandler"/>
<dsp:form action="pendingapprovalOrders.jsp">
   <dsp:input bean="ApprovalFormHandler.ApproveOrderSuccessURL"
         value="pendingapprovalOrders.jsp" type="hidden"/>
   <dsp:input bean="ApprovalFormHandler.ApproveOrderErrorURL"
         value="pendingapprovalOrders.jsp" type="hidden"/>
   <dsp:input bean="ApprovalFormHandler.RejectOrderSuccessURL"
         value="pendingapprovalOrders.jsp" type="hidden"/>
   <dsp:input bean="ApprovalFormHandler.RejectOrderErrorURL"
         value="pendingapprovalOrders.jsp" type="hidden"/>
   Order Id <dsp:input bean="ApprovalFormHandler.orderId" size="10" value=""
            type="text"/>
   Approver Message <dsp:input bean="ApprovalFormHandler.approverMessage"
         size="500" value="" type="text"/>
   <dsp:input bean="ApprovalFormHandler.approveOrder" value="  Approve Order  "
         type="submit"/>
   <dsp:input bean="ApprovalFormHandler.rejectOrder" value="  Reject Order  "
         type="submit"/>
</dsp:form>
```

**Note**: In actual implementations, JSP files will be in web applications and not relative to doc roots.

## Displaying a History of Approved and Rejected Orders

Use the ApprovedDroplet servlet bean 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.

ApprovedDroplet takes the following input parameters:

- approverid: the ID of the current user profile; the approver. This parameter is required.
- startIndex: The index of the first order to return. If startIndex is null, then it defaults to 0. This parameter is optional and typically is used to break large result sets into manageable pieces.
- numOrders: The number of orders to return on the query. This parameter is optional, and typically is used to break large result sets into manageable pieces.

ApprovedDroplet sets the following output parameters:

- result: The array of Order objects.
- count: The number of Order objects in the result output parameter.
- totalCount: The total number of Order objects that satisfied the criteria.
- nextIndex: The index of the first order in the next set of results. If startIndex or numOrders was null, then this parameter will also be null.
- previousIndex: The index of the first order in the previous set of results. If startIndex or numOrders was null, then this parameter will also be null.

  nextIndex and previousIndex allow the user to cycle back and forth between result sets.
- startRange: The 1-based index of the first Order in the set of results.
- endRange: The 1-based index of the last Order in the set of results.
- errorMsg: The error message to display to the user if an error occurs.

It renders the following open parameters (oparams):

- output: This oparam renders the array of Order objects set in the result output parameter.
- empty: The oparam rendered if there are no orders that have been approved and/or rejected by the current user.
- error: The oparam rendered if an error occurs.

**Note:** The ApprovedDroplet servlet bean has a security feature that allows the current user, the approver, to view only the orders of customers for whom he or she is allowed to approve orders. This feature is enabled by default. To disable the feature, set the enableSecurity property to false.

The following JSP example retrieves from the repository the orders that have been approved and/or rejected by the current user, an approver, and lists each order's repository ID on the page.

```
<dsp:droplet name="ApprovedDroplet">
   <dsp:param bean="/atg/userprofiling/Profile.repositoryId" name="approverid"/>
   <dsp:param value="0" name="startIndex"/>
   <dsp:param value="10" name="numOrders"/>
   <dsp:oparam name="output">
      <dsp:droplet name="ForEach">
         <dsp:param param="result" name="array"/>
         <dsp:param value="order" name="elementName"/>
         <dsp:oparam name="output">
            <dsp:valueof param="order.repositoryId"/><br>
         </dsp:oparam>
         <dsp:oparam name="error">
            <dsp:valueof param="errorMsg"/><br>
         </dsp:oparam>
      </dsp:droplet>
   </dsp:oparam>
</dsp:droplet>
```

# 13 Filtering Commerce Item Collections

ATG Commerce extends the collection filtering feature provided in ATG Adaptive Scenario Engine by providing independent and chained collection filtering components that are designed to work with products.

This chapter describes collection filtering in the following sections:

**How Product Collection Filtering Works**

**Using ATG Collection Filtering Components**

**Filtering Multisite Gift and Wish Lists**

See these resources for more information on collection filtering:

- A general discussion of collection filtering is located in the *Filtering Collections* chapter of the *ATG Personalization Programming Guide*.
- A reference description of the `CollectionFilter` servlet bean is in *Appendix B: ATG Servlet Beans* of the *ATG Page Developer's Guide*.
- The API collection filtering documentation resides in the sections for the `atg.service.collections.filter` and `atg.commerce.collections.filter` packages of the *ATG API Reference*.

## How Product Collection Filtering Works

You can filter products from a collection using independent collection filters or filters in a chain. This example shows how to render a collection of products that satisfy the conditions defined in a chain of filters. The rendered collection is cached for future use.

Consider a Web page in which you want to display a list of active, available products to customers. To do this, use a collection filtering servlet bean (`ProductFilterDroplet`) in a JSP to access a collection filtering component (`ProductFilter`) that will then apply its chain of filters to a collection of products. See *CollectionFilter* for more on `ProductFilterDroplet` and other collection filtering servlet beans.

It is the responsibility of `ProductFilter` to chain together separate collection filters and to execute that chain. This example assumes that the `ProductFilter.filters` property identifies two collection filters: `StartEndDateFilter` and `InventoryFilter`. These filters collectively remove products that aren't available for sale (`StartEndDateFilter`) and aren't in stock (`InventoryFilter`).

The JSP code would look like this:

```
<dspel:droplet name="/atg/collections/filter/droplet/ProductFilterDroplet">
   <dsp:param name="collection" param="item.childproducts/>
   <dsp:param name="collectionIdentifierKey" value="catid-0067-hardscape"/>

   <dspel:oparam name="output">
   Featured Plants:<p>
      <dsp:droplet name="/atg/dynamo/droplet/ForEach">
         <dsp:param name="array" param="filteredCollection"/>

         <dsp:oparam name="output">
            <dspel:valueof param="element"/>
         </dspel:oparam>
      </dsp:droplet>
   </dspel:oparam>

   <dspel:oparam name="empty">
      There are currently no outdoor plants
   </dspel:oparam>
</dspel:droplet>
```

When this JSP executes, the `ProductFilterDroplet` passes the products to the `ProductFilter` component and executes the chain:

- The first component is `StartEndDateFilter`, which compares the current date to the values in the `startDate` and `endDate` properties for each product. Products that are not active (have not been "started" or have already been "ended") are discarded and the remaining products are passed to the `InventoryFilter`
- `InventoryFilter` corresponds with the Inventory Manager to determine the availability of all products in the slot. Products that are out of stock are removed from the collection.
- When caching is enabled on `ProductFilter`, the `FilterCache` component saves information that represents the prefiltered collection as well as the filtered result. Subsequent renderings of this JSP will compare the cached prefiltered collection with the current prefiltered one. When appropriate, the cached filtered result is used. For more information about caching, see the *Filtering Collections* chapter of the *ATG Personalization Programming Guide*.

## Using ATG Collection Filtering Components

Collection filtering components are components that reduce the objects in a collection based on a unique set of conditions. ATG Commerce includes three collection filtering components:

- Using InventoryFilter
- Using ExcludeItemsInCartFilter
- Using ProductFilter

- Using CartSharingFilter

These components receive a collection from another resource (scenario or servlet bean) and return the resultant collection to the calling resource. To involve a collection filtering component in a scenario, define a `Filter Slot Contents` action element to use a collection filtering component. See the *ATG Personalization Programming Guide* for information on this scenario action. Specific instructions for using servlet beans to access collection filtering components are included for each component in the following sections.

Keep in mind that although a collection can hold any kind of object, the collection filtering components described here are defined to work with `RepositoryItems` of type `Product`. Any non-`Product` items in the collection are ignored by the component and included in the result set.

## Using InventoryFilter

The `/atg/registry/CollectionFilters/InventoryFilter` component, which is a component of class `atg.commerce.collections.filter.InventoryFilter`, is used to eliminate products from a collection that have a specific inventory availability.

`InventoryFilter` compares inventory status as defined in the `InventoryManager` for each product's SKUs to the statuses specified in the `InventoryFilter.IncludeInventoryStates` property. If any SKU for a product has a status included in the `IncludeInventoryStates` property, the product remains in result set collection. By default, `InventoryFilter.IncludeInventoryStates` has values `1000` (in stock), `1002` (pre-orderable), and `1003` (back orderable) so any product containing one of these values is added to the collection that's returned.

You can access the `InventoryFilter` component through the `InventoryFilterDroplet` servlet bean (directly) and the `ProductFilterDroplet` (as part of a filter chain). Although you have the option to cache collection filter content, caching is discouraged for `InventoryFilter` if you are using an Inventory Manager, like `CachingInventoryManager`, that has its own caching mechanism. For more information on caching, see the *Filtering Collections* chapter of *ATG Personalization Programming Guide*.

## Using ExcludeItemsInCartFilter

The `/atg/registry/CollectionFilters/ExcludeItemsInCartFilter` component, which is a component of class `atg.commerce.collections.filter.ExcludeItemsInCartFilter`, is used to remove any products from a collection that are in a user's shopping cart. This component relies on the `shoppingCartPath` property to locate the current user's shopping cart. The default value is `/atg/commerce/ShoppingCart`.

You can access the `ExcludeItemsInCartFilter` component through the `ExcludeItemInCartFilterDroplet` servlet bean (directly) Although you have the option to cache collection filter content, you are advised against caching the results from `ExcludeItemsInCartFilter` because it's unlikely that the cached content will be used again, making caching a waste of resources. For more information on caching, see the *Filtering Collections* chapter of *ATG Personalization Programming Guide*.

### Using ProductFilter

The /atg/registry/CollectionFilters/ProductFilter component, which is a component of class atg.service.collections.filter.ChainedFilter, is used to create a chain of collection filters that apply their conditions to a collection of products.

ProductFilter passes the products to each filter defined in the ProductFilter.filters property successively and produces a final collection that satisfies the conditions specified by each collection filter. By default ProductFilter.filters is set to StartEndDateFilter and InventoryFilter.

You can access the ProductFilter component on a JSP using the ProductFilterDroplet servlet bean. When you ProductFilterDroplet, the resultant content is cached by default (cacheEnabled is set to true). For more information on the ChainedFilter class and on caching, see the *Filtering Collections* chapter of the *ATG Personalization Programming Guide*.

### Using CartSharingFilter

If you are using ATG's multisite feature, the /atg/registry/CollectionFilters/CartSharingFilter component, which is a component of class atg.commerce.collections.filter.ItemSiteFilter, filters input item collections by their site IDs.

This filter returns only products that are within the same sharing group, based on the atg.ShoppingCart shareable type configuration (see *Configuring Commerce for Multisite* in the *ATG Commerce Programming Guide*). Two additional properties, includeDisabledSites and includeInactiveSites, determine which site states are taken into account (both are false by default).

You can access the CartSharingFilter component in a JSP; see the CollectionFilter section for information.

## Filtering Multisite Gift and Wish Lists

ATG 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 the filtering functionality described in this section.

Filtering is particularly important for wish lists. Customers can only have one wish list, making it more likely that items from multiple sites will exist in a single wish list. To limit wish list item display to only those items that are appropriate for the site context, you must filter out any items affiliated with sites that are outside of the site context.

Two components facilitate the filtering of gift lists and gift items:

- The `/atg/commerce/collections/filter/droplet/GiftlistSiteFilterDroplet` calls the `GiftlistSiteFilter` component to filter a specified collection of gift lists/gift items and then renders the filtered results.
- The `/atg/registry/CollectionFilters/GiftlistSiteFilter` component filters collections of gift lists or gift items. It returns only those lists/items that are appropriate for the site context.

The `GiftlistSiteFilterDroplet` is a globally-scoped instance of the class `atg.service.collections.filter.droplet.CollectionFilter` designed to render a filtered set of gift lists/gift items. It has the following properties:

- `filter`: Reference to the `/atg/registry/CollectionFilters/GiftlistSiteFilter` component.
- `extraParameterNames`: A comma-separated list that identifies the additional parameters, namely `siteIds` and `siteScope`, that a JSP can specify as input parameters to `GiftlistSiteFilterDroplet`. `GiftlistSiteFilterDroplet` passes these parameters on to `GiftlistSiteFilter` so that it can filter gift lists based on site context. If the JSP doesn't pass values for these parameters, `GiftlistSiteFilter` defaults to using the current site and the `GiftlistManager.siteScope` value, respectively.

`GiftlistSiteFilter` is a globally-scoped component of class `atg.commerce.gifts.GiftlistSiteFilter`. This class extends the generic collection filtering class `atg.service.collections.filter.CachedCollectionFilter` by overriding the `generateFilteredCollection()` method to take site scope and site ID parameters into consideration when filtering gift lists or gift items. `GiftlistSiteFilter` resolves the site scope and site ID values as follows:

- If a site scope is passed in, `GiftlistSiteFilter` uses that scope during filtering. If not, `GiftlistSiteFilter` calls the `GiftlistManager` and uses its `siteScope` setting for filtering. Note that, with the default `siteScope` setting of `all`, all gift lists/items are always returned and no filtering occurs, so you should not use gift list/item filters unless you are using a `siteScope` other than `all`.
- If a list of site IDs is passed in, `GiftlistSiteFilter` uses that list during filtering. If no site IDs are passed in, the current site's `siteId` is used during filtering.

When filtering gift lists, the `GiftlistSiteFilter` determines the site scope and then compares the `siteId`'s of the gift lists/gift items in the repository to the IDs in its list, as shown in the following table:

| **Compatibility Test** | **All** | **Current** | **ShareableType ID** |
|---|---|---|---|
| Gift list/gift item's `siteId` is in the site IDs list | Include gift list/gift item in filtered results | Include gift list/gift item in filtered results | Include gift list/gift item in filtered results |

| | | | |
|---|---|---|---|
| Gift list/gift item's `siteId` is not in the site IDs list | Include gift list/gift item in filtered results | Do not include gift list/gift item in filtered results | Include gift list/gift item in filtered results if the gift list/gift item's `siteId` is in the specified sharing group (for example, the shopping cart sharing group) with any of the sites in the site IDs list. |
| Gift list/gift item's `siteId` is null<br><br>**Note:** This is also the wish list case. | The gift list/item is universal and should be included in filtered results | The gift list/item is universal and should be included in filtered results | The gift list/item is universal and should be included in filtered results |

Properties for the `GiftlistSiteFilter` component include:

- `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.
- `includeDisabledSites`: If `true`, the filter does not filter out items that appear on disabled sites. The default is `false`.
- `includeInactiveSites`: If `true`, the filter does not filter out items that appear on inactive sites. The default is `false`.

Note that, unlike the `GiftlistSearch` form handler, the `GiftlistSiteFilter` component does not have `siteScope` or `siteIds` properties. Instead, the site scope and list of site IDs are passed to `GiftlistSiteFilter` by the `GiftlistSiteFilterDroplet`, as described below.

This JSP excerpt shows one example of how you can use `GiftlistSiteFilterDroplet` to filter gift lists. No site scope is passed in, so the `GiftlistSiteFilter` uses the `GiftlistManager` component's `siteScope`, which for the purposes of this example is set to the `atg.ShoppingCart` shareable type component. Also, no site IDs are provided, so the filtered gift lists will come from the current site and sites that share a shopping cart with the current site only.

```
<dsp:droplet
name="/atg/commerce/collections/filter/droplet/GiftlistSiteFilterDroplet">
  <&-- Specify the collection to filter --%>
  <dsp:param name="collection" bean="Profile.giftlists"/>

  <dsp:oparam name="output">

    <%-- Iterate through the collection. --%>
      <dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
      <dsp:droplet name="ForEach">
        <dsp:param name="array" param="filteredCollection"/>
        <dsp:oparam name="output">
          <dsp:setvalue param="giftList" paramvalue="element"/>
          <dsp:getvalueof var="eventName" param="giftList.eventName"/>
          <c:out value="${eventName}"/>
        </dsp:oparam>
      </dsp:droplet>

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

This JSP excerpt filters a collection of wish list items. A site scope value of `current` is passed to the filter along with a set of site IDs, resulting in a collection of items from the specified sites only.

```
<dsp:droplet
name="/atg/commerce/collections/filter/droplet/GiftlistSiteFilterDroplet">
  <&-- Specify the collection to filter and the site scope. --%>
  <dsp:param name="collection" bean="Profile.wishlist.giftlistItems"/>
  <dsp:param name="siteScope" value="current"/>
  <dsp:param name="siteIds" value="siteA,siteB"/>

  <dsp:oparam name="output">

    <%-- Iterate through the collection. --%>
      <dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
      <dsp:droplet name="ForEach">
        <dsp:param name="array" param="filteredCollection"/>

        <dsp:oparam name="output">
          <dsp:setvalue param="giftItem" paramvalue="element"/>
          <dsp:getvalueof var="displayName" param="giftItem.displayName"/>
          <c:out value="${displayName}"/>
        </dsp:oparam>
      </dsp:droplet>

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

**Notes:**

- The `GiftlistSiteFilter` and `GiftlistSiteFilterDroplet` components can be configured to use caching to improve filtering performance, just as you would for any filter based on the `CollectionFilter` class. See *Caching Filtered Content* in the *ATG Personalization Programming Guide*.
- The `GiftlistSiteFilter` component can be used independently of the `GiftlistSiteFilterDroplet`. To do so, you must pass the `siteIds` and `siteScope` values in the `pExtraParameters` map when calling the `GiftlistSiteFilter.generateFilteredCollection()` method (when using the

droplet, the droplet creates the `pExtraParameters` map before it calls the `GiftlistSiteFilter` component). For more information, see the *ATG API Reference*.

# 14 Using ATG Commerce Portal Gears

If you are an ATG Commerce *and* ATG Portal user, you can use ATG Commerce gears in the portal pages of your commerce site to provide customers with a personalized gateway to their order information. For example, the Order Status gear provides customers with direct access to their five most recently placed orders, with access to all orders just one click away.

This chapter provides information on the portal gears (sometimes referred to as "portlets") that are provided with ATG Business Commerce and/or ATG Consumer Commerce. It includes the following sections:

**Order Status Gear**

**Order Approval Gear**

The chapter assumes that you are familiar with the gear development and administration concepts covered in the *ATG Portal Development Guide* and the *ATG Portal Administration Guide*.

**Note:** The ATG Commerce portal gears are packaged into individual modules that include Web applications you need to deploy on your application server.

## Order Status Gear

The Order Status gear makes current and historical order information available to portal end users.

This section explains how to run and use the Order Status gear and describes its implementation and default configuration. It covers the following topics:

Setting Up the Order Status Gear

Using the Order Status Gear

Configuring the Order Status Gear

Order Status Gear Implementation

### Setting Up the Order Status Gear

The Order Status gear is part of both ATG Business Commerce and ATG Consumer Commerce.

To set up the Order Status gear:

1. During application assembly, specify the ATG Business Commerce, ATG Consumer Commerce module, Motorprise module, or your own commerce application that contains order data. Also specify the `CommerceGears.orderstatus` module (which requires ATG Portal).

   For example, the following modules represent the Order Status gear and Motorprise:

   ```
   CommerceGears.orderstatus MotorpriseJSP
   ```

   For a list of module names and assembly instructions, see the *ATG Programming Guide*.

2. Deploy your application as instructed by your application server documentation.

3. Register the Order Status gear with the Portal Administration Framework (PAF) by uploading its gear manifest file to the PAF. The gear manifest is located at `<ATG10dir>\CommerceGears\orderstatus\orderstatus-manifest.xml`.

   For detailed information on how to register a gear, see the *Portal Administration* chapter in the *ATG Portal Administration Guide*.

4. Create a community whose members will use the gear and add the gear to one of the community's pages.

   For detailed information on how to create a community and add a gear to one of its pages, see the *Community Administration* chapter in the *ATG Portal Administration Guide*.

5. Point your browser to the URL for the community page that contains the Order Status gear (for example, `http://`*`host:port`*`/portal/mycommunity/home`).

   Before the Order Status gear is displayed, you must log in to the page. For information on the default port, see the *ATG Installation and Configuration Guide*.

6. Log in to the page as the following user to explore the gear:

   - Stuart Lee, a Motorprise buyer. Stuart's username and password are `stuart:stuart`.

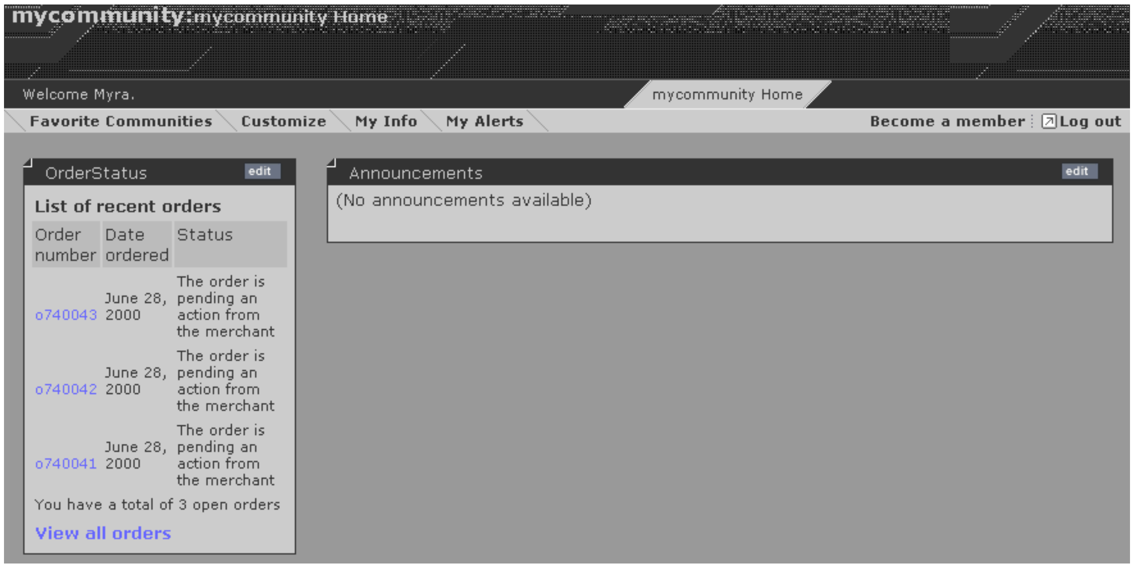
   If you are running the Order Status gear with your own commerce application, you'll need to log in as an appropriate user.

## Using the Order Status Gear

When an end user logs in to a portal page that includes the Order Status gear, the user initially views the gear in Shared mode. By default, the gear displays the user's five most recently placed orders, providing each order's number, order date, and current status. Additionally, the gear displays the total count of the user's open orders. From this page, the user can do any of the following:

- Click any order number to display the order details page for the given order.
- Click the View All Orders link to display all of the user's orders.
- Click the Edit button in the gear's top, right corner to configure the gear's user parameters.

The following figure illustrates the Order Status gear in Shared mode.



List of recent orders

| Order number | Date ordered | Status |
|---|---|---|
| o740043 | June 28, 2000 | The order is pending an action from the merchant |
| o740042 | June 28, 2000 | The order is pending an action from the merchant |
| o740041 | June 28, 2000 | The order is pending an action from the merchant |

You have a total of 3 open orders

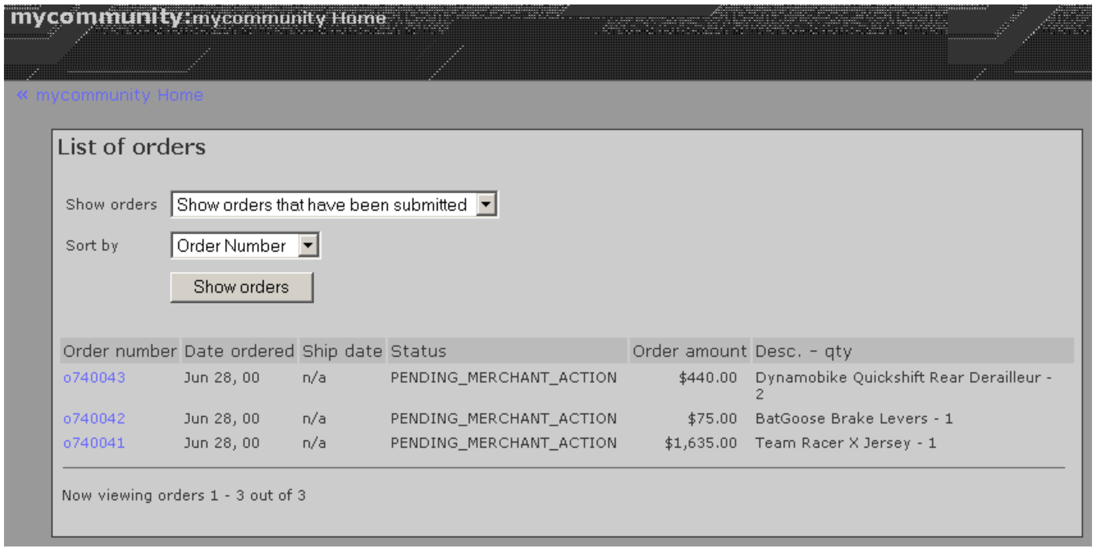
View all orders

(No announcements available)

*Shared view of Order Status gear*

When the user clicks the View All Orders link, the portal page is re-rendered and displays the Order Status gear in Full Page mode. The gear displays all of the user's orders, providing each order's number, order date, ship date (if applicable), current status, amount, and description. By default, if the user's orders exceed 10 in number, they are displayed across multiple pages.

The following figure illustrates the Order Status gear in Full Page mode.



List of orders

Show orders: Show orders that have been submitted

Sort by: Order Number

| Order number | Date ordered | Ship date | Status | Order amount | Desc. - qty |
|---|---|---|---|---|---|
| o740043 | Jun 28, 00 | n/a | PENDING_MERCHANT_ACTION | $440.00 | Dynamobike Quickshift Rear Derailleur - 2 |
| o740042 | Jun 28, 00 | n/a | PENDING_MERCHANT_ACTION | $75.00 | BatGoose Brake Levers - 1 |
| o740041 | Jun 28, 00 | n/a | PENDING_MERCHANT_ACTION | $1,635.00 | Team Racer X Jersey - 1 |

Now viewing orders 1 - 3 out of 3

*Full view of Order Status gear*

In Full Page mode, the user can change the display of the orders by specifying the state of the orders to display (for example, Shipped or Submitted). The user can also specify a sorting instruction for the orders (order number, state, date ordered). Clicking the Show Orders button then re-displays the list of orders according to the specified instructions. As in Shared mode, the user can click any order number to display the order details page for the given order.

## Configuring the Order Status Gear

The Order Status gear includes three gear configuration pages with which to customize the function and display of the gear:

- The `installConfig` page for portal administrators
- The `instanceConfig` page for community leaders
- The `userConfig` page for portal end users

The table below describes the instance parameters for the Order Status gear that can be configured on the `installConfig` page. Typically, they are set by the portal administrator.

| **Instance parameter** | **Description** | **Default Value** |
|---|---|---|
| `OrderPage` | The URL of the order details page used by the commerce application; this page is displayed when the portal end user clicks an order number in the Order Status gear.<br><br>**Important:** By default, this parameter is set to a generic URL. If you're running the gear with the Motorprise store, you must change this parameter to `/Motorprise/en/user/order.jsp`.<br><br>If you're running the gear with your own commerce application, you must change it to the JSP that is appropriate for the application. Note that if the JSP does not support multiple locales, you'll need to configure a separate gear instance for each required locale. | `en/user/order.jsp` |

The table below describes the instance parameters for the Order Status gear that can be configured on the `instanceConfig` page. Typically, they are set by the community leader.

Each of the instance parameters stores a boolean value that indicates whether, in Full Page mode, the portal end user can filter and display only those orders in the specified order state. If a parameter is set to true, then the associated state is included in the filter drop-down list.

| Instance parameter | Description | Default Value |
|---|---|---|
| `ShowShippedFilterFull` | Indicates whether the end user can filter and display only Shipped orders. | true |
| `ShowSubmittedFilterFull` | Indicates whether the end user can filter and display only Submitted orders. | true |
| `ShowApprovedFilterFull` | Indicates whether the end user can filter and display only Approved orders. | true |
| `ShowRejectedFilterFull` | Indicates whether the end user can filter and display only Rejected orders. | true |
| `ShowPendingApprovalFilterFull` | Indicates whether the end user can filter and display only Pending Approval orders. | true |
| `ShowPendingRemoveFull` | Indicates whether the end user can filter and display only Pending Remove orders. | true |
| `ShowRemovedFull` | Indicates whether the end user can filter and display only Removed orders. | true |
| `ShowPendingCustomerActionFull` | Indicates whether the end user can filter and display only Pending Customer Action orders. | true |
| `ShowPendingCustomerReturnFull` | Indicates whether the end user can filter and display only Pending Customer Return orders. | true |

The table below describes the user parameters for the Order Status gear that can be configured by the portal end user on the `userConfig` page.

| User Parameter | Description | Default Value |
|---|---|---|
| `NumberOfOrdersFull` | The maximum number of orders to display *per page* in Full Page mode. | 10 |

| | | |
|---|---|---|
| `NumberOfOrdersShared` | The maximum number of orders to display in Shared mode. If set to zero, recent order information is not displayed. | 5 |
| `ShowOpenOrdersShared` | Boolean value that indicates whether to show in Shared mode the number of open orders placed by the user.<br><br>"Open" orders are those orders whose state is specified in `OrderLookup.openStates`. (For more information on the `OrderLookup` component, see the Order Status Gear Implementation section.) | true |

## Order Status Gear Implementation

The Order Status gear relies upon existing ATG Portal and core ATG Commerce functionality. The subsections that follow describe various aspects of its implementation.

### *Gear Modes and Display Modes*

The following table lists the gear modes used in the Order Status gear, as well as their corresponding display modes, device outputs, and gear content and configuration pages.

| **Gear Mode** | **Display Mode** | **Device Output** | **Page Fragment** |
|---|---|---|---|
| `content` | Shared | HTML | `OrderStatusShared.jsp` |
| | Full | HTML | `OrderStatusFull.jsp` |
| `installConfig` | Full | HTML | `installConfig.jsp` |
| `instanceConfig` | Full | HTML | `instanceConfig.jsp` |
| `userConfig` | Full | HTML | `userConfig.jsp` |

For general information about gear modes, display modes, and device outputs, see the *Designing a Gear* chapter in the *ATG Portal Development Guide*. For general information about creating gear content and configuration pages, see the *Creating Gear Page Fragments* chapter in the same guide.

### *Components*

The Order Status gear makes use of the following major components:

| **Component** | **Description** |
|---|---|
| `/atg/portal/gear/GearConfigFormHandler` | Class `atg.portal.framework.GearConfigFormHandler`.<br><br>`GearConfigFormHandler` is provided with the PAF. It is used in the Order Status gear configuration pages to create and manage the forms with which to set the gear's instance and user parameters.<br><br>For more information on `GearConfigFormHandler`, see the *Gears and the Portal Application Framework* and *Creating Gear Page Fragments* chapters in the *ATG Portal Development Guide*. |
| `/atg/commerce/states/OrderStates` | Class `atg.commerce.states.OrderStates` for ATG Consumer Commerce; class `atg.b2bcommerce.states.B2BOrderStates` for ATG Business Commerce.<br><br>The `OrderStates` component is used in `instanceConfig.jsp` to retrieve a list of possible order states. For each possible order state, a checkbox is rendered. If the community leader checks the checkbox, then the associated order state is included in the list of states by which the portal end user can filter orders in Full Page mode.<br><br>For more information on `OrderStates`, see the *ATG Commerce States* section of the *Working With Purchase Process Objects* chapter in the *ATG Commerce Programming Guide*. |

| | |
|---|---|
| `/atg/commerce/gears/orderstatus/OrderStatusGear` | Class `atg.commerce.gears.orderstatus.OrderStatusFormHandler.`<br><br>The `OrderStatusGear` form handler is used in both the Shared and Full Page content pages. On `OrderStatusShared.jsp`, the component is used to display orders whose states match those specified in its `sharedViewStates` property. By default, `OrderStatusGear.sharedViewStates` is set to the following:<br><br>`submitted,\`<br>`processing,\`<br>`pending_merchant_action,\`<br>`pending_customer_action,\`<br>`no_pending_action`<br><br>On `OrderStatusFull.jsp`, it is used to create and manage the form that the end user can use to specify order state and sorting criteria when displaying orders. |
| `/atg/commerce/order/OrderLookup` | Class `atg.commerce.order.OrderLookup` for ATG Consumer Commerce; class `atg.b2bcommerce.order.B2BOrderLookup` for ATG Business Commerce.<br><br>The `OrderLookup` servlet bean is used in the gear content pages to retrieve and display the user's orders.<br><br>For more information on `OrderLookup`, see the Implementing Order Retrieval chapter. |

### *Tag Libraries*

The Order Status gear uses the following standard tag libraries:

- Core tag library
- DSP tag library
- PAF tag library
- Jakarta's i18n tag library

For information on the DSP tag libraries, see the *ATG Page Developer's Guide*. For information on the PAF tag library and Jakarta's i18n tag library, see the *ATG Portal Development Guide*.

No custom tag libraries were written for this gear.

# Order Approval Gear

The Order Approval gear displays the orders requiring approval by the current approver and, if needed, provides a mechanism for approving and/or rejecting those orders.

This section explains how to run and use the Order Approval gear and describes its implementation and default configuration. It covers the following topics:

Setting Up the Order Approval Gear

Using the Order Approval Gear

Configuring the Order Approval Gear

Order Approval Gear Implementation

**Note:** The Order Approval gear is installed with ATG Business Commerce. You must run both ATG Business Commerce and ATG Portal to run the Order Status gear.

## Setting Up the Order Approval Gear

To access the Order Approval gear:

1. During application assembly, specify ATG Business Commerce, Motorprise (or your own commerce application that contains order data), and the `CommerceGears.orderapproval` module (which requires ATG Portal). For example:

   ```
   CommerceGears.orderapproval MotorpriseJSP
   ```

   For a list of ATG modules and assembly instructions, see the *ATG Programming Guide*.

2. Deploy your application as instructed by your application server documentation.

3. Register the Order Approval gear with the Portal Administration Framework (PAF) by uploading its gear manifest file to the PAF. The gear manifest file is located at `<ATG10dir>\CommerceGears\orderapproval\orderapproval-manifest.xml`.

   For detailed information on how to register a gear, see the *Portal Administration* chapter in the *ATG Portal Administration Guide*.

4. Create a community whose members will use the gear and add the gear to one of the community's pages.

   For detailed information on how to create a community and add a gear to one of its pages, see the *Community Administration* chapter in the *ATG Portal Administration Guide*.

5. Point your browser to the URL for the community page that contains the Order Approval gear (for example, `http://`*`hostname:port`*`/portal/mycommunity/home`).

   Before the Order Approval gear is displayed, you must log in to the page as a user who is an approver. For information on the default port, see the *ATG Installation and Configuration Guide*.

6. Log in to the page as Ernesto Hernandez, a Motorprise approver, to explore the gear. Ernesto's username and password are `ernesto:ernesto`.

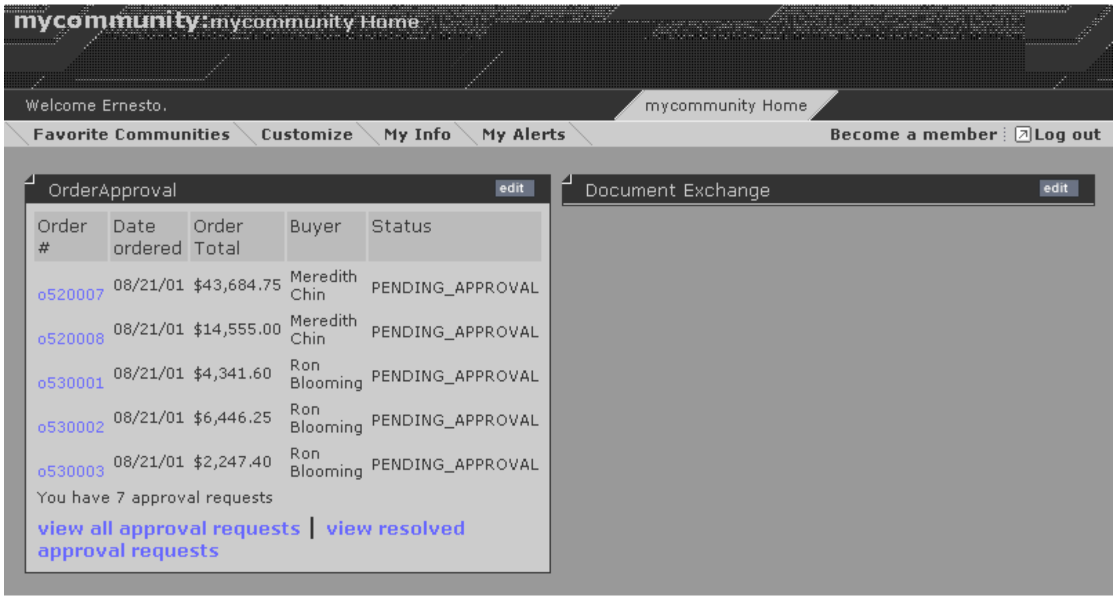
   If you're running the Order Approval gear with your own commerce application, you'll need to log in as an appropriate user.

## Using the Order Approval Gear

When an end user logs in to a portal page that includes the Order Approval gear, the user initially views the gear in Shared mode. By default, the gear displays the five most recently placed orders that require the user's approval, providing each order's number, order date, total cost, buyer, and current status (which is PENDING_APPROVAL). Additionally, the gear displays the total number of orders that require the user's approval. From this page, the user can do any of the following:

- Click any order number to display the order details page for the order.
- Click the View All Approval Requests link to display all the orders that require the user's approval.
- Click the View Resolved Approval Requests link to display all the orders which the user has already approved or rejected.
- Click the Edit button in the gear's top, right corner to configure the gear's user parameters.

The following figure illustrates the Order Approval gear in Shared mode.



| Order # | Date ordered | Order Total | Buyer | Status |
|---|---|---|---|---|
| o520007 | 08/21/01 | $43,684.75 | Meredith Chin | PENDING_APPROVAL |
| o520008 | 08/21/01 | $14,555.00 | Meredith Chin | PENDING_APPROVAL |
| o530001 | 08/21/01 | $4,341.60 | Ron Blooming | PENDING_APPROVAL |
| o530002 | 08/21/01 | $6,446.25 | Ron Blooming | PENDING_APPROVAL |
| o530003 | 08/21/01 | $2,247.40 | Ron Blooming | PENDING_APPROVAL |

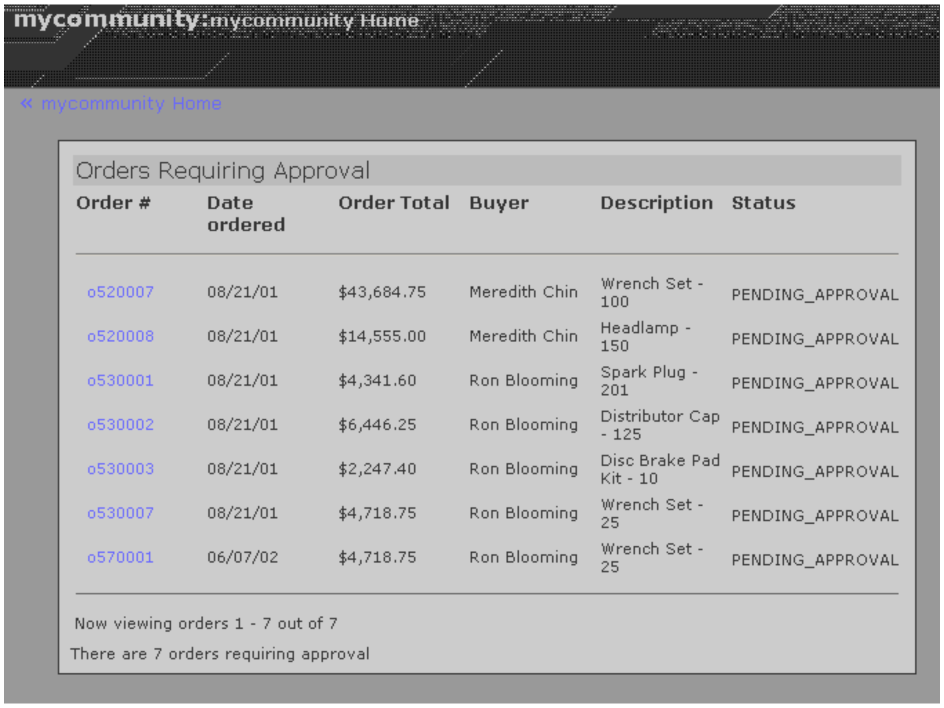
You have 7 approval requests

view all approval requests | view resolved approval requests

*Shared view of Order Approval gear*

When the user clicks the View All Approval Requests link, the portal page is re-rendered and displays the *Order List* page in Full Page mode. By default, on this page the gear lists all of the orders that require the user's approval, providing each order's number, order date, total cost, buyer, and current status. If the number of orders that require the user's approval exceeds 10, they are displayed across multiple pages.

The following figure illustrates the *Order List* page.



Orders Requiring Approval

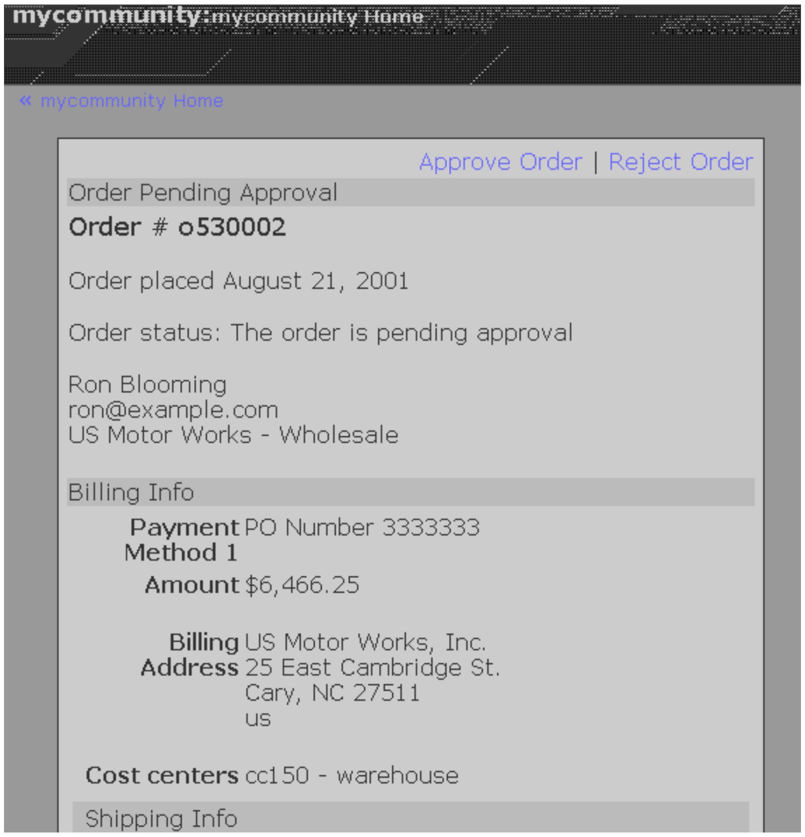
| Order # | Date ordered | Order Total | Buyer | Description | Status |
|---|---|---|---|---|---|
| o520007 | 08/21/01 | $43,684.75 | Meredith Chin | Wrench Set - 100 | PENDING_APPROVAL |
| o520008 | 08/21/01 | $14,555.00 | Meredith Chin | Headlamp - 150 | PENDING_APPROVAL |
| o530001 | 08/21/01 | $4,341.60 | Ron Blooming | Spark Plug - 201 | PENDING_APPROVAL |
| o530002 | 08/21/01 | $6,446.25 | Ron Blooming | Distributor Cap - 125 | PENDING_APPROVAL |
| o530003 | 08/21/01 | $2,247.40 | Ron Blooming | Disc Brake Pad Kit - 10 | PENDING_APPROVAL |
| o530007 | 08/21/01 | $4,718.75 | Ron Blooming | Wrench Set - 25 | PENDING_APPROVAL |
| o570001 | 06/07/02 | $4,718.75 | Ron Blooming | Wrench Set - 25 | PENDING_APPROVAL |

Now viewing orders 1 - 7 out of 7

There are 7 orders requiring approval

*Full view of Order Approval gear (Order List page displayed)*

On the *Order List* page, the user can click any order number to display the order details page for the order. If the Order Approval gear is configured to use the approval process of the running commerce application, clicking an order number displays an order details page of the commerce application. If the Order Approval gear is configured to use its own approval process, clicking an order number displays the gear's *Order Details* page, which is shown in the following figure.



Order Pending Approval

**Order # o530002**

Order placed August 21, 2001

Order status: The order is pending approval

Ron Blooming
ron@example.com
US Motor Works - Wholesale

Billing Info

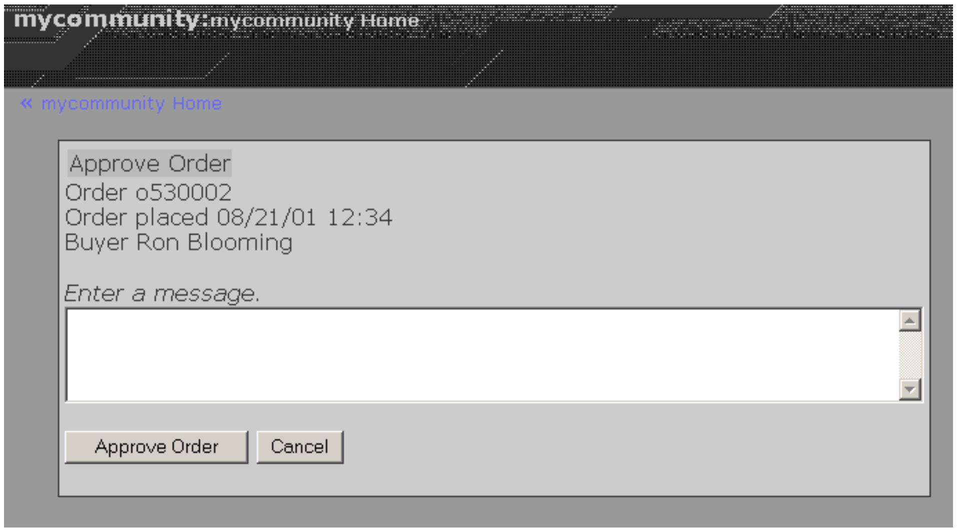
| | |
|---|---|
| **Payment Method 1** | PO Number 3333333 |
| **Amount** | $6,466.25 |
| **Billing Address** | US Motor Works, Inc.<br>25 East Cambridge St.<br>Cary, NC 27511<br>us |
| **Cost centers** | cc150 - warehouse |

Shipping Info

*Full view of Order Approval gear (Order Details page displayed)*

By default, the gear displays the order's basic, billing, and shipping information on the *Order Details* page. The approver can approve the order by clicking the Approve Order link or reject the order by clicking the Reject Order link.

If the approver clicks the Approve Order link, the gear displays the *Order Approval* page, which is shown in the following figure.



Approve Order
Order o530002
Order placed 08/21/01 12:34
Buyer Ron Blooming

Enter a message.

*Full view of Order Approval gear (Order Approval page displayed)*

On the *Order Approval* page, the approver can enter a reason for the approval decision and complete the approval by clicking the Approve Order button.

Note that if the approver clicks the Reject Order link on the *Order Details* page, the *Order Rejection* page is rendered instead. In either situation, once the approver approves or rejects the order, the approver completes the process by confirming the decision on a confirmation page.

## Configuring the Order Approval Gear

The Order Approval gear includes three gear configuration pages with which to customize the function and display of the gear:

- The `installConfig` page for portal administrators
- The `instanceConfig` page for community leaders
- The `userConfig` page for portal end users

The table below describes the instance parameters for the Order Approval gear that can be configured on the `installConfig` page. Typically, they are set by the portal administrator.

| **Instance parameter** | **Description** | **Default value** |
|---|---|---|
| `UseOrderApprovalOfGear` | Indicates whether to use the gear's order approval process. If set to false, the approval process of the commerce application is used instead. | true |

| | | |
|---|---|---|
| `ApprovedOrderPageURL` | The URL of the order details page used by the commerce application for approved and rejected orders. This page is displayed when the approver clicks the order number of an order that he or she has *already* approved or rejected.<br><br>**Important:** By default, this parameter is set to a generic URL. If you're running the gear with the Motorprise store, you must change this parameter to `/Motorprise/en/user/order.jsp`.<br><br>If you're running the gear with your own commerce application, you must change it to the JSP that is appropriate for the application. Note that if the JSP does not support multiple locales, you'll need to configure a separate gear instance for each required locale. | `checkout/order.jsp` |
| `PendingApprovalOrderPageURL` | The URL of the order details page used by the running commerce application for orders that require approval.<br><br>If the `UseOrderApprovalOfGear` parameter is set to false (described above), this page is displayed when the portal end user clicks the order number of an order that requires approval (that is, the order's status is PENDING_APPROVAL).<br><br>**Important:** By default, this parameter is set to a generic URL. If you're running the gear with the Motorprise store, you must change this parameter to `/Motorprise/en/user/order_pending_approval.jsp`.<br><br>If you're running the gear with your own commerce application, you must change it to the JSP that is appropriate for the application. Note that if the JSP does not support multiple locales, you'll need to configure a separate gear instance for each required locale. | `checkout/order.jsp` |

The table below describes the instance parameters for the Order Approval gear that can be configured on the `instanceConfig` page. Typically, they are set by the community leader.

Note that these instance parameters determine the function and display of the content pages that are used in the gear's order approval process. Consequently, if the gear's approval process isn't being used (that is, the approval process of the commerce application is being used instead), then these parameters have no relevance or impact on how the gear functions.

| **Instance parameter** | **Description** | **Default value** |
|---|---|---|
| `ShowOrderInfoInDetails` | Boolean value that indicates whether to display the order's basic information on the gear's *Order Details* page. | true |
| `ShowBillingInfoInDetails` | Boolean value that indicates whether to display the order's billing information on the gear's *Order Details* page. | true |
| `ShowShippingInfoInDetails` | Boolean value that indicates whether to display the order's shipping information on the gear's *Order Details* page. | true |
| `ShowOrderInfoInApprove` | Boolean value that indicates whether to display the order's basic information on the gear's *Order Approval* page. | true |
| `ShowMessageInApprove` | Boolean value that indicates whether to provide a message box on the gear's *Order Approval* page. | true |
| `ShowOrderInfoInReject` | Boolean value that indicates whether to display the order's basic information on the gear's *Order Rejection* page. | true |
| `ShowMessageInReject` | Boolean value that indicates whether to provide a message box on the gear's *Order Rejection* page. | true |

The table below describes the user parameters for the Order Approval gear that can be configured by the portal end user on the `userConfig` page.

| User parameter | Description | Default value |
|---|---|---|
| `NumberOfOrdersShared` | The maximum number of orders to display in Shared mode. If set to zero, no recent orders that require approval are displayed. | 5 |
| `ShowPendingApprovalCountShared` | Boolean value that indicates whether to show the total number of orders that require approval in Shared mode. | true |
| `NumberOfOrdersPerPageFull` | The maximum number of orders to display *per page* in Full Page mode. (If the total number of orders that require approval exceeds this number, the orders are split across multiple pages.) | 10 |
| `NumberOfOrdersFull` | The maximum total number of orders to display in Full Page mode. If set to –1, then all orders are displayed. | -1 |

## Order Approval Gear Implementation

The Order Approval gear relies upon existing ATG Portal, core ATG Commerce, and ATG Business Commerce functionality. The subsections that follow describe various aspects of its implementation.

### *Gear Modes and Display Modes*

The following table lists the gear modes used in the Order Approval gear, as well as their corresponding display modes, device outputs, and JSP fragments.

| Gear Mode | Display Mode | Device Output | Page Fragment |
|---|---|---|---|
| `content` | Shared | HTML | `OrderApprovalShared.jsp` |
| | Full | HTML | `OrderApprovalFull.jsp` |
| `installConfig` | Full | HTML | `installConfig.jsp` |
| `instanceConfig` | Full | HTML | `instanceConfig.jsp` |
| `userConfig` | Full | HTML | `userConfig.jsp` |

Note that `OrderApprovalFull.jsp` includes several other page fragments provided with the Order Approval gear, for example, `orderDetail.jsp` and `approveOrder.jsp`. You can find these JSPs in `<ATG10dir>/CommerceGears/orderapproval/src/orderapproval.war/web/html/content/`.

For general information about gear modes, display modes, and device outputs, see the *Designing a Gear* chapter in the *ATG Portal Development Guide*. For general information about creating gear content and configuration pages, see the *Creating Gear Page Fragments* chapter in the same guide.

### *Components*

The Order Approval gear makes use of the following major components:

| Component | Description |
|---|---|
| `/atg/portal/gear/GearConfigFormHandler` | Class `atg.portal.framework.GearConfigFormHandler`.<br><br>`GearConfigFormHandler` is provided with the PAF. It is used in the Order Approval gear configuration pages to create the forms with which to set the gear's instance and user parameters.<br><br>For more information on `GearConfigFormHandler`, see the *Gears and the Portal Application Framework* and *Creating Gear Page Fragments* chapters in the *ATG Portal Development Guide*. |
| `/atg/userdirectory/droplet/HasFunction` | Class `atg.userdirectory.droplet.HasFunction`.<br><br>The `HasFunction` servlet bean is used in `OrderApprovalShared.jsp` and `OrderApprovalFull.jsp` first to check whether the user who has logged in is an approver and then to render the page content accordingly. If the user is an approver, the gear content is displayed. If the user isn't an approver, a message indicating that the user isn't authorized to view the gear is displayed. |
| `/atg/commerce/approval/ApprovalRequiredDroplet` | Class `atg.b2bcommerce.approval.ApprovalRequiredDroplet`.<br><br>The `ApprovalRequiredDroplet` servlet bean is used to retrieve and display the orders that require approval by the current user. It is used in both the gear's Shared and Full Page content pages.<br><br>By default, the `ApprovalRequiredDroplet.sortAscending` property is set to true; this displays the most recent orders that require the approver's attention first. To display the oldest orders first, simply set this property to false.<br><br>For more information about `ApprovalRequiredDroplet`, see the Implementing an Order Approval Process chapter. |

| | |
|---|---|
| /atg/commerce/approval/ApprovalFormHandler | Class `atg.b2bcommerce.approval.ApprovalFormHandler`.<br><br>`ApprovalFormHandler` is used in the gear's Full Page content pages to create and manage the forms with which the approver can approve and reject orders.<br><br>For more information about `ApprovalFormHandler`, see the Implementing an Order Approval Process chapter. |
| /atg/commerce/gears/orderapproval/ApprovalResolvedDroplet | Class `atg.b2bcommerce.approval.ApprovalRequiredDroplet`.<br><br>The `ApprovalResolvedDroplet` servlet bean is used to retrieve and display the orders that have been approved and/or rejected by the current user. It is used in the Full Page content page that displays the approver's "resolved approval requests."<br><br>For more information about the `ApprovalRequiredDroplet` class from which this Order Approval gear component is instantiated, see the Implementing an Order Approval Process chapter. |
| /atg/commerce/order/OrderLookup | Class `atg.b2bcommerce.order.B2BOrderLookup`.<br><br>The `OrderLookup` servlet bean is used in the gear's Full Page content pages to retrieve and display a given order.<br><br>For more information on `OrderLookup`, see the Implementing Order Retrieval chapter. |
| /atg/commerce/catalog/ProductLookup | Class `atg.commerce.catalog.custom.CatalogItemLookupDroplet`.<br><br>The `ProductLookup` servlet bean is used to retrieve and display product information for the approver's orders. It is used in the *Order List* gear content page.<br><br>For more information on `ProductLookup`, see the CatalogItemLookupDroplet reference entry in *Appendix: ATG Commerce Servlet Beans*. |

### *Tag Libraries*

The Order Approval gear uses the following standard tag libraries:

- Core tag library
- DSP tag library
- PAF tag library
- Jakarta's i18n tag library

For information on the DSP tag libraries, see the *ATG Page Developer's Guide*. For information on the PAF tag library and Jakarta's i18n tag library, see the *ATG Portal Development Guide*.

No custom tag libraries were written for this gear.

# Appendix: ATG Commerce Servlet Beans

This appendix provides reference entries for the ATG Servlet Beans that are included with ATG Business Commerce and/or ATG Consumer Commerce. The servlet beans are grouped below by functional area so you can easily identify related ones. Detailed reference entries follow. Note that, because sometimes multiple servlet beans are instantiated from the same class, the reference entries are organized and alphabetized according to the class from which they are instantiated.

For general information about servlet beans and how to use them in JSPs, refer to the *Using ATG Servlet Beans* chapter in the *ATG Page Developer's Guide*. For information about how to create your own custom servlet beans, refer to the *Creating and Using ATG Servlet Beans* chapter in the *ATG Programming Guide*.

### *Abandoned Order Services Servlet Beans*

Class `atg.commerce.order.abandoned.ConvertAbandonedOrderDroplet`

- `/atg/commerce/order/abandoned/ConvertAbandonedOrderDroplet`

Class `atg.commerce.order.abandoned.ReanimateAbandonedOrderDroplet`

- `/atg/commerce/order/abandoned/ReanimateAbandonedOrderDroplet`

Class `atg.commerce.order.abandoned.SetLastUpdatedDroplet`

- `/atg/commerce/order/abandoned/SetLastUpdatedDroplet`

### *Approval Process Servlet Beans*

Class `atg.b2bcommerce.approval.ApprovalRequiredDroplet`

- `/atg/commerce/approval/ApprovalRequiredDroplet`

Class `atg.b2bcommerce.approval.ApprovedDroplet`

- `/atg/commerce/approval/ApprovedDroplet`

### *Catalog Servlet Beans*

Class `atg.commerce.catalog.DisplaySkuProperties`

- `/atg/commerce/catalog/DisplaySkuProperties`

Class `atg.commerce.catalog.comparison.ProductListContains`

- `/atg/commerce/catalog/comparison/ProductListContains`

Class `atg.commerce.catalog.custom.CatalogItemLookupDroplet`

- `/atg/commerce/catalog/CategoryLookup`
- `/atg/commerce/catalog/ProductLookup`
- `/atg/commerce/catalog/SKULookup`

Class `atg.commerce.catalog.custom.CatalogPossibleValues`

- `/atg/commerce/catalog/RepositoryValues`

Class `atg.commerce.catalog.custom.ForEachItemInCatalog`

- `/atg/commerce/catalog/ForEachItemInCatalog`

Class `atg.repository.servlet.ItemLookupDroplet`

- `/atg/commerce/catalog/CategoryLookup`
- `/atg/commerce/catalog/MediaLookup`
- `/atg/commerce/catalog/ProductLookup`
- `/atg/commerce/catalog/SKULookup`

Class `atg.repository.servlet.NavHistoryCollector`

- `/atg/commerce/catalog/CatalogNavHistoryCollector`

Class `atg.repository.servlet.PossibleValues`

- `/atg/commerce/catalog/RepositoryValues`

Class `atg.userprofiling.ViewItemEventSender`

- `/atg/commerce/catalog/CategoryBrowsed`
- `/atg/commerce/catalog/ProductBrowsed`

### *Claimable Servlet Beans*

Class: `atg.commerce.claimable.AvailableStoreCredits`

- `/atg/commerce/claimable/AvailableStoreCredits`

Class: `atg.commerce.claimable.GiftCertificateAmountAvailable`

- `/atg/commerce/claimable/GiftCertificateAmountAvailable`

### *Collection Filtering Servlet Beans*

Class `atg.service.collections.filter.droplet.CollectionFilter`

- `/atg/collections/filter/droplet/InventoryFilterDroplet`
- `/atg/collections/filter/droplet/ProductFilterDroplet`

- /atg/collections/filter/droplet/StartEndDateFilterDroplet
- /atg/commerce/collections/filter/droplet/ExcludeItemsInCartFilterDroplet

### *Fulfillment Servlet Beans*

Class atg.commerce.fulfillment.ShippableGroupsDroplet

- /atg/commerce/fulfillment/droplet/ShippableGroupsDroplet

Class atg.commerce.fulfillment.ShippingDroplet

- /atg/commerce/fulfillment/droplet/ShippingDroplet

### *Gear Servlet Beans*

Class atg.b2bcommerce.approval.ApprovalRequiredDroplet

- /atg/commerce/gears/orderapproval/ApprovalResolvedDroplet

### *Gift List Servlet Beans*

Class atg.commerce.gifts.GiftitemDroplet

- /atg/commerce/gifts/BuyItemFromGiftlist
- /atg/commerce/gifts/RemoveItemFromGiftlist

Class atg.commerce.gifts.GiftlistDroplet

- /atg/commerce/gifts/GiftlistDroplet

Class atg.commerce.gifts.GiftShippingGroupDroplet

- /atg/commerce/gifts/IsGiftShippingGroup

Class atg.commerce.gifts.GiftShippingGroupsDroplet

- /atg/commerce/gifts/GiftShippingGroups

Class atg.repository.servlet.ItemLookupDroplet

- /atg/commerce/gifts/GiftitemLookupDroplet
- /atg/commerce/gifts/GiftlistLookupDroplet

### *Inventory Servlet Beans*

Class atg.commerce.inventory.InventoryDroplet

- /atg/commerce/inventory/InventoryLookup

### *Order Management Servlet Beans*

Class atg.b2bcommerce.order.B2BOrderLookup

- /atg/commerce/order/OrderLookup

Class `atg.commerce.order.AddItemToCartServlet`

- `/atg/commerce/order/AddItemToCartServlet`

Class `atg.commerce.order.IsHardGoodsDroplet`

- `/atg/commerce/order/IsHardGoods`

Class `atg.commerce.order.OrderLookup`

- `/atg/commerce/order/AdminOrderLookup`
- `/atg/commerce/order/OrderLookup`

### *Purchase Process Servlet Beans*

Class `atg.b2bcommerce.order.purchase.CostCenterDroplet`

- `/atg/commerce/order/purchase/CostCenterDroplet`

Class `atg.commerce.order.purchase.PaymentGroupDroplet`

- `/atg/commerce/order/purchase/PaymentGroupDroplet`

Class `atg.commerce.order.purchase.RepriceOrder`

- `/atg/commerce/order/purchase/RepriceOrderDroplet`

Class `atg.commerce.order.purchase.ShippingGroupDroplet`

- `/atg/commerce/order/purchase/ShippingGroupDroplet`

### *Pricing Servlet Beans*

Class `atg.commerce.pricing.AvailableShippingMethodsDroplet`

- `/atg/commerce/pricing/AvailableShippingMethods`

Class: `atg.commerce.pricing.GetApplicablePromotions`

- `/atg/commerce/pricing/GetApplicablePromotions`

Class `atg.commerce.pricing.ItemPricingDroplet`

- None (abstract class)

Class `atg.commerce.pricing.PriceEachItemDroplet`

- `/atg/commerce/pricing/PriceEachItem`

Class `atg.commerce.pricing.PriceItemDroplet`

- `/atg/commerce/pricing/PriceItem`

Class `atg.commerce.pricing.PriceRangeDroplet`

- `/atg/commerce/pricing/PriceRangeDroplet`

Class `atg.commerce.pricing.ShipItemRelPrice`

- `/atg/commerce/pricing/ShipItemRelPrice`

Class `atg.commerce.pricing.pricelists.ComplexPriceDroplet`

- `/atg/commerce/pricing/priceLists/ComplexPriceDroplet`

Class `atg.commerce.pricing.pricelists.PriceDroplet`

- `/atg/commerce/pricing/priceLists/PriceDroplet`

Class: `atg.commerce.pricing.CurrencyCodeDroplet`

- `/atg/commerce/pricing/CurrencyCodeDroplet`

Class: `atg.commerce.pricing.UnitPriceDetailDroplet`

- `/atg/commerce/pricing/UnitPriceDetailDroplet`

### *Promotion Servlet Beans*

Class `atg.commerce.promotion.ClosenessQualifierDroplet`

- `/atg/commerce/promotion/ClosenessQualifierDroplet`

Class `atg.commerce.promotion.CouponDroplet`

- `/atg/commerce/promotion/CouponDroplet`

Class `atg.commerce.promotion.PromotionDroplet`

- `/atg/commerce/promotion/PromotionDroplet`

### *Shopping Process Tracking Servlet Beans*

Class `atg.markers.bp.droplet.AddBusinessProcessStage`

- `/atg/commerce/bp/droplet/AddShoppingProcessStageDroplet`

Class `atg.markers.bp.droplet.HasBusinessProcessStage`

- `/atg/commerce/bp/droplet/HasShoppingProcessStageDroplet`

Class `atg.markers.bp.droplet.MostRecentBusinessProcessStage`

- `/atg/commerce/bp/droplet/MostRecentShoppingProcessStageDroplet`

Class `atg.markers.bp.droplet.RemoveBusinessProcessStage`

- `/atg/commerce/bp/droplet/RemoveShoppingProcessStageDroplet`

### *Multisite Servlet Beans*

Class `atg.multisite.droplet.SiteIdForItem`

- `/atg/commerce/mulsite/SiteIdForCatalogItem`

## AddItemToCartServlet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.AddItemToCartServlet` |
| **Component(s)** | `/atg/commerce/order/AddItemToCartServlet` |

The `AddItemToCartServlet` servlet bean adds an item to a shopping cart via a URL. This enables product information, such as the SKU and quantity of an item, to be passed as part of a URL to ATG Commerce from a third-party application.

`AddItemToCartServlet` is called by a servlet in the request-handling pipeline, `CommerceCommandServlet` (class `atg.commerce.order.CommerceCommandServlet`). When `CommerceCommandServlet` receives a `dcs_action` input parameter of `addItemToCart`, it calls `AddItemToCartServlet`.

If only the required parameters for `AddItemToCartServlet` are supplied, then the servlet bean creates and adds to the cart a commerce item of class `atg.commerce.order.CommerceItemImpl`. If additional parameters are supplied, then `AddItemToCartServlet` functions differently depending on which optional parameters are supplied.

Refer to *Adding an Item to an Order Via a URL* in the *Working With Purchase Process Objects* chapter in the *ATG Commerce Programming Guide* for more information on the implementation and behavior of both `CommerceCommandServlet` and `AddItemToCartServlet`.

### *Input Parameters*

**url_catalog_ref_id** (Required)
The SKU of the item to add to the cart.

**url_product_id** (Required)
The product ID of the item to add to the cart.

**url_quantity** (Required)
The quantity of the item to add to the cart.

**url_shipping_group_id**
The ID of the shipping group to which to add the item.

**url_item_type**
The commerce item type to use to create the commerce item.

**url_commerce_item_id**
The ID of a commerce item that is removed from the order. Use this parameter when you want to replace one item in the cart with another.

**dcs_ci_***
An identifier for setting a `CommerceItem` property.

If a parameter with this prefix is supplied, then the `CommerceItem` property to which it refers is set to the given value in the commerce item. For example, if the `dcs_ci_catalogKey` parameter is supplied with a given value of `en_US`, then the `catalogKey` property is set to `en_US`.

**dcs_conf_<n>**
An identifier for setting a configurable property of a `ConfigurableCommerceItem`.

Memory and hard drives are two examples of configurable properties of a computer item. That is, a customer can order various amounts of memory and one or more hard drives for a single computer. Both the memory and the hard drive are represented as subSKUs of the base SKU, the computer item.

If a parameter with this prefix is supplied, then the configurable property to which it refers is set to the specified value. This parameter's format is as follows:

```
dcs_conf_<n>=<sku id, product id, individual quantity>
```

<n> is an integer representing the configurable property in the list of configurable properties for the given `ConfigurableCommerceItem`. "sku id", "product id", and "individual quantity" provide the data with which to construct a `SubSkuCommerceItem` to represent the configurable property for the given `ConfigurableCommerceItem`. "sku id" is the SKU id of the given subSKU. "product id" is the product id of the given subSKU. "individual quantity" is the quantity of the given subSKU to add to a *single* `ConfigurableCommerceItem`. For example, if you are adding 2 hard drives to each of 1,000 computers, then the individual quantity of the hard drive subSKU is 2.

**dcs_subsku**
An identifier for a subSKU. Specify this parameter when you are adding a configurable SKU and its subSKUs to a cart. Values for this parameter must use this format:

```
SKU_ID,product_ID, quantity,
```

To specify multiple subSKUs, create a string that includes information about each subSKU separated by commas. For example:

```
79054,12159,1,79303,11900,4,90931,20133,2
```

### *Output Parameters*

None. `AddItemToCartServlet` is called by a servlet in the request-handling pipeline, `CommerceCommandServlet`.

### *Open Parameters*

None. `AddItemToCartServlet` is called by a servlet in the request-handling pipeline, `CommerceCommandServlet`.

### *Example*

No JSP example is provided. `AddItemToCartServlet` is called by a servlet in the request-handling pipeline, `CommerceCommandServlet`.

## AddBusinessProcessStage

| | |
|---|---|
| **Class Name** | `atg.markers.bp.droplet.AddBusinessProcessStage` |
| **Component** | `/atg/commerce/bp/droplet/AddShoppingProcessStageDroplet` |

This servlet bean marks orders when they reach a new stage in the shopping process. It is an instance of `AddBusinessProcessStage` with the `businessProcessName` property set to `ShoppingProcess`.

### *Input Parameters*

**businessProcessName**
The name of the business process. If not specified, then we use the value of the servlet bean's `defaultBusinessProcessName` property, which is `ShoppingProcess` by default.

**businessProcessStage** (Required)
The stage within the business process.

### *Output Parameters*

**errorMsg**
The error message describing a failure.

### *Open Parameters*

**output**
Rendered on successful completion

**error**
Rendered on error.

### *Example*

```
<dsp:droplet name="AddShoppingProcessStageDroplet">
  <dsp:param name="businessProcessStage" value="ShippingPriceViewed"/>
</dsp:droplet>
```

## ApprovalRequiredDroplet

| | |
|---|---|
| **Class Name** | `atg.b2bcommerce.approval.ApprovalRequiredDroplet` |
| **Component(s)** | `/atg/commerce/approval/ApprovalRequiredDroplet` (ATG Business Commerce only)<br>`/atg/commerce/gears/orderapproval/ApprovalResolvedDroplet` (Order Approval portal gear only) |

The `ApprovalRequiredDroplet` servlet bean supports the order approval process by retrieving all orders requiring approval by a given approver. It queries the order repository and returns all orders that meet the following 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`.
- The order's site either matches the `siteID` or falls within the specified `siteScope`.

Note that the `ApprovalRequiredDroplet` servlet bean has a security feature that allows the current user, the approver, to view only the orders of customers for whom he or she is allowed to approve orders. This feature is enabled by default. To disable the feature, set the `enableSecurity` property to false.

### *Input Parameters*

**approverid** (Required)
The ID of the current user profile; the approver.

**numOrders**
The number of orders to return from the query. This parameter is optional and typically is used to break large result sets into manageable pieces.

**siteIds**

A collection of site IDs used to limit the query to orders associated with the specified sites. If `siteIds` is specified, `siteScope` is ignored.

**siteScope**

If you are using ATG's multisite feature, you can filter orders by site. Use `siteScope` as an alternative to `siteIds`, and provide one of the following scopes:

- `null` or `all`: Finds orders for all sites
- `current`: Finds orders for the current site, as determined by the site context
- `shareableTypeId`: pass in the ID of a shareable type, such as `atg.ShoppingCart`, to find orders for all sites in the sharing group for that type.

The default `siteScope` can be set through a configurable property on the component. The default value is `all`.

**startIndex**
The index of the first order to return. If `startIndex` is null, then it defaults to 0. This parameter is optional and typically is used to break large result sets into manageable pieces.

### *Output Parameters*

**result**
The array of `Order` objects.

**count**
The number of `Order` objects in the `result` output parameter.

**totalCount**
The total number of `Order` objects that satisfied the criteria.

**nextIndex**
The index of the first order in the next set of results. If `startIndex` or `numOrders` was null, then this parameter will also be null.

**previousIndex**
The index of the first order in the previous set of results. If `startIndex` or `numOrders` was null, then this parameter will also be null.

`nextIndex` and `previousIndex` allow the user to cycle back and forth between result sets.

**startRange**
The 1-based index of the first `Order` in the set of results.

**endRange**
The 1-based index of the last `Order` in the set of results.

**errorMessage**
The error message to display to the user if an error occurs.

### *Open Parameters*

**output**
This open parameter renders the array of order objects set in the result output parameter.

**empty**
This parameter is rendered if there are no orders that require approval by the current user.

**error**
This parameter is rendered if an error occurs.

### *Example*

The following example retrieves from the repository the orders requiring approval by the current user, an approver, and lists each order's repository id on the page.

```
<dsp:droplet name="ApprovalRequiredDroplet">
 <dsp:param bean="/atg/userprofiling/Profile.repositoryId"
name="approverid"/>
 <dsp:param value="0" name="startIndex"/>
 <dsp:param value="10" name="numOrders"/>
 <dsp:oparam name="output">
<dsp:droplet name="ForEach">
   <dsp:param param="result" name="array"/>
   <dsp:setvalue param="order" paramvalue="element"/>
       <dsp:oparam name="output">
          <dsp:valueof param="order.repositoryId"/><br>
       </dsp:oparam>
       <dsp:oparam name="error">
          <dsp:valueof param="errorMsg"/><br>
       </dsp:oparam>
    </dsp:droplet>
 </dsp:oparam>
</dsp:droplet>
```

## ApprovedDroplet

| | |
|---|---|
| **Class Name** | `atg.b2bcommerce.approval.ApprovedDroplet` |
| **Component(s)** | `/atg/commerce/approval/ApprovedDroplet` (ATG Business Commerce only) |

The `ApprovedDroplet` servlet bean supports the order approval process by retrieving all orders that have been approved and/or rejected by a given approver. It queries the order repository and returns all orders that have the approver's profile ID in the `approverIds` property.

Note that the `ApprovedDroplet` servlet bean has a security feature that allows the current user, the approver, to view only the orders of customers for whom he or she is allowed to approve orders. This feature is enabled by default. To disable the feature, set the `enableSecurity` property to false.

### *Input Parameters*

**approverid** (Required)
The ID of the current user profile; the approver.

**numOrders**
The number of orders to return on the query. This parameter is optional and typically is used to break large result sets into manageable pieces.

**startIndex**
The index of the first order to return. If `startIndex` is null, then it default to `0`. This parameter is optional and typically is used to break large result sets into manageable pieces.

### *Output Parameters*

**result**
The array of `Order` objects.

**count**
The number of `Order` objects in the `result` output parameter.

**totalCount**
The total number of `Order` objects that satisfied the criteria.

**nextIndex**
The index of the first order in the next set of results. If `startIndex` or `numOrders` was null, then this parameter will also be null.

**previousIndex**
The index of the first order in the previous set of results. If `startIndex` or `numOrders` was null, then this parameter will also be null.

`nextIndex` and `previousIndex` allow the user to cycle back and forth between result sets.

**startRange**
The 1-based index of the first `Order` in the set of results.

**endRange**
The 1-based index of the last `Order` in the set of results.

**errorMessage**
The error message to display to the user if an error occurs.

### *Open Parameters*

**output**
This open parameter renders the array of `Order` objects set in the `result` output parameter.

**empty**
The open parameter rendered if there are no orders that have been approved and/or rejected by the current user.

**error**
The open parameter rendered if an error occurs.

### Example

The following example retrieves from the repository the orders that have been approved and/or rejected by the current user, an approver, and lists each order's repository ID on the page.

```
<dsp:droplet name="ApprovedDroplet">
 <dsp:param bean="/atg/userprofiling/Profile.repositoryId" name="approverid"/>
 <dsp:param value="0" name="startIndex"/>
 <dsp:param value="10" name="numOrders"/>
 <dsp:oparam name="output">
  <dsp:droplet name="ForEach">
    <dsp:param param="result" name="array"/>
    <dsp:param value="order" name="elementName"/>
    <dsp:oparam name="output">
     <dsp:valueof param="order.repositoryId"/><br>
    </dsp:oparam>
    <dsp:oparam name="error">
     <dsp:valueof param="errorMsg"/><br>
    </dsp:oparam>
  </dsp:droplet>
 </dsp:oparam>
</dsp:droplet>
```

## AvailableShippingMethodsDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.AvailableShippingMethodsDroplet` |
| **Component(s)** | `/atg/commerce/pricing/AvailableShippingMethods` |

The `AvailableShippingMethods` servlet bean displays available shipping methods for a particular shipping group. The class's service method calls into `ShippingPricingEngine`'s `getAvailableMethods` method to return a list of Strings representing the shipping methods. These shipping methods correspond to the `shippingMethod` property of the `HardgoodShippingGroup` order class.

### Input Parameters

**shippingGroup** (Required)
The `ShippingGroup` to be shipped.

**pricingModels**
A collection of shipping pricing models. If this parameter is null, then the session-scoped `PricingModelHolder` is resolved and the collection is retrieved. The path to the `PricingModelHolder`

component is configured through the `userPricingModelsPath` property of the `AvailableShippingMethodsDroplet`.

**profile**
The `RepositoryItem` that represents the customer requesting the shipping methods. If this parameter is null, then the session-scoped `Profile` is resolved. The path to the `Profile` component is configured through the `profilePath` property of the `AvailableShippingMethodsDroplet`.

**locale**
The locale of the customer requesting the shipping methods. This parameter may be either a `java.util.Locale` object or a String that represents a locale. If this parameter is not found, then by default the locale is retrieved from the request. If this locale cannot be determined, then the default locale for this component is used.

### *Output Parameters*

**availableShippingMethods**
A list of Strings representing the shipping methods, which can be used for setting a `shippingMethod` value in a `HardgoodShippingGroup`.

### *Open Parameters*

**output**
An `oparam` that includes the `availableShippingMethods` parameter.

### *Example*

The following example uses the `AvailableShippingMethods` servlet bean to provide a select box of available shipping methods that are bound to the `shippingMethod` property of the first shipping group.

```
<dsp:droplet name="/atg/commerce/pricing/AvailableShippingMethods">
<dsp:param bean="ShoppingCartModifier.shippingGroup" name="shippingGroup"/>
<dsp:oparam name="output">
 <dsp:select bean="ShoppingCartModifier.shippingGroup.shippingMethod">
 <dsp:droplet name="ForEach">
 <dsp:param param="availableShippingMethods" name="array"/>
 <dsp:param value="method" name="elementName"/>
 <dsp:oparam name="output">
   <dsp:getvalueof id="option16" param="method" idtype="java.lang.String">
<dsp:option value="<%=option16%>"/>
</dsp:getvalueof><dsp:valueof param="method"/>
 </dsp:oparam>
 </dsp:droplet>
 </dsp:select>
</dsp:oparam>
</dsp:droplet>
```

## AvailableStoreCredits

| | |
|---|---|
| **Class Name** | `atg.commerce.claimable.AvailableStoreCredits` |
| **Component(s)** | `/atg/commerce/claimable/AvailableStoreCredits` |

The `AvailableStoreCredits` servlet bean gives you access to store credits associated with a given user's profile. You can then use this information in your page code.

### *Input Parameters*

**profile** (Required)
The `RepositoryItem` that represents the customer for whom you want to display store credit information.

### *Output Parameters*

**storeCredits**
A list of store credits associated with the provided user profile.

### *Open Parameters*

**output**
The open parameter is always rendered.

**empty**
The open parameter rendered if there are no store credits for the current user.

### *Example*

The following example uses the `AvailableStoreCredits` servlet bean to provide information on a user's store credits.

```
<dsp:droplet name="AvailableStoreCredits">
   <dsp:param name="profile" bean="Profile"/>
   <dsp:oparam name="output">

     <dsp:getvalueof var="onlineCredits" vartype="java.lang.Object"
param="storeCredits"/>
     <c:if test="${not empty onlineCredits}">
       <div id="atg_store_onlineCredits">
         <h3><fmt:message key="myaccount_onlineCredits.savedOnlineCredits"/></h3>
         <c:forEach var="onlineCredit" items="${onlineCredits}"
varStatus="onlineCreditStatus">
           <dsp:setvalue param="storeCredit" value="${onlineCredit}"/>
           <dsp:getvalueof var="storeCredit" param="storeCredit"/>
           <c:if test="${not empty storeCredit}">
```

```
             <div class="atg_store_onlineCreditsDetails">
               <dsp:getvalueof var="count" vartype="java.lang.Double"
value="${onlineCreditStatus.count}"/>
               <h4>
                 <fmt:message key="common.credit"/><fmt:message
key="common.numberSymbol"/>
                 <fmt:formatNumber value="${count}" type="number"/>
               </h4>
               <div>
                 <fmt:message
key="myaccount_onlineCredits.remainingCredit"/><fmt:message
key="common.labelSeparator"/>
               </div>
               <dsp:getvalueof var="amountRemaining" vartype="java.lang.Double"
param="storeCredit.amountAvailable"/>
               <dsp:getvalueof var="currencyCode" vartype="java.lang.String"
param="currencyCode"/>
               <div class="atg_store_onlineCreditTotal">
                 <fmt:formatNumber value="${amountRemaining}" type="currency"
currencyCode="${currencyCode}" />
               </div>
             </div>
           </c:if>
         </c:forEach>
       </div>
     </c:if>
   </dsp:oparam>
 </dsp:droplet>
```

## B2BOrderLookup

| | |
|---|---|
| **Class Name** | `atg.b2bcommerce.order.B2BOrderLookup` |
| **Component(s)** | `/atg/commerce/order/OrderLookup`<br>(ATG Business Commerce only) |

The `OrderLookup` servlet bean retrieves one or more `Order` objects, depending on the supplied input parameters. It enables you to retrieve a single order, all orders assigned to a particular cost center, all orders placed by a particular user, or all orders placed by a particular user that are in a specific state.

`OrderLookup` has a security feature that allows the current user to view only her own orders. By default, this feature is enabled. To disable the feature, set the `enableSecurity` property to false.

### *Input Parameters*

**orderId** (Either this, `userId`, or `costCenterId` is required.)
The ID of the order to retrieve.

**userId** (Either this, `orderId`, or `costCenterId` is required.)
The ID of the user profile whose orders will be retrieved.

**costCenterId** (Either this, `orderId`, or `userId` is required.)
The ID of the cost center whose orders will be retrieved.

**state**
The desired state of the orders to retrieve.

This parameter can be used in conjunction with `userId`. You can specify one of the following:

- any one of the states defined in `atg.b2bcommerce.states.B2BOrderStates`
- open
- closed

If you specify “open,” then all orders whose states are specified in the `openStates` property of the `OrderLookup` component are returned; by default, this list of states is set to the following:

```
submitted
processing
pending_merchant_action
pending_customer_action
```

If you specify “closed,” then all orders whose states are specified in the `closedStates` property of the `OrderLookup` component are returned; by default, this list of states is set to the following:

```
no_pending_action
```

You can override either list of states by using the optional `openStates` or `closedStates` input parameter (see below).

**openStates**
A comma-separated list of states that correspond to “open” state.

This parameter can be used in conjunction with the `state` input parameter when the `state` input parameter is set to “open.” Use this optional parameter when you want to override the configured list of states in the `openStates` property of the `OrderLookup` component.

**closedStates**
A comma-separated list of states that correspond to “closed” state.

This parameter can be used in conjunction with the `state` input parameter when the `state` input parameter is set to “closed.” Use this optional parameter when you want to override the configured list of states in the `closedStates` property of the `OrderLookup` component.

**sortBy**
A string that specifies an `Order` property by which to sort the orders.

This parameter can be used in conjunction with `userId`. When using this parameter, you can specify the name of any Order Repository property (that is, the name of any property defined in `orderrepository.xml`), such as `id`, `state`, or `submittedDate`.

**sortAscending**
True or false. This parameter is used in conjunction with the `sortBy` input parameter. If set to true, the `Order` objects in the resulting array are sorted in ascending order by the property specified in the `sortBy` input parameter. The default value is false.

**numOrders**
The number of orders to return for the given query.

**startIndex**
The index of the first order in the result set. This parameter is useful for cycling through a large number of orders.

**queryTotal**
Indicates whether the number of retrieved orders will be calculated into a total that's accessible through the `totalCount` and `total_count` output parameters. Setting this property to `false` prevents the total count from being generated, regardless of the value specified in the `queryTotal` property. Omitting this parameter causes the default value, `true`, to be used. Use this parameter to ensure that queries to the database are made only when necessary.

**queryTotalOnly**
Indicates whether the total number of orders and the orders themselves are produced from the servlet bean. Setting this parameter to `true` makes the total number of retrieved orders available through the `totalCount` and `total_count` output parameters. The orders themselves are not retrieved or accessible. Use this parameter to ensure that queries to the database are made only when necessary.

Omitting this parameter, which is the same as setting it to `false`, saves a list of order objects to the `output` open parameter as well as the total number of orders to the `totalCount` and `total_count` output parameters.

If `queryTotal=false` (orders, no total) and `queryTotalOnly=true` (total, no orders), a total is generated only as specified in the `queryTotalOnly` parameter.

### *Output Parameters*

**result**
The array of `Order` objects. If the `orderId` input parameter was used, then this parameter contains a single `Order` object.

**errorMsg**
If an error occurred, this is the detailed error message for the user.

**count**
The size of the array of `Order` objects.

**totalCount**
If the queryTotal property is set to true, this parameter indicates the total number of orders that meet the criteria for the order lookup.

**total_count**
Identical to the totalCount output parameter (above).

**startRange**
The index number that marks the beginning of a range of orders. For example, if 5 orders were returned from a given OrderLookup query, the startRange is set to 1.

**endRange**
The index number that marks the end of a range of orders. For example, if 5 orders were returned from a given OrderLookup query with a startRange of 6, the endRange is set to 10.

**nextIndex**
The index of the first order in the next set of results. If the startIndex or numOrders input parameter was null, then this parameter will be null.

**previousIndex**
The index of the first order in the previous set of results. If the startIndex or numOrders input parameter was null, then this parameter will be null.

### *Open Parameters*

**output**
The open parameter rendered if the orders are successfully retrieved.

**empty**
The open parameter rendered if there are no orders to return.

**error**
The open parameter rendered if an error occurs.

### *Example*

The following example describes how to use the OrderLookup servlet bean to retrieve all open orders for the current user and to display their IDs.

```
<dsp:droplet name="/atg/commerce/order/OrderLookup">
 <dsp:param bean="/atg/userprofiling/Profile.repositoryId" name="userId"/>
 <dsp:param value="open" name="state"/>
 <dsp:oparam name="output">
    <dsp:droplet name="/atg/dynamo/droplet/ForEach">
       <dsp:param param="result" name="array"/>
       <dsp:oparam name="outputStart">
          <OL>
       </dsp:oparam>
       <dsp:oparam name="output">
```

```
          <LI> <dsp:valueof param="element.id">no order number</dsp:valueof>
       </dsp:oparam>
       <dsp:oparam name="outputEnd">
          </OL>
       </dsp:oparam>
       <dsp:oparam name="empty">
          No open orders.
       </dsp:oparam>
    </dsp:droplet>
 </dsp:oparam>
 <dsp:oparam name="error">
    <span class=profilebig>ERROR:
       <dsp:valueof param="errorMsg">no error message</dsp:valueof>
    </span>
 </dsp:oparam>
</dsp:droplet>
```

## CatalogItemLookupDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.catalog.custom.CatalogItemLookupDroplet` |
| **Component(s)** | `/atg/commerce/commerce/catalog/CategoryLookup`<br>`/atg/commerce/catalog/ProductLookup`<br>`/atg/commerce/catalog/SKULookup` |

Servlet beans instantiated from `CatalogItemLookupDroplet` use the ID of a `RepositoryItem` to look up the item in a repository. If the item isn't found, the `empty` open parameter is rendered. If the item is found, the servlet bean then checks if the item belongs to the user's catalog in his current Profile. Or, if a catalog is specified via the `catalog` input parameter, the servlet bean instead checks if the item belongs to the specified catalog. In either case, if the item is found, then the servlet bean renders the `output` open parameter. If the item isn't found, then the servlet bean renders the `wrongCatalog` open parameter.

Through properties, you can configure the repository and the item descriptor type to use when looking up a given item, and you can define a mapping from a key to an alternate set of repositories. For example, you might have a `ProductLookup` servlet bean with the following properties:

```
$class=atg.commerce.catalog.custom.CatalogItemLookupDroplet
repository=/atg/commerce/catalog/ProductCatalog
itemDescriptor=product
alternateRepositories=\
      fr_FR=/atg/commerce/catalog/FrenchProductCatalog
```

```
ja_JP=/atg/commerce/catalog/JapaneseProductCatalog
de_DE=/atg/commerce/catalog/GermanProductCatalog
```

If the `useParams` property is true, then the repository and item descriptor can be resolved through input parameters.

### *Input Parameters*

**id** (Required)
The id of the item to lookup.

**catalog**
The catalog in which to look for the item. The catalog must be a `RepositoryItem`.

Note that this parameter usually isn't needed. If unset, the servlet bean checks for the item in the catalog that is in the user's profile. However, if you want to check for the item in a catalog other than the user's current catalog, or if there is no current session (for example, within the context of an e-mail template), then you can explicitly provide the catalog via this parameter.

**elementName**
If specified, this name will be used for the parameter set within the `output` open parameter.

**itemDescriptor**
The name of the item descriptor to use to load the item.
(Note: Use of this parameter is not recommended. A better approach is to specify the item descriptor through the `itemDescriptor` property of the servlet bean.)

**repository**
The repository in which to look for the item.
(Note: Use of this parameter is not recommended. A better approach is to define different instances of `CatalogItemLookupDroplet` for each set of repositories you want to use.)

**repositoryKey**
If specified, this parameter will be used as a key to map to a secondary set of repositories.

### *Output Parameters*

**element**
The `RepositoryItem` that corresponds to the supplied id. (This parameter name can be changed by setting the `elementName` input parameter.)

**error**
The open parameter rendered if an error occurs while looking up the item.

### *Open Parameters*

**output**
The open parameter rendered if the item is found in the repository and belongs to the current user's catalog (or instead, if specified, to the catalog passed in via the `catalog` input parameter).

**wrongCatalog**
The open parameter rendered if the item is found in the repository but doesn't belong to the current user's catalog (or instead, if specified, to the catalog passed in via the `catalog` input parameter). This makes the item accessible if you need it.

**empty**
The open parameter rendered in all other cases, such as when the item is not found in the repository or the user did not specify a required parameter.

**noCatalog**
The open parameter is rendered if the droplet cannot determine which catalog is associated with the user. This would be the case if you called the droplet without an explicit catalog parameter *and* there was no catalog in the user profile.

### *Example*

```
<dsp:droplet name="ProductLookup">
  <dsp:param param="productId" name="id"/>
  <dsp:param bean="/OriginatingRequest.requestLocale.localeString"
             name="repositoryKey"/>
  <dsp:param value="product" name="elementName"/>
  <dsp:oparam name="output">
   <dsp:getvalueof id="a10" param="product.template.url"
               idtype="java.lang.String">
<dsp:a href="<%=a10%>">
     <dsp:param param="product.repositoryId" name="prod_id"/>
     <dsp:valueof param="product.displayName"/>
   </dsp:a></dsp:getvalueof>
  </dsp:oparam>
</dsp:droplet>
```

If the mapping for the provided key is found, then that repository is used. Otherwise, the system falls back to the default repository and searches for the item.

If the item cannot be found in the repository searched (e.g. the French product catalog), then the servlet bean once again falls back to the default repository and attempts to find the item using the same ID. This is only useful if the items have the same ID in each repository. For example, suppose you are viewing a site in French, and you attempt to look at product 1234. If the ID is defined in the French product catalog, you will see the French version of that product. However, if 1234 is not defined in the French product catalog, then you will see the default English version of product 1234.

This behavior can be modified with the use of the `useDefaultRepository` and `getDefaultItem` properties. If `useDefaultRepository` is false and an alternate repository cannot be found, then no repository is searched and the `empty` open parameter is rendered. Similarly, if an alternative repository is selected, but the item cannot be found, and if `getDefaultItem` is false, then the `empty` open parameter is rendered.

## CatalogPossibleValues

| | |
|---|---|
| **Class Name** | `atg.commerce.catalog.custom.CatalogPossibleValues` |
| **Component(s)** | `/atg/commerce/catalog/RepositoryValues` |

The `CatalogPossibleValues` servlet bean ensures that customer searches only return products from catalogs the customer can view.

### *Input Parameters*

**itemDescriptorName** (Required)
This is the name of the item-descriptor in the repository XML file

**catalog**
The optional value that, if used, specifies a catalog ID. Only items located in the catalog specified here will be returned by the search. If you don't specify a catalog, the catalog specified in the user's profile is used.

**propertyName**
The optional value that, if used, must refer to a linked property. If this parameter is specified, repository items of the linked type will be returned.

**repository**
This parameter defines the repository to search. It can also be set via a properties file.

**sortProperties**
A string that specifies how to sort the list of repository items. This parameter is specified as a comma separated list of property names. The first name specifies the primary sort, the second specifies the secondary sort, etc. If the first character of each keyword is a -, this sort is performed in descending order. If it is a + or it is not a -, it is sorted in ascending order. Note: This parameter is only valid for repository items, it will not work with enumerated data-types.

### *Output Parameter*

**values**
Within the body of the output open parameter, this parameter is set to the list of possible values for the named item descriptor and property. The value will either be a list of tags (for enumerated properties) or an array of repository items.

### *Open Parameter*

**output**
This parameter is rendered once with the results of the search.

### *Example*

In this example, `RepositoryValues` is an instance of `CatalogPossibleValues`.

```
<dsp:droplet name="RepositoryValues">
 <dsp:param value="category" name="itemDescriptorName"/>
  <dsp:oparam name="output">
   <dsp:droplet name="ForEach">
    <dsp:param param="values" name="array"/>
    <dsp:param value="+displayName" name="sortProperties"/>
    <dsp:oparam name="output">
    <dsp:getvalueof id="option14" param="element.repositoryId"
               idtype="java.lang.String">
<dsp:option value="<%=option14%>"/>
</dsp:getvalueof>
    <dsp:valueof param="element.displayName"/>
   </dsp:oparam>
   </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>
```

## ClosenessQualifierDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.promotion.ClosenessQualifierDroplet` |
| **Component** | `atg/commerce/promotion/ClosenessQualifierDroplet` |

The `ClosenessQualifierDroplet` renders a list of `closenessQualifier` items associated with a given order. You can limit the type of `closenessQualifiers` that are returned by specifying a type: item, order, shipping and tax.

### *Input Parameters*

**elementName**
Names an `output` parameter that holds the returned `closenessQualifiers`. If no value is specified here, the `closenessQualifers` output parameter is used.

**order**
The order ID for which you want to access `closenessQualifiers`. If no order is specified here, the order ID is taken from the active shopping cart. The shopping cart is located through a reference from `ClosenessQualiferDroplet.promotionUpsellTools` property to the `/atg/commerce/promotion/PromotionUpsellTools.shoppingCartPath` property.

**type**
The type of `closenessQualifier` you want to access. Options include: `item`, `order`, shipping, `tax`, and `all`. Note that a value of `order` does not indicate that all `closenessQualifiers` for an order should be returned, but rather that `closenessQualifiers` designated for the order item type should be returned. Omitting this parameter causes Closeness Qualifiers of all types to be returned.

### Output Parameters

**closenessQualifiers**
The list of Closeness Qualifiers returned for the specified order. This parameter is applicable only when a replacement has not been specified by the `elementName` input parameter.

**errorMsg**
The error message to display to the user if an error occurs.

### Open Parameters

**empty**
This parameter is rendered if no Closeness Qualifiers are returned.

**error**
This parameter is rendered if an error occurs during processing.

**output**
This parameter is rendered if Closeness Qualifiers are returned.

### Example

In this example, the `ClosenessQualifierDroplet` determines whether the active user has earned any shipping-related Closeness Qualifiers. Excluding the order parameter causes ATG Commerce to make this determination based on the items in the current user's shopping cart. When the user is eligible for a Closeness Qualifier, the associated media item displays a message to the user.

```
<dsp:droplet name="/atg/commerce/promotion/ClosenessQualifierDroplet">
  <dsp:param name="type" value="shipping"/>
  <dsp:param name="elementName" value="closenessQualifiers"/>
  <dsp:oparam name="output">

     <dsp:droplet name="/atg/dynamo/droplet/ForEach">
       <dsp:param name="array" param="closenessQualifiers"/>
       <dsp:param name="elementName" value="closenessQualifier"/>
       <dsp:oparam name="output">

        <dsp:getvalueof id="media_url"
         param="closenessQualifier.upsellMedia" idtype="String">
         <dsp:include page="<%=media_url%>"/>
        </dsp:getvalueof>

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

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

## CollectionFilter

| | |
|---|---|
| **Class Name** | `atg.service.collections.filter.droplet.CollectionFilter` |
| **Components** | `/atg/commerce/collections/filter/droplet /InventoryFilterDroplet` `/atg/commerce/collections/filter/droplet /ProductFilterDroplet` `/atg/commerce/collections/filter/droplet /ExcludeItemInCartFilterDroplet` `/atg/commerce/collections/filter/droplet /CartSharingFilterDroplet` `/atg/commerce/collections/filter/droplet /GiftListSiteFilterDroplet` |

The `CollectionFilter` servlet beans use collection filtering components to reduce objects in a collection. Each servlet bean works with a different collection filtering component.

- `InventoryFilterDroplet` for a collection of products, determines which SKUs are available, as determined by the Inventory Manager at the time of execution. This servlet bean accesses `InventoryFilter`, which handles the filtering action.
- `ProductFilterDroplet` executes both `InventoryFilterDroplet` and `StartEndDateFilterDroplet` to return only those items that satisfy both sets of requirements. For example, a product that's in stock and has an active `startDate` will be returned while one with the same `startDate` but is out of stock won't be returned. This servlet bean accesses `ProductFilter`, which handles the filtering action.
- `ExcludeItemInCartFilterDroplet` returns those products that are not in the user's shopping cart. This servlet bean relies on `ExcludeItemsInCartFilter` to determine the items in the shopping cart and to eliminate them from the collection.
- `CartSharingFilterDroplet` uses the `CartSharingFilter` to return only products from sites that are within the cart sharing group.
- `GiftlistSiteFilterDroplet` filters a specified collection of gift lists or gift items.

The primary discussion of this class resides in *Appendix B: ATG Servlet Beans* of the *ATG Page Developer's Guide*. For details about collection filtering components and caching, see the *Filtering Collections* chapter in the *ATG Personalization Programming Guide*.

Note that each droplet's `filter` input parameter has a different default value of class `CollectionFilter`:

- `InventoryFilterDroplet` uses `/atg/registry/CollectionFilters/InventoryFilter`
- `ExcludeItemInCartFilterDroplet` uses `/atg/registry/CollectionFilters/ExcludeItemInCartFilter`

- `ProductFilterDroplet` uses `/atg/registry/CollectionFilters/ProductFilter`
- `CartSharingFilterDroplet` uses `/atg/registry/COllectionFilters/CartSharingFilter`

### *ProductFilterDroplet Example*

In this example, `ProductFilterDroplet` causes the filters (`InventoryFilter` and `StartEndDateFiler`)specified in the `ProductFilter` to apply their filtering mechanisms to a collection of products. The resultant collection of products that are in stock and active are displayed.

```
<dspel:droplet name="/atg/commerce/catalog/CategoryLookup">
   <dspel:param name="Id" param="catId"/>
   <dspel:oparam name="output">

   <%
    String collIdentifierKey = request.getParameter("catId") + "-childprd";
   %>
      <dspel:droplet name="/atg/collections/filter/droplet/ProductFilterDroplet">
         <dsp:param name="collection" param="item.childproducts/>
         <dsp:param name="collectionIdentifierKey" value="<%=collIdentifierKey
          %>"/>

         <dspel:oparam name="output">
             Featured Plants:
             <p><dsp:droplet name="/atg/dynamo/droplet/ForEach">
                   <dsp:param name="array" param="filteredCollection"/>

                   <dsp:oparam name="output">
                      <dspel:valueof param="element"/>
                   </dspel:oparam>
                </dsp:droplet>
         </dspel:oparam>

         <dspel:oparam name="empty">
            There are currently no outdoor plants
         </dspel:oparam>
      </dspel:droplet>
   </dspel:oparam>
</dspel:droplet>
```

### *GiftListSiteFilterDroplet Examples*

This JSP excerpt shows one example of how you can use `GiftlistSiteFilterDroplet` to filter gift lists. No site scope is passed in, so the `GiftlistSiteFilter` uses the `GiftlistManager` component's `siteScope`, which for the purposes of this example is set to the `atg.ShoppingCart` shareable type component. Also, no site IDs are provided, so the filtered gift lists will come from the current site and sites that share a shopping cart with the current site only.

```
<dsp:droplet
name="/atg/commerce/collections/filter/droplet/GiftlistSiteFilterDroplet">
  <&-- Specify the collection to filter --%>
  <dsp:param name="collection" bean="Profile.giftlists"/>

  <dsp:oparam name="output">

    <%-- Iterate through the collection. --%>
      <dsp:droplet name="="/atg/dynamo/droplet/ForEach">
        <dsp:param name="array" param="filteredCollection"/>

        <dsp:oparam name="output">
          <dsp:setvalue param="giftList" paramvalue="element"/>
          <dsp:getvalueof var="eventName" param="giftList.eventName"/>
          <c:out value="${eventName}"/>
        </dsp:oparam>
      </dsp:droplet>

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

This JSP excerpt filters a collection of wish list items. A site scope value of `current` is passed to the filter but no site IDs are passed, resulting in a collection of items from the current site only.

```
<dsp:droplet
name="/atg/commerce/collections/filter/droplet/GiftlistSiteFilterDroplet">
  <&-- Specify the collection to filter and the site scope. --%>
  <dsp:param name="collection" bean="Profile.wishlist.giftlistItems"/>
  <dsp:param name="siteScope" value="current"/>

  <dsp:oparam name="output">

    <%-- Iterate through the collection. --%>
      <dsp:droplet name="="/atg/dynamo/droplet/ForEach">
        <dsp:param name="array" param="filteredCollection"/>

        <dsp:oparam name="output">
          <dsp:setvalue param="giftItem" paramvalue="element"/>
          <dsp:getvalueof var="displayName" param="giftItem.displayName"/>
          <c:out value="${displayName}"/>
        </dsp:oparam>
      </dsp:droplet>

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

**Notes:**

- The `GiftlistSiteFilter` and `GiftlistSiteFilterDroplet` components can be configured to use caching to improve filtering performance, just as you would for any filter based on the `CollectionFilter` class. See *Caching Filtered Content* in the *ATG Personalization Programming Guide*.
- To pass site IDs to the `GiftlistFilterDroplet`, use a comma-separated list.

## ComplexPriceDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.priceLists.ComplexPriceDroplet` |
| **Component(s)** | `/atg/commerce/pricing/priceLists/ComplexPriceDroplet` (ATG Business Commerce only) |

The `ComplexPriceDroplet` servlet bean takes a complex price and returns the levels contained within it.

### *Input Parameters*

**complexPrice** (Required)
The ID of the complex price.

### *Output Parameters*

**levelMinimums**
The smallest quantity that applies to each level. This always begins with 1.

**levelMaximums**
The largest quantity that applies to each level. This is always 1 item shorter than `levelMinimums` since the last level has no maximum.

**prices**
The list of prices for each level. The last price in this array is always the `defaultPrice`.

**numLevels**
The length of the `quantities` and `prices` arrays.

### *Output Parameter*

**error**
The parameter rendered if there is an error.

### *Open Parameter*

**output**
The oparam rendered if the complex price is processed successfully.

### *Example*

The following example shows the JSP code for the `ComplexPriceDroplet`:

```
<dsp:droplet name="ComplexPriceDroplet">
 <dsp:param param="complexPrice" name="complexPrice"/>
 <dsp:oparam name="output">
  <table border=1>
   <dsp:droplet name="For">
    <dsp:param param="numLevels" name="howMany"/>
    <dsp:param value="index" name="indexName"/>
    <dsp:oparam name="output">
        <tr>
      <td>
       <dsp:valueof param="leveMinimums[param:index]"/> -
       <dsp:valueof param="levelMaximums[param:index]">?</dsp:valueof>
      </td>
      <td>
        <dsp:valueof param="prices[param:index]"/>
      </td>
    </tr>
 </dsp:oparam>
</dsp:droplet>
```

## ConvertAbandonedOrderDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.abandoned.ConvertAbandonedOrder Droplet` |
| **Component(s)** | `/atg/commerce/order/abandoned/ConvertAbandonedOrde rDroplet`<br>(Abandoned Order Services module only) |

The `ConvertAbandonedOrderDroplet` servlet bean replaces the abandoned, reanimated, or lost designation for a given order with a converted designation. 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.

See the *Using Abandoned Order Services* chapter in the *ATG Commerce Programming Guide* for detailed information on the Abandoned Order Services module.

### *Input Parameters*

**orderId** (Required)
The ID of the current order.

### *Output Parameters*

None.

### *Open Parameters*

**output**
This parameter is rendered when an order is converted.

**error**
This parameter is rendered if an error occurs.

### *Example*

```
<dsp:droplet name="ConvertAbandonedOrderDroplet">
 <dsp:param name="orderId" bean="/atg/commerce/ShoppingCart.current.id"/>…
</dsp:droplet>
```

## CostCenterDroplet

| | |
|---|---|
| **Class Name** | `atg.b2bcommerce.order.purchase.CostCenterDroplet` |
| **Component(s)** | `/atg/commerce/order/purchase/CostCenterDroplet` (ATG Business Commerce only) |

The `CostCenterDroplet` servlet bean is a request-scoped component whose service method is responsible for the following tasks:

- `CostCenter` initialization - `CostCenter` objects representing the user's authorized cost centers are created and added to the `CostCenterMapContainer`.

- `CommerceIdentifierCostCenter` initialization - New `CommerceIdentifierCostCenter` instances are created specific to the current `Order` and added to the `CommerceIdentifierCostCenterContainer`.

These tasks enable the user to associate their authorized cost centers with an order's various `CommerceIdentifier` objects. During initialization, `CostCenterDroplet` optionally creates one `CommerceIdentifierCostCenter` object for each object of a `CommerceIdentifier` type (for example, each `CommerceItem`, each `ShippingGroup`, the `Order`, and the `Tax`).

For more information on the cost center classes and framework, see the Managing Cost Centers chapter.

### *Input Parameters*

**clearAll**
If this parameter is set to true, `CostCenterDroplet` clears both `CommerceIdentifierCostCenterContainer` and the `CostCenterMapContainer`.

**clearCostCenterContainer**
If this parameter is set to true, `CostCenterDroplet` clears the `CommerceIdentifierCostCenters` in the `CommerceIdentifierCostCenterContainer`.

**clearCostCenterMap**
If this parameter is set to true, `CostCenterDroplet` clears the `CostCenters` in the `CostCenterMapContainer`.

**initCostCenters**
If this parameter is set to true, `CostCenterDroplet` places the `CostCenter` objects that represent the user's authorized cost centers into the `CostCenterMapContainer`.

**initItemCostCenters**
If this parameter is set to true, `CostCenterDroplet` creates a `CommerceIdentifierCostCenter` object for each `CommerceItem` in the order and adds them to the `CommerceIdentifierCostCenterContainer`.

**initShippingCostCenters**
If this parameter is set to true, `CostCenterDroplet` creates a `CommerceIdentifierCostCenter` object for each `ShippingGroup` in the order and adds them to the `CommerceIdentifierCostCenterContainer`.

**initTaxCostCenters**
If this parameter is set to true, `CostCenterDroplet` creates a `CommerceIdentifierCostCenter` object for the tax and adds it to the `CommerceIdentifierCostCenterContainer`.

**initOrderCostCenters**
If this parameter is set to true, `CostCenterDroplet` creates a `CommerceIdentifierCostCenter` object for the order and adds it to the `CommerceIdentifierCostCenterContainer`.

**useAmount**
The `useAmount` parameter describes the type of relationship that will exist between the commerce items and cost centers. This is dictated by whether your developer has allowed users to divide commerce items among cost centers by quantity or by amount. If this parameter is set to true, then amounts are used to

determine the cost center relationships. If the value is set to false, then quantities are used to determine the cost center relationships. The default value is false.

**order**
The user's order to which to assign cost centers.

### *Output Parameters*

**costCenters**
The list of the user's valid cost centers.

**ciccMap**
A map whose keys are the commerce identifiers, and whose values are the list of `CommerceIdentifierCostCenters` associated with that commerce identifier.

**order**
The user's order that was passed in.

### *Open Parameters*

**output**
The open parameter rendered if the operations are successful.

### *Example*

A common use of `CostCenterDroplet` is to build in its `output` parameter a form that 1) displays the output set by the servlet bean, and 2) enables the user to reassign items and other costs to other valid cost centers. This usage explains why, in the JSP example below, the `clearAll`, `clearCostCenterMap`, and `clearCostCenterContainer` parameters are all set to false. If the servlet bean is used with a form that gets reloaded after submission, setting these parameters to false prevents the erasure of any changes the user has made and submitted.

```
<dsp:droplet name="CostCenterDroplet">
  <dsp:param bean="ShoppingCartModifier.order" name="order"/>
  <dsp:param value="false" name="clearAll"/>
  <dsp:param value="false" name="clearCostCenterMap"/>
  <dsp:param value="false" name="clearCostCenterContainer"/>
  <dsp:param value="true" name="initCostCenters"/>
  <dsp:param value="true" name="initItemCostCenters"/>
  <dsp:param value="true" name="initShippingCostCenters"/>
  <dsp:param value="true" name="initTaxCostCenters"/>
  <dsp:param value="false" name="useAmount"/>
  <dsp:oparam name="output">some form
  </dsp:oparam>
</dsp:droplet>
```

## CouponDroplet

| | | |
|---|---|---|
| **Class Name** | `atg.commerce.promotion.CouponDroplet` | `atg.commerce.pricing.Currenc` |
| **Component(s)** | `/atg/commerce/promotion/CouponDroplet` | `/atg/commerce/pricing/Curren` |

The `CouponDroplet` servlet bean takes either a promotion object or promotion ID and generates a coupon for it in the Claimable repository. You could include the `CouponDroplet` in a targeted e-mail JSP, thereby creating a coupon code that is ready to send to a customer.

### *Input Parameters*

**promoId** (Either this or `promotions` must be used)
The ID of the promotion used to create the coupon.

**promotion**
No longer used.

**promotions** (Either this or `promoId` must be used)
The promotions object or objects used to create the coupon.

**displayName** (optional)
The display name of the coupon. If none is provided, the name of the first associated promotion is used for the coupon's display name. The default is null.

**usePromotionSiteConstraint** (optional)
If `true`, this flag means that the coupon that is created uses the same site constraints as the promotion with which it is associated. If more than one promotion is used to create the coupon, the promotions do not all have to have the same site constraints. The default is `false`.

### *Output Parameters*

**coupon**
The coupon object. The claim code for this coupon can be obtained by using `coupon.id`. Note that claim codes for coupons are case-sensitive. For example, `COUP100` and `coup100` are two different claim codes.

### *Open Parameters*

**output**
The open parameter rendered on successful creation of a coupon object.

**error**
The open parameter rendered if an error occurs during creation of a coupon.

### *Example*

The following example shows the JSP code for the `CouponDroplet`:

```
<dsp:importbean bean="/atg/commerce/promotion/CouponDroplet"/>
<h2>here is a coupon that was created: </h2>
<dsp:droplet name="CouponDroplet">
<dsp:param value="promo60001" name="promoId"/>
<dsp:oparam name="output">
   <dsp:valueof param="coupon.id">no value</dsp:valueof>
</dsp:oparam>
<dsp:oparam name="error">
</dsp:oparam>
</dsp:droplet>
```

## CurrencyCodeDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.CurrencyCodeDroplet` |
| **Component(s)** | `/atg/commerce/pricing/CurrencyCodeDroplet` |

The `CurrencyCodeDroplet` servlet bean takes a locale as input and returns the currency code for that locale.

### *Input Parameters*

**locale** (Required)
The current locale.

### *Output Parameter*

**currencyCode**
The ISO 4217 currency code for the locale.

### *Open Parameters*

**output**
The open parameter is always rendered.

### *Example*

The following example shows the JSP code for the `CurrencyCodeDroplet` servlet bean. In the example, the locale is retrieved from the user's profile and used to format the amount of a gift certificate.

```
<dsp:droplet name="CurrencyCodeDroplet">
  <dsp:param name="locale" bean="Profile.PriceList.locale"/>
  <dsp:oparam name="output">
    <dsp:getvalueof var="currencyCode" vartype="java.lang.String"
```

```
param="currencyCode"/>
    <dsp:getvalueof var="amount" vartype="java.lang.Double"
param="giftCertificate.amount"/>
  </dsp:oparam>
</dsp:droplet>
<%-- The format of message to display is:
A gift certificate has been purchased for you in the amount of {0} by {1} {2} --%>
<fmt:message key="emailtemplates_giftCertificate.purchasedInfo">
  <fmt:param>
    <fmt:formatNumber value="${amount}" type="currency"
currencyCode="${currencyCode}"/>
  </fmt:param>
….
```

## DisplaySkuProperties

| | |
|---|---|
| **Class Name** | `atg.commerce.catalog.DisplaySkuProperties` |
| **Component(s)** | `/atg/commerce/catalog/DisplaySkuProperties` |

The `DisplaySkuProperties` servlet bean takes a SKU item as input and renders a set of specified properties as a concatenated string. For example, you could use `DisplaySkuProperties` to display the `displayName`, `price`, and `description` properties of the SKU.

### *Input Parameters*

**sku** (Required)
The SKU item.

**delimiter**
Character to use as a separator between the different property values in the concatenated string. If you omit this parameter, then a space is used as the delimiter.

**propertyList or product**
You can use the `propertyList` parameter to specify the list of SKU properties to display as a comma-separated list, or you can use the `product` parameter to specify the parent product of the SKU. In the latter case, the list of properties is takes from the product's `displayableSkuAttributes` property.

**displayElementName**
The name to use as the parameter set within the `output` open parameter.

### *Output Parameter*

**displayElement**
The concatenated text string containing the property values. (You can specify a different name for this parameter through the optional `displayElementName` input parameter.)

### *Open Parameters*

**output**
The open parameter rendered if the output text string is not empty.

**empty**
The open parameter rendered if the output text string is empty.

### *Example*

The following example shows the JSP code for the `DisplaySkuProperties` servlet bean. In the example, the SKU is passed to `DisplaySkuProperties` by another servlet bean in which `DisplaySkuProperties` is nested.

```
<dsp:droplet name="/atg/commerce/catalog/DisplaySkuProperties">
   <dsp:param value=" | " name="delimiter"/>
   <dsp:param param="element" name="sku"/>
   <dsp:param value="displayName,listPrice,description" name="propertyList"/>
   <dsp:oparam name="output">
   <p><dsp:valueof param="displayElement"/>
   </dsp:oparam>
   <dsp:oparam name="empty">
   <p>There is no information available about this item.
   </dsp:oparam>
</dsp:droplet>
```

## ExcludeItemsInCartFilterDroplet

| | |
|---|---|
| **Class Name** | `atg.service.collections.filter.droplet.CollectionFilter` |
| **Component(s)** | `/atg/commerce/collections/filter/droplet/ExcludeItemsInCartFilterDroplet` |

The `ExcludeItemsInCartFilterDroplet` servlet bean allows you to filter items that are already in a shopper's cart out of a list.

### Input Parameters

**collectionIdentifierKey**
This value identifies the unfiltered collection. If it is not provided, the consultCache and updateCache parameters are forced to false.

**filter**
Any collection filter component.

**profile**
The value can be any profile repository item.

**collection** (Required)
The unfiltered collection.

**consultCache**
The string value can be "true" or "false".

**updateCache**
The string value can be "true" or "false".

### Output Parameters

**filteredCollection**
The filtered collection.

**errorMsg**
An error message when processing errors occur.

### Open Parameter

**output**
This tag is rendered once upon successful completion of the filter.

**empty**
This tag is rendered if the filtered collection is null or contains no objects.

**error**
This tag is rendered if an error occurs.

### Example

The following JSP example uses the `filteredCollection` parameter in `ExcludeItemsInCart` to display a filtered list of upsell products.

```
<dsp:droplet name="/atg/commerce/collections/filter/droplet/ExcludeItemsInCart
FilterDroplet">
   <dsp:param name="collection" param="category.upsellProducts"/>
   <dsp:oparam name="output">
      You may also like these<p>
      <dsp:droplet name="/atg/droplet/ForEach">
         <dsp:param name="array" param="filteredCollection" />
         <dsp:oparam name="output">
            Product <dsp:valueof param="element.repositoryId"/><p>
               <dsp:valueof param="element.description"/><p>
         </dsp:oparam>
      </dsp:droplet>
   </dsp:oparam>
</dsp:droplet>
```

## ForEachItemInCatalog

| | |
|---|---|
| **Class Name** | `atg.commerce.catalog.custom.ForEachItemInCatalog` |
| **Component(s)** | `/atg/commerce/catalog/ForEachItemInCatalog` |

The `ForEachItemInCatalog` servlet bean renders its `output` open parameter once for each element in the `array` input parameter that exists in the user's the current catalog.

The servlet bean is an instance of `atg.commerce.catalog.custom.ForEachItemInCatalog`, which extends `atg.droplet.ForEach`. `ForEachItemInCatalog` has one additional parameter, `profile`, which is used to get the current catalog for the user (from the `Profile.catalog` property).

### *Input Parameters*

**array** (Required)
The parameter that defines the list of items to output. This parameter can be Collection (Vector, List, or Set), Enumeration, Iterator, Map, Dictionary, or array.

**profile**
The current user's profile.

**sortProperties**
A string that specifies how to sort the list of items in the output array. Sorting can be performed on properties of JavaBeans, Dynamic Beans, or on Dates, Numbers, or Strings.

To sort JavaBeans, specify the value of `sortProperties` as a comma-separated list of property names. The first name specifies the primary sort, the second specifies the secondary sort, etc. If the first character of each keyword is a +, this sort is performed in ascending order. If it has a -, it is a descending order.

Example: To sort an output array of JavaBeans first alphabetically by title property and second in descending order of the size property:

```
<param name="sortProperties" value="+title,-size">
```

To sort Dates, Numbers, or Strings, specify the value of `sortProperties` with either a single "+" or a single "-" to indicate ascending or descending order respectively.

Example: To sort an output array of Strings in alphabetical order:

```
        <param name="sortProperties" value="+">
```

**reverseOrder**
A boolean value that specifies whether the traversal order in the array should be back to front or front to back. To sort from back to front, set this parameter to true. To sort from front to back, set this parameter to false. Note that this parameter only takes effect if the `sortProperties` input parameter is not set.

### *Output Parameters*

**index**
This parameter is set to the zero-based index of the current element of the array each time that the output parameter is rendered. The value of index for the first iteration is 0.

**count**
This parameter is set to the one-based index of the current element of the array each time that the output parameter is rendered. The value of count for the first iteration is 1.

**key**
If the array parameter is a Map or Dictionary, this parameter is set to the value of the key in the Map or Dictionary.

**element**
This parameter is set to the current element of the array each time that the index increments and the output parameter is rendered.

**size**
This parameter is set to the size of the array, if applicable. If the array is an Enumeration or Iterator, size is set to -1.

### *Open Parameters*

**output**
This parameter is rendered once for each element in the array.

**outputStart**
If the array is not empty, this parameter is rendered before any output elements. It can be used to render the heading of a table, for instance.

**outputEnd**
If the array is not empty, this parameter is rendered after all output elements. It can be used to render text following a table, for instance.

**empty**
This optional parameter is rendered if the array contains no elements.

### *Example*

The following JSP example uses `ForEachItemInCatalog` to display a list of related products for a given product on a product display page. Each related product is a link that redisplays the product display page with the selected related product.

```
<dsp:droplet name="/atg/commerce/catalog/ForEachItemInCatalog">
 <dsp:param param="element.relatedProducts" name="array"/>

  <dsp:oparam name="outputStart">
   <table border=0 cellpadding=1 width=100%>
   <tr><td><span class=smallbw>&nbsp;Related Items</span></td></tr>
   <tr><td></td></tr>
  </dsp:oparam>

   <dsp:oparam name="output">
    <tr><td>
    <dsp:getvalueof id="a28" param="element.template.url"
                idtype="java.lang.String">
<dsp:a href="<%=a28%>">
    <dsp:param param="element.repositoryId" name="id"/>
    <dsp:param value="jump" name="navAction"/>
    <dsp:param param="element" name="Item"/>
    <dsp:valueof param="element.displayName">
         No name</dsp:valueof></dsp:a></dsp:getvalueof><br>
    </td></tr>
   </dsp:oparam>

  <dsp:oparam name="outputEnd">
   </table>
  </dsp:oparam>
</dsp:droplet>
```

## GetApplicablePromotions

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.GetApplicablePromotions` |
| **Component(s)** | `/atg/commerce/pricing/GetApplicablePromotions` |

The `GetApplicablePromotions` servlet bean is used to determine which promotions can be applied to a given product or SKU.

The `GetApplicablePromotions` servlet bean is instantiated from `atg.commerce.pricing.GetApplicablePromotions`, which extends `atg.commerce.pricing.PriceItemDroplet`.

### *Input Parameters*

**item** (Required)
Either a `RepositoryItem` that represents the item for which promotions are to be evaluated, or a `CommerceItem` that can be evaluated directly. If the supplied item is a `RepositoryItem`, then a new `CommerceItem` is created.

**pricingModels**
The collection of pricing models (promotions) used to price the items. If this value is not supplied, then by default a collection of pricing models from the user's `PricingModelHolder` component is used. This component is resolved through the `userPricingModelsPath` property.

**locale**
The locale in which the evaluation should take place.

**profile**
The user for whom pricing is performed. If this value is not supplied, then the profile is resolved through the property `profilePath`.

**product**
The object that represents the product definition of the item to evaluate. Typically, items are SKUs. In that case, this is the product of the given SKU.

**elementName**
The name to use as the parameter set within the `output` open parameter.

**quantity**
The Long quantity of the input product. This parameter is used when constructing a `CommerceItem` from the supplied information.

### *Output Parameters*

**element**
The evaluated `CommerceItem`. This parameter name can be changed by setting the `elementName` input parameter.

**promotions**
The promotions that apply to this `CommerceItem`.

### *Open Parameters*

**output**
The open parameter rendered if the `CommerceItem` has been priced successfully.

### *Example*

In the following example, the promotions, locale and profile are extracted from the request because they are not supplied as parameters.

```
<dsp:droplet name="/atg/commerce/pricing/GetApplicablePromotions">
  <dsp:param name="product" param="product"/>
  <dsp:param name="item" param="product.childSkus[0]"/>
  <dsp:oparam name="output">
     <dsp:droplet name="ForEach">
            <dsp:param name="array" param="promotions"/>
          <dsp:oparam name="output">
               Promotion: <dsp:valueof param="element.displayName"/><br>
          </dsp:oparam>
     </dsp:droplet>
   </dsp:droplet>
```

## GiftCertificateAmountAvailable

| | |
|---|---|
| **Class Name** | `atg.commerce.claimable.GiftCertificateAmountAvailable` |
| **Component(s)** | `/atg/commerce/claimable/GiftCertificateAmountAvailable` |

The `GiftCertificateAmountAvailable` servlet bean allows you to see how much of the gift certificate amount remains after the current purchase.

### *Input Parameters*

**usedAmount** (Required)
The amount that is to be used on the gift certificate.

**giftCertificateNumber** (Required)
The gift certificate's claim code.

### *Output*

**amountAvailable**
The amount remaining on the gift certificate after deducting the `usedAmount`.

### *Open Parameters*

**output**
The open parameter is always rendered.

### *Example*

The following code example demonstrates how to use the `GiftCertificateAmountAvailable` droplet to to display the amount remaining on a user's gift certificate.

```
<dsp:droplet name="GiftCertificateAmountAvailable">
  <dsp:param name="giftCertificateNumber"
param="paymentGroup.giftCertificateNumber"/>
  <dsp:oparam name="output">
    <dt>
      <fmt:message key="common.remainingAmount"/><fmt:message
key="common.labelSeparator"/>
    </dt>
    <dd>
      <dsp:getvalueof var="amountRemaining" vartype="java.lang.Double"
param="amountAvailable"/>
      <fmt:formatNumber value="${amountRemaining}" type="currency"
currencyCode="${currencyCodeVar}"/>
    </dd>
  </dsp:oparam>
</dsp:droplet>
```

## GiftitemDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.gifts.GiftitemDroplet` |
| **Component(s)** | `/atg/commerce/gifts/BuyItemFromGiftlist`<br>`/atg/commerce/gifts/RemoveItemFromGiftlist` |

Servlet beans instantiated from the `GiftitemDroplet` class allow customers to buy or remove items from their own personal gift lists. Two Commerce servlet beans have been instantiated from `GiftitemDroplet`; they are `BuyItemFromGiftlist` and `RemoveItemFromGiftlist`.

### *Input Parameters*

**giftId** (Required)
The ID of the gift.

**giftlistId** (Required)
The ID of the gift list.

### *Output*

None.

### *Open Parameters*

**output**
The open parameter rendered if item is bought or removed successfully from list.

**error**
The open parameter rendered if an error occurs during processing.

### *Example*

The following code example demonstrates how to use the `RemoveItemFromGiftlist` component to remove items 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>
```

## GiftlistDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.gifts.GiftlistDroplet` |
| **Component(s)** | `/atg/commerce/gifts/GiftlistDroplet` |

The `GiftlistDroplet` servlet bean adds or removes other customers' gift lists from a customer's profile.

### *Input Parameters*

**action** (Required)
The action to perform on the gift list.

**giftlistId** (Required)
The ID of the gift list.

**profile**
The profile of the current customer. If not passed, the profile will be resolved by Nucleus.

### *Output Parameters*

None.

### *Open Parameters*

**output**
The open parameter rendered if the gift list is added or removed successfully from a profile.

**error**
The open parameter rendered if an error occurs while adding or removing the gift list.

### *Example*

The following code example demonstrates how to use the `GiftlistDroplet` to add a retrieved gift list 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:droplet>
</dsp:oparam>
</dsp:droplet>
```

## GiftShippingGroupDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.gifts.GiftShippingGroupDroplet` |
| **Component(s)** | `/atg/commerce/gifts/IsGiftShippingGroup` |

The `IsGiftShippingGroup` servlet bean checks a shipping group within a given order for gifts.

### *Input Parameters*

**sg** (Required)
The shipping group to check for gifts.

### *Output Parameter*

**giftlistId**
This parameter holds the gift list ID for a gift that's part of the shipping group.

### *Open Parameters*

**true**
The open parameter rendered if the shipping group contains one or more gifts.

**false**
The open parameter rendered if the shipping group does not contain gifts.

### *Example*

The following example shows the JSP code for the `IsGiftShippingGroup` servlet bean:

```
<dsp:droplet name="/atg/commerce/gifts/IsGiftShippingGroup">
<dsp:param param="sg" name="sg"/>
<dsp:oparam name="true">
   Gift in shipping group
</dsp:oparam>
<dsp:oparam name="false">
    No gift in shipping group
</dsp:oparam>
<dsp:oparam name="error">
    Error
</dsp:oparam>
</dsp:droplet>
```

## GiftShippingGroupsDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.gifts.GiftShippingGroupsDroplet` |
| **Component(s)** | `/atg/commerce/gifts/GiftShippingGroups` |

The `GiftShippingGroups` servlet bean checks for gifts within a given order. If gifts exist in the order, then it builds a collection of the shipping groups that contain the gifts.

### *Input Parameters*

**order** (Required)
The order to check for gifts.

### *Output Parameters*

**giftsg**
The collection of shipping groups in the order that contain one or more gifts.

**allgifts**
This parameter is boolean. The parameter's value is true if all the items in the order are gifts. The parameter's value is false if only some of the items in the order are gifts.

### *Open Parameters*

**true**
The open parameter rendered if the order contains gifts.

**false**
The open parameter rendered if the order does not contain gifts.

### *Example*

The following JSP example shows the parameters you can use to invoke the `GiftShippingGroups` servlet bean:

```
<dsp:droplet name="/atg/commerce/gifts/GiftShippingGroups">
<dsp:param param="order" name="order"/>
<dsp:oparam name="true">
   Gifts in order
</dsp:oparam>
<dsp:oparam name="false">
   No gifts
</dsp:oparam>
<dsp:oparam name="error">
   Error
</dsp:oparam>
</dsp:droplet>
```

## HasBusinessProcessStage

| | |
|---|---|
| **Class Name** | `atg.markers.bp.droplet.HasBusinessProcessStage` |
| **Component** | `/atg/commerce/bp/droplet/HasShoppingProcessStageDroplet` |

This servlet bean checks whether the order has reached a specified stage in the shopping process. It is an instance of `HasBusinessProcessStage` with the `businessProcessName` property set to `ShoppingProcess`.

### Input Parameters

**businessProcessName**
The name of the business process. If not specified, then we use the value of the servlet bean's `defaultBusinessProcessName` property, which is `ShoppingProcess` by default.

**businessProcessStage** (Required)
The stage within the business process. If you use the token value `!_anyvalue_!`, then the servlet renders the `true` open parameter if any business process stage has been reached.

### Output Parameters

**errorMsg**
The error message describing a failure.

### Open Parameters

**true**
Rendered if the stage has been reached.

**false**
Rendered if the stage has not been reached.

**error**
Rendered on error.

### Example

```
<dsp:droplet name="HasShoppingProcessStageDroplet">
  <dsp:param name="businessProcessStage" value="ShippingPriceViewed"/>
  <dsp:oparam name="true">
...
  </dsp:oparam>
  <dsp:oparam name="false">
...
  </dsp:oparam>
</dsp:droplet>
```

## InventoryDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.inventory.InventoryDroplet` |
| **Component(s)** | `/atg/commerce/inventory/InventoryLookup` |

The `InventoryLookup` servlet bean returns inventory information for a specified item.

### *Input Parameters*

**itemId** (Required)
The `catalogRefId` of the product catalog SKU whose inventory information will be retrieved.

**useCache**
If set to true, cached inventory data will be retrieved. Depending on how the inventory is updated, cached data may be out of date. For general store browsing where performance is critical, `useCache` should be set to true. If it is essential that the retrieved inventory information match the latest information in the repository, then set `useCache` to false.

### *Output Parameters*

**inventoryInfo**
The information object that holds the retrieved inventory information. The `inventoryInfo` object has the following properties:

- `availabilityStatus`: The numerical availability status
- `availabilityStatusMsg`: A string that maps to the numerical `availabilityStatus` as follows:
- 1000: INSTOCK
- 1001: OUTOFSTOCK
- 1002: PREORDERABLE
- 1003: BACKORDERABLE
- `availabilityDate`: The date on which the item will become available.
- `stockLevel`: The total number of units currently in stock.
- `preorderLevel`: 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.

**error**
Any exception that may have occurred while looking up the inventory information.

### *Open Parameters*

**output**
The open parameter rendered if the inventory information is successfully retrieved.

### *Example*

The following code sample is an example of using the `InventoryLookup` servlet bean:

```
<dsp:droplet name="/atg/commerce/inventory/InventoryLookup"> </td>
   <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>
```

## IsHardGoodsDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.IsHardGoodsDroplet` |
| **Component(s)** | `/atg/commerce/order/IsHardGoods` |

The `IsHardGoods` servlet bean takes an order and determines if it contains any items that will be shipped via a hardgood shipping group.

### *Input Parameters*

**order** (Required)
The order object that is inspected for a `HardgoodShippingGroup` with commerce items.

### *Output Parameters*

None.

### *Open Parameters*

**true**
The open parameter rendered if the order contains items in a `HardGoodShippingGroup`.

**false**
The open parameter rendered if the order doesn’t contain items in a `HardGoodShippingGroup`.

### *Example*

The following example illustrates the JSP code for the `IsHardGoods` servlet bean:

```
<dsp:droplet name="/atg/commerce/order/IsHardsGoods">
   <dsp:param param="someorder" name="order"/>
```

```
    <dsp:oparam name="true">
       Order contains items in a Hardgood shipping group
    </dsp:oparam>
    <dsp:oparam name="false">
       No items in a Hardgood shipping group
    </dsp:oparam>
</dsp:droplet>
```

## ItemLookupDroplet

| | |
|---|---|
| **Class Name** | `atg.repository.servlet.ItemLookupDroplet` |
| **Component(s)** | `/atg/commerce/catalog/CategoryLookup`<br>`/atg/commerce/catalog/MediaLookup`<br>`/atg/commerce/catalog/ProductLookup`<br>`/atg/commerce/catalog/SKULookup`<br>`/atg/commerce/gifts/GiftitemLookupDroplet`<br>`/atg/commerce/gifts/GiftlistLookupDroplet` |

Servlet beans instantiated from the `ItemLookupDroplet` class use an item's ID to look up the item in one or more repositories and render the item on the page. The `ItemLookupDroplet` class is included with the Adaptive Scenario Engine. For more information about its various input, output, and open parameters, see the `ItemLookupDroplet` entry in *Appendix B: ATG Servlet Beans* in the *ATG Page Developer's Guide*.

A number of Commerce servlet beans are instantiated from the `ItemLookupDroplet` class. These servlet beans are listed in the table at the top of this page.

### *Example*

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

```
<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 id="nameval2" param="giftlist.owner.id"
                    idtype="java.lang.String">
<dsp:oparam name="<%=nameval2%>">
        </dsp:oparam>
</dsp:getvalueof>
        </dsp:droplet>
     </dsp:oparam>
   </dsp:droplet>
 </dsp:oparam>
</dsp:droplet>
```

## ItemPricingDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.ItemPricingDroplet` |
| **Component(s)** | `None` |

`ItemPricingDroplet` is an abstract class that is used as the base class for pricing items and displaying the results to the customer. Both PriceEachItemDroplet and PriceItemDroplet extend this class.

If you extend this class, you must override the `performPricing` method to return the `CommerceItem`(s) that have been priced. These items are then bound into the `output` open parameter with the default name `element`.

### *Input Parameters*

The following *optional* parameters are permitted:

**pricingModels**
A collection of pricing models (promotions) that should be used to price the items. If this value is not supplied, then by default a collection of pricing models are used from the user's `PricingModelHolder` component. This component is resolved through the `userPricingModelsPath` property.

**locale**
The locale in which the pricing should take place.

**profile**
The user for whom pricing is performed. If this value is not supplied, then the profile is resolved through the `profilePath` property.

**product**
The object that represents the product definition of the item to price. Typically, the items that are priced are SKUs. In that case, this is the product that encompasses all of the SKUs.

**elementName**
The name to use as the parameter set within the `output` open parameter.

### *Output Parameters*

**element**
The priced `CommerceItem` or collection of priced `CommerceItems`. This parameter name can be changed by setting the `elementName` input parameter.

### *Open Parameters*

**output**
The open parameter rendered if the `CommerceItem`(s) has been successfully priced.

### *Example*

None. `ItemPricingDroplet` is an abstract class that is meant to be subclassed. Refer to PriceEachItemDroplet and PriceItemDroplet for JSP examples.

## MostRecentBusinessProcessStage

| | |
|---|---|
| **Class Name** | `atg.markers.bp.droplet.MostRecentBusinessProcessStage` |
| **Component** | `/atg/commerce/bp/droplet/MostRecentShoppingProcessStageDroplet` |

This servlet bean checks if the shopping process stage the order has reached most recently matches the specified stage. It is an instance of `MostRecentBusinessProcessStage` with the `businessProcessName` property set to `ShoppingProcess`.

### *Input Parameters*

**businessProcessName**
The name of the business process. If not specified, then we use the value of the servlet bean's `defaultBusinessProcessName` property, which is `ShoppingProcess` by default.

**businessProcessStage** (Required)
The stage within the business process.

### *Output Parameters*

**errorMsg**
The error message describing a failure.

**marker**
The matching stage reached, if found.

### *Open Parameters*

**true**
Rendered if the specified shopping process stage has been reached.

**false**
Rendered if the specified shopping process stage has not been reached.

**error**
Rendered on error.

### *Example*

```
<dsp:droplet name="MostRecentShoppingProcessStageDroplet">
  <dsp:param name="businessProcessStage" value="ShippingPriceViewed"/>
  <dsp:oparam name="true">
...
  </dsp:oparam>
  <dsp:oparam name="false">
...
  </dsp:oparam>
</dsp:droplet>
```

## NavHistoryCollector

| | |
|---|---|
| **Class Name** | `atg.repository.servlet.NavHistoryCollector` |
| **Component(s)** | `/atg/commerce/catalog/CatalogNavHistoryCollector` |

The `CatalogNavHistoryCollector` servlet bean can be used to create a "breadcrumb trail." That is, it constructs a list of the items the customer has visited to arrive at the current page and then creates and displays links to the items. This trail enables the customer to easily go back to previously visited items.

The `CatalogNavHistoryCollector` servlet bean manages the customer's navigation path by adding or removing the repository items that the customer has viewed from `CatalogNavHistory.navHistory`, the property that stores the stack of repository items.

### *Input Parameters*

**item** (either this or `itemName` must be used)
The repository item that is currently being viewed. This is the item that will be added to `CatalogNavHistory.navHistory`.

**itemName** (either this or `item` must be used)
The name of the current page. When this page is displayed as a breadcrumb, this name will be used as the anchor text in the link back to the page.

**navAction**
The operation to be performed on the `navHistory` stack. Choices are push, pop, and jump. An unset `navAction` will be treated as push.

**navCount** (Required)
Used to detect the use of the Back button in order to reset the navigation path. To use this feature, at the top of each page set a page parameter named `navCount` to the value of the `CatalogNavHistory.navCount` property. For example:

```
        <param name="navCount" value="bean:CatalogNavHistory.navCount">
```

Then, when you call the `CatalogNavHistoryCollector` servlet bean, set the value of the `navCount` input parameter to the current value of the `CatalogNavHistory.navCount` property. `CatalogNavHistoryCollector` compares this value to the value of the `navCount` page parameter. If the values are not equal, then the Back button was used to get to the page.

### *Output Parameters*

None.

### *Open Parameters*

**output**
None.

### *Example*

This example demonstrates how to *collect* the navigation history using `CatalogNavHistoryCollector`. This JSP fragment should be invoked in each page that adds an item to the `navHistory` stack.

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

<dsp:oparam name="output">

   <dsp:droplet name="/atg/commerce/catalog/CatalogNavHistoryCollector">
   <dsp:param param="element" name="item"/>
   <dsp:param value="push" name="navAction"/>
   <dsp:param bean="/atg/commerce/catalog/CatalogNavHistory.navCount"
              name="navCount"/>
   </dsp:droplet>

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

This second example demonstrates how to *render* a list of the locations the customer has visited. These locations are displayed as a path, such as Fruit > Citrus Fruit > Oranges. Each category or product name in the path is a link back to the corresponding item. Clicking one of these links causes the items below it in the hierarchy to pop off the stack.

```
<dsp:droplet name="/atg/dynamo/droplet/ForEach">
 <dsp:param bean="CatalogNavHistory.navHistory" name="array"/>
 <dsp:oparam name="output">
    <dsp:getvalueof id="a6" param="element.template.url"
                idtype="java.lang.String">
<dsp:a href="<%=a6%>">
    <dsp:param param="element.repositoryId" name="itemId"/>
    <dsp:param value="pop" name="navAction"/>
    <dsp:param bean="CatalogNavHistory.navCount" name="navCount"/>
    <dsp:valueof param="element.displayName"/>
    </dsp:a></dsp:getvalueof>
 </dsp:oparam>
</dsp:droplet>
```

# OrderLookup

| | |
|---|---|
| **Class Name** | `atg.commerce.order.OrderLookup` |
| **Component(s)** | `/atg/commerce/order/AdminOrderLookup`<br><br>`/atg/commerce/order/OrderLookup`<br>(ATG Consumer Commerce only) |

The `OrderLookup` servlet bean retrieves one or more `Order` objects, depending on the supplied input parameters. It enables you to retrieve a single order, all orders placed by a particular user, or all orders placed by a particular user that are in a specific state. For more information on the `OrderLookup` servlet bean, see the Implementing Order Retrieval chapter.

`OrderLookup` has a security feature that allows the current user to view only her own orders. By default, this feature is enabled for `/atg/commerce/order/OrderLookup`. To disable the feature, set the `enableSecurity` property to false.

### *Input Parameters*

**orderId** (Either this or `userId` is required.)
The ID of the order to retrieve.

**userId** (Either this or `orderId` is required.)
The ID of the user profile whose orders will be retrieved.

**state**
The desired state of the orders to retrieve.

This parameter can be used in conjunction with `userId`. You can specify one of the following:

- any one of the states defined in `atg.commerce.states.OrderStates`
- open
- closed

If you specify "open," then all orders whose states are specified in the `openStates` property of the `OrderLookup` component are returned; by default, this list of states is set to the following:

```
submitted
processing
pending_merchant_action
pending_customer_action
```

If you specify "closed," then all orders whose states are specified in the `closedStates` property of the `OrderLookup` component are returned; by default, this list of states is set to the following:

```
no_pending_action
```

You can override either list of states by using the optional `openStates` or `closedStates` input parameter.

**openStates**
A comma-separated list of states that correspond to "open" state.

This parameter can be used in conjunction with the `state` input parameter when the `state` input parameter is set to "open." Use this optional parameter when you want to override the configured list of states in the `openStates` property of the `OrderLookup` component.

**closedStates**
A comma-separated list of states that correspond to "closed" state.

This parameter can be used in conjunction with the `state` input parameter when the `state` input parameter is set to "closed." Use this optional parameter when you want to override the configured list of states in the `closedStates` property of the `OrderLookup` component.

**sortBy**
A string that specifies an `Order` property by which to sort the orders.

This parameter can be used in conjunction with `userId`. When using this parameter, you can specify the name of any Order Repository property (that is, the name of any property defined in `orderrepository.xml`), such as `id`, `state`, or `submittedDate`.

**sortAscending**
True or false. This parameter is used in conjunction with the `sortBy` input parameter. If set to true, the `Order` objects in the resulting array are sorted in ascending order by the property specified in the `sortBy` input parameter. The default value is false.

**numOrders**
The number of orders to return for the given query.

**startIndex**
The index of the first order in the result set. This parameter is useful for cycling through a large number of orders.

**queryTotal**
Indicates whether the number of retrieved orders will be calculated into a total that's accessible through the `totalCount` and `total_count` output parameters. Setting this property to `false` prevents the total count from being generated, regardless of the value specified in the `queryTotal` property. Omitting this parameter causes the default value, `true`, to be used. Use this parameter to ensure that queries to the database are made only when necessary.

**queryTotalOnly**
Indicates whether the total number of orders and the orders themselves are produced from the servlet bean. Setting this parameter to `true` makes the total number of retrieved orders available through the `totalCount` and `total_count` output parameters. The orders themselves are not retrieved or accessible. Use this parameter to ensure that queries to the database are made only when necessary.

Omitting this parameter, which is the same as setting it to `false`, saves a list of order objects to the `output` open parameter as well as the total number of orders to the `totalCount` and `total_count` output parameters.

If `queryTotal=false` (orders, no total) and `queryTotalOnly=true` (total, no orders), a total is generated only as specified in the `queryTotalOnly` parameter.

**siteIds**

A collection of site IDs used to limit the query to orders associated with the specified sites.

**siteScope**

If you are using ATG's multisite feature, you can filter orders by site. Use `siteScope` if no `siteId` is specified, and provide one of the following scopes:

- `null` or `all`: Finds orders for all sites
- `current`: Finds orders for the current site, as determined by the site context
- `shareableTypeId`: pass in the ID of a shareable type, such as `atg.ShoppingCart`, to find orders for all sites in the sharing group for that type.

The default `siteScope` can be set through a configurable property on the component. The default value is null.

### *Output Parameters*

**result**
The array of `Order` objects. If the `orderId` input parameter was used, then this parameter contains a single `Order` object.

**errorMsg**
If an error occurred, this is the detailed error message for the user.

**count**
The size of the array of `Order` objects.

**totalCount**
If the `queryTotal` property is set to true, this parameter indicates the total number of orders that meet the criteria for the order lookup.

**total_count**
Identical to the `totalCount` output parameter (above).

**startRange**
The index number that marks the beginning of a range of orders. For example, if 5 orders were returned from a given `OrderLookup` query, the `startRange` is set to 1.

**endRange**
The index number that marks the end of a range of orders. For example, if 5 orders were returned from a given `OrderLookup` query with a `startRange` of 6, the `endRange` is set to 10.

**nextIndex**
The index of the first order in the next set of results. If the `startIndex` or `numOrders` input parameter was null, then this parameter will be null.

**previousIndex**
The index of the first order in the previous set of results. If the `startIndex` or `numOrders` input parameter was null, then this parameter will be null.

### *Open Parameters*

**output**
The open parameter rendered if the orders are successfully retrieved.

**empty**
The open parameter rendered if there are no orders to return.

**error**
The open parameter rendered if an error occurs.

### *Example*

The following example describes how to use the `OrderLookup` servlet bean to retrieve all open orders for the current user and to display their IDs.

```
<dsp:droplet name="/atg/commerce/order/OrderLookup">
 <dsp:param bean="/atg/userprofiling/Profile.repositoryId" name="userId"/>
 <dsp:param value="open" name="state"/>
 <dsp:oparam name="output">
```

```
    <dsp:droplet name="/atg/dynamo/droplet/ForEach">
       <dsp:param param="result" name="array"/>
       <dsp:oparam name="outputStart">
          <OL>
       </dsp:oparam>
       <dsp:oparam name="output">
          <LI> <dsp:valueof param="element.id">no order number</dsp:valueof>
       </dsp:oparam>
       <dsp:oparam name="outputEnd">
          </OL>
       </dsp:oparam>
       <dsp:oparam name="empty">
          No open orders.
       </dsp:oparam>
    </dsp:droplet>
 </dsp:oparam>
 <dsp:oparam name="error">
    <span class=profilebig>ERROR:
       <dsp:valueof param="errorMsg">no error message</dsp:valueof>
    </span>
 </dsp:oparam>
</dsp:droplet>
```

## PaymentGroupDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.purchase.PaymentGroupDroplet` |
| **Component(s)** | `/atg/commerce/order/purchase/PaymentGroupDroplet` |

The `PaymentGroupDroplet` servlet bean is used to initialize a user's `PaymentGroups` and `CommerceIdentifierPaymentInfo` objects for use by the `PaymentGroupFormHandler`. The `PaymentGroupDroplet` servlet bean is instantiated from `atg.commerce.order.purchase.PaymentGroupDroplet`. The `PaymentGroupDroplet` class is composed of the following containers:

- `PaymentGroupMapContainer` - a container for the user's named `PaymentGroup` objects.
- `CommerceIdentifierPaymentInfoContainer` - a container for the `CommerceIdentifierPaymentInfo` objects for the `CommerceIdentifier` objects in the user's `Order`.

For more information on these containers, the `PaymentGroupDroplet` servlet bean, and the `PaymentGroupFormHandler` form handler, refer to the *Preparing a Complex Order for Checkout* section in the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

### *Input Parameters*

**clear**
When this parameter is set to true, PaymentGroupDroplet clears both the user's CommerceIdentifierPaymentInfoContainer and PaymentGroupMapContainer.

**clearPaymentGroups**
When this parameter is set to true, PaymentGroupDroplet clears the user's PaymentGroupMapContainer. This should be done at least once per Order if the PaymentGroup objects are subject to change after placing an Order (most likely because the PaymentGroup objects are from the ClaimableRepository).

**clearPaymentInfos**
When this parameter is set to true, PaymentGroupDroplet clears the user's CommerceIdentifierPaymentInfoContainer. This should be done at least once per Order to create fresh CommerceIdentifierPaymentInfo objects that refer to each Order's unique CommerceIdentifier objects.

**createAllPaymentInfos**

When this parameter is set to true, PaymentGroupDroplet creates an OrderPaymentInfo for all PaymentGroups in the user's profile. This option supports a different type of user interface than that which is described in the Creating Potential Payment Groups section. In this UI, the user is presented with a form that has a list of PaymentGroups. The user provides the amount to be paid by each PaymentGroup directly in the form, effectively setting the amount property for each OrderPaymentInfo object. If the user adds additional PaymentGroups during the checkout process, you should call the PaymentGroupDroplet again to create OrderPaymentInfo objects for the newly added PaymentGroups.

This option is False by default.

**initBasedonOrder**

When this parameter is set to true, PaymentGroupDroplet creates a CommerceIdentifierPaymentInfo object for each PaymentGroup relationship object in the Order. The types of CommerceIdentifierPaymentInfo objects that are created correspond to the PaymentGroup relationship types. For example, if a PaymentGroupCommerceItemRelationship exists in the Order, PaymentGroupDroplet creates a corresponding CommerceItemPaymentInfo object and adds it to the CommerceIdentifierPaymentInfoContainer. Each CommerceIdentifierPaymentInfo object is initialized with the PaymentGroup that exists in its corresponding PaymentGroup relationship object.

This option is provided for the scenario where a customer has already gone part way through the checkout process and the order already contains some PaymentGroup relationship objects. Set to False by default.

**initItemPayment**

When this parameter is set to true, PaymentGroupDroplet creates a CommerceItemPaymentInfo object for each CommerceItem in the order and adds them to the CommerceIdentifierPaymentInfoContainer. If a user has a default PaymentGroup in his or her profile, the CommerceItemPaymentInfo object is initialized with that PaymentGroup. Set to False by default.

**Note:** A CommerceItemPaymentInfo object is a CommerceIdentifierPaymentInfo object whose CommerceIdentifier is a CommerceItem; it is used for CommerceItem payment information.

**initOrderPayment**

When this parameter is set to true, PaymentGroupDroplet creates an OrderPaymentInfo object and adds it to the CommerceIdentifierPaymentInfoContainer. If a user has a default PaymentGroup in his or her profile, the OrderPaymentInfo object is initialized with that PaymentGroup. Set to True by default.

**Note:** An OrderPaymentInfo object is a CommerceIdentifierPaymentInfo object whose CommerceIdentifier is an Order; it is used for Order payment information.

**initPaymentGroups**
When this parameter is set to true, the PaymentGroup types supplied in the paymentGroupTypes input parameter will be initialized.

**initShippingPayment**

When this parameter is set to true, PaymentGroupDroplet creates a ShippingGroupPaymentInfo object for each ShippingGroup in the order and adds them to the CommerceIdentifierPaymentInfoContainer. If a user has a default PaymentGroup in his or her profile, the ShippingGroupPaymentInfo object is initialized with that PaymentGroup. Set to False by default.

**Note:** A ShippingGroupPaymentInfo object is a CommerceIdentifierPaymentInfo object whose CommerceIdentifier is a ShippingGroup; it is used for ShippingGroup payment information.

**initTaxPayment**

When this parameter is set to true, PaymentGroupDroplet creates a TaxPaymentInfo object and adds it to the CommerceIdentifierPaymentInfoContainer.

**Note:** A TaxPaymentInfo object is a CommerceIdentifierPaymentInfo object whose CommerceIdentifier is an Order; it is used for tax payment information.

**order**
The user's order. You can use this parameter to override the default setting for PaymentGroupDroplet.order.

**paymentGroupTypes**
A comma-separated list of PaymentGroup types, such as creditCard, storeCredit, giftCertificate, that is used to determine which PaymentGroupInitializer components are executed.

The PaymentGroupInitializer components are responsible for creating and initializing the appropriate PaymentGroup objects and adding them to the PaymentGroupMapContainer. Each possible `PaymentGroup` type is configured to reference a `PaymentGroupInitializer` component in the `PaymentGroupInitializer ServiceMap`. The keys into the map are the `Strings` supplied in this `paymentGroupTypes` input parameter; the values are the underlying `PaymentGroupInitializer` components that do the initialization work for that `PaymentGroup` type.

By default, the `PaymentGroupDroplet` servlet bean is configured with the following `PaymentGroupInitializer` components:

```
## ServiceMap of paymentGroupTypes to PaymentGroupInitializer Nucleus components

paymentGroupInitializers=\
giftCertificate=/atg/commerce/order/purchase/GiftCertificateInitializer,\
storeCredit=/atg/commerce/order/purchase/StoreCreditInitializer,\
creditCard=/atg/commerce/order/purchase/CreditCardInitializer
```

### *Output Parameters*

**paymentInfo**
The Map referenced by the `CommerceIdentifierPaymentInfoContainer`.

**order**
The `Order` object that represents the user's order.

**paymentGroups**
The Map referenced by the `PaymentGroupMapContainer`.

### *Open Parameters*

**output**
The open parameter rendered always.

### *Example*

This example creates `CreditCard`, `StoreCredit`, and `GiftCertificate PaymentGroup` objects based on their availability for the current user. Additionally, it creates `CommerceItemPaymentInfo` objects, `ShippingGroupPaymentInfo` objects, and a `TaxPaymentInfo` object. The example enables the user to pay for `CommerceIdentifiers` at the line item level with any of their available `PaymentGroup` objects.

```
<dsp:droplet name="PaymentGroupDroplet">
  <dsp:param value="true" name="clear"/>
  <dsp:param value="giftCertificates, storeCredit, creditCard"
              name="paymentGroupTypes"/>
  <dsp:param value="true" name="initPaymentGroups"/>
  <dsp:param value="true" name="initItemPayment"/>
  <dsp:param value="true" name="initTaxPayment"/>
  <dsp:param value="true" name="initShippingPayment"/>
  <dsp:oparam name="output">Manipulation of objects here…
  </dsp:output>
</dsp:droplet>
```

## PossibleValues

| | |
|---|---|
| **Class Name** | `atg.repository.servlet.PossibleValues` |
| **Component(s)** | `/atg/commerce/catalog/RepositoryValues` |

The `RepositoryValues` servlet bean queries a repository and returns an array of possible values for a given repository item type.

The `RepositoryValues` servlet bean is instantiated from class `atg.repository.servlet.PossibleValues`. A `PossibleValues` servlet bean, instantiated from the same class, is included with the ATG Adaptive Scenario Engine . For more information about its various parameters and a JSP example, see *Appendix B: ATG Servlet Beans* in the *ATG Page Developer's Guide*.

## PriceDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.priceLists.PriceDroplet` |
| **Component(s)** | `/atg/commerce/pricing/priceLists/PriceDroplet` (ATG Business Commerce only) |

The `PriceDroplet` servlet bean returns a price for a given product or SKU. `PriceDroplet` should not be confused with PriceItemDroplet, which uses the `PricingEngine` to actually calculate the price for a single item.

### *Input Parameters*

**product** (Either this or `sku` must be used)
The product for which a price is desired.

**sku** (Either this or `product` must be used)
The SKU for which a price is desired.

**parentSku**
If the SKU to be priced is an option in a configurable SKU, this is the base sku item within the context of which the configurable option should be priced.

**priceList**
The `priceList` from which to retrieve the price.

### *Output Parameters*

**price**
The price repository item.

**error**
The error that occurred when retrieving the price.

### *Open Parameters*

**output**
Rendered if the price for the product or SKU is retrieved successfully.

**empty**
Rendered if there is no price for the given product or SKU. If you are using combined price lists and SKU-based pricing, you can use this parameter to render the SKU-based price if there is no price list price for the item (see the *Using Price Lists in Combination with SKU-Based Pricing* section of the *ATG Commerce Programming Guide* for information on how to configure this feature).

### *Example*

The following example illustrates the JSP code for the `PriceDroplet` servlet bean:

```
<dsp:droplet name="/atg/commerce/pricing/priceLists/PriceDroplet">
   <dsp:param name="sku" value="sku" />
   <dsp:param name="product" value="product" />
   <dsp:oparam name="output">
      <dsp:droplet name="/atg/dynamo/droplet/Switch">
         <dsp:param name="value" param="price.pricingScheme" />
         <dsp:oparam name="listPrice">
            <dsp:valueof param="price.listPrice" converter="currency">no
             price</dsp:valueof>
         </dsp:oparam>
      </dsp:droplet>
   </dsp:oparam>
</dsp:droplet>
```

# PriceEachItemDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.PriceEachItemDroplet` |
| **Component(s)** | `/atg/commerce/pricing/PriceEachItem` |

The `PriceEachItem` servlet bean is to price dynamically a collection of items by taking promotions into account. If you need to show only static prices, you can retrieve the list or sale prices directly from the SKU object.

The `PriceEachItem` servlet bean is instantiated from `atg.commerce.pricing.PriceEachItemDroplet`, which extends `atg.commerce.pricing.ItemPricingDroplet`.

For information on a servlet bean that can be used to price dynamically a *collection* of items, refer to PriceItemDroplet.

### *Input Parameters*

**items** (Required)
Either a collection of `RepositoryItems` that represent the items to be priced, or a collection of `CommerceItems` that can be priced directly. If the supplied items are `RepositoryItems,` then a new collection of `CommerceItems` is created.

**pricingModels**
The collection of pricing models (promotions) used to price the items. If this value is not supplied, then by default a collection of pricing models from the customer's `PricingModelHolder` component is used. This component is resolved through the `userPricingModelsPath` property.

**locale**
The locale in which the pricing should take place.

**profile**
The user for whom pricing is performed. If this value is not supplied, then the profile is resolved through the `profilePath` property.

**product**
The object that represents the product definition of the items to price. Usually, the items that are priced are SKUs. In that case, this is the product that encompasses all of the SKUs.

**elementName**
The name to use as the parameter set within the `output` open parameter.

### *Output Parameters*

**element**
The collection of priced `CommerceItems`. This parameter name can be changed by setting the `elementName` input parameter.

### Open Parameters

**output**
The open parameter rendered if the `CommerceItems` have been priced successfully.

### Example

In the following example, the promotions, locale, and profile are extracted from the request, since they are not supplied as parameters.

```
<dsp:droplet name="/atg/commerce/pricing/PriceEachItem">
 <dsp:param param="product.childSKUs" name="items"/>
 <!-- the product param is already defined in this scope so we do not
  need to set it -->
 <dsp:oparam name="output">
 <!-- Now iterate over each of the CommerceItems to display the prices -->
  <dsp:droplet name="/atg/dynamo/droplet/ForEach">
   <dsp:param param="element" name="array"/>
   <dsp:param value="pricedItem" name="elementName"/>
   <dsp:oparam name="output">
    <dsp:valueof param="pricedItem.auxiliaryData.catalogRef.displayName"/> -
 <!-- Toggle a different display depending if the item is on sale or not -->
    <dsp:droplet name="Switch">
    <dsp:param param="pricedItem.priceInfo.onSale" name="value"/>
    <dsp:oparam name="false">
     <dsp:valueof param="pricedItem.priceInfo.amount" converter="currency">
                     no price</dsp:valueof>
    </dsp:oparam>
    <dsp:oparam name="true">
     List price for <dsp:valueof param="pricedItem.priceInfo.listPrice"
                         converter="currency">no price
            </dsp:valueof>
                 on sale for <dsp:valueof param="pricedItem.priceInfo.salePrice"
                                    converter="currency"/>!
    </dsp:oparam>
    </dsp:droplet><BR>
   </dsp:oparam>
  </dsp:droplet>
 </dsp:oparam>
</dsp:droplet>
```

## PriceItemDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.PriceItemDroplet` |
| **Component(s)** | `/atg/commerce/pricing/PriceItem` |

The `PriceItem` servlet bean is used to dynamically price a single item by taking promotions into account. Note that it does not apply order-dependent promotions, but only those that apply at the item level. If you need to show only a static price, you can retrieve the list or sale price directly from the SKU object.

The `PriceItem` servlet bean is instantiated from `atg.commerce.pricing.PriceItemDroplet`, which extends `atg.commerce.pricing.ItemPricingDroplet`.

For information on a servlet bean that can be used to price dynamically a *collection* of items, refer to PriceEachItemDroplet.

### *Input Parameters*

**item** (Required)
Either a `RepositoryItem` that represents the item to be priced, or a `CommerceItem` that can be priced directly. If the supplied item is a `RepositoryItem`, then a new `CommerceItem` is created for pricing.

**pricingModels**
The collection of pricing models (promotions) used to price the items. If this value is not supplied, then by default a collection of pricing models from the user's `PricingModelHolder` component is used. This component is resolved through the `userPricingModelsPath` property.

**locale**
The locale in which the pricing should take place.

**profile**
The user for whom pricing is performed. If this value is not supplied, then the profile is resolved through the property `profilePath`.

**product**
The object that represents the product definition of the item to price. Typically, items that are priced are SKUs. In that case, this is the product of the given SKU.

**elementName**
The name to use as the parameter set within the `output` open parameter.

**quantity**
The Long quantity of the input product that should be priced. This parameter is used when constructing a `CommerceItem` from the supplied information.

### *Output Parameters*

**element**
The priced `CommerceItem`. This parameter name can be changed by setting the `elementName` input parameter.

### *Open Parameters*

**output**
The open parameter rendered if the `CommerceItem` has been priced successfully.

### *Example*

In the following example, the promotions, locale and profile are extracted from the request because they are not supplied as parameters.

```
<dsp:droplet name="/atg/commerce/pricing/PriceItem">
<dsp:param param="sku" name="item"/>
<dsp:param param="product" name="product"/>
<dsp:param value="pricedItem" name="elementName"/>
<dsp:oparam name="output">
    <dsp:valueof param="pricedItem.priceInfo.amount" converter="currency">
                        no price</dsp:valueof>
</dsp:oparam>
</dsp:droplet>
```

## PriceRangeDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.PriceRangeDroplet` |
| **Component(s)** | `/atg/commerce/pricing/PriceRangeDroplet` |

When given a product, this droplet determines the highest and lowest price for the range of SKUs associated with the product.

### *Input Parameters*

**productId** (required)
The id of product repository item that needs to be priced

**pricelist** (optional)
The price list to be used for pricing. If it is not set, the the profile's assigned price list will be used.

**salePriceList** (optional)
The sale price list to be used for pricing. If it is not set, the the profile's assigned sale price list will be used.

### *Output Parameters*

**lowestPrice**
Double representing the lowest price found.

**highestPrice**
Double representing the highest price found.

### *Open Paramaters*

**output**
 - always serviced

### *Example*

The example here shows the `PriceRangeDroplet`’s use.

```
<dsp:droplet name="/atg/commerce/custsvc/pricing/PriceRangeDroplet">
   <dsp:param name="productId" value="productId">
   <dsp:param name="priceList" bean="priceList">
   <dsp:oparam name="output">
      <dsp:valueof param="lowestPrice">no price no price
```

## ProductListContains

| | |
|---|---|
| **Class Name** | `atg.commerce.catalog.comparison.ProductListContains` |
| **Component(s)** | `/atg/commerce/catalog/comparison/ProductListContains` |

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

### *Input Parameters*

**productList** (Required)
The `ProductComparisonList` object to examine.

**productID** (Required)
The repository ID of the product to look for in `productList`.

**categoryID**
The repository ID of the category to look for in `productList`.

If you don’t specify a category ID, 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.

**skuID**
The repository ID of the SKU to look for in `productList`.

If you don't specify a SKU, 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.

**repositoryKey**
The key to pass to `CatalogTools` to select a product catalog repository in which to look for the item. The key-to-catalog mapping is defined in `CatalogTools`. If this parameter is unset, the default product catalog repository is used.

This optional parameter is useful for localization, which often requires the use of alternate product catalogs for different locales.

### *Output Parameters*

None

### *Open Parameters*

**true**
Rendered if the product comparison list contains the specific product, category, and SKU.

**false**
Rendered if the product comparison list doesn't contain the specified product, category, and SKU.

### *Example*

This JSP example shows how to add or remove a single product from a product comparison list. The example assumes that the `ProductListContains` servlet bean is embedded in a product display page using the following:

```
<dsp:include page="example.jsp"><dsp:param name="product"
            value="current product"/></dsp:include>
```

where *current product* is an expression that provides access to the product displayed on the page.

The given product is passed into the servlet bean in the `productId` input parameter. The `ProductListContains` servlet bean then checks whether it is stored in the product comparison list in `ProductList`.

If the product is in the product comparison list, then the servlet bean renders the `true` open parameter on the product display page. The user can then click the "Remove from comparison list" submit button to remove the product from the list. If the product isn't in the product comparison list, then the servlet bean renders the `false` open parameter on the product display page. The user can then click the "Add to comparison list" submit button to add the product to the list.

```
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductList"/>
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductListContains"/>
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductListHandler"/>
```

```
<dsp:form action="product.jsp" method="POST">
<dsp:droplet name="ProductListContains">
 <dsp:param bean="ProductList" name="productList"/>
 <dsp:param param="product.repositoryId" name="productID"/>

 <dsp:oparam name="true">
  <dsp:input bean="ProductListHandler.productID" paramvalue="productID"
             type="hidden"/>
  <dsp:input bean="ProductListHandler.removeProduct" value="Remove from comparison
       list" type="submit"/>
 </dsp:oparam>

 <dsp:oparam name="false">
  <dsp:input bean="ProductListHandler.productID" paramvalue="productID"
             type="hidden"/>
  <dsp:input bean="ProductListHandler.addProduct" value="Add to
   comparison list" type="submit"/>
 </dsp:oparam>

</dsp:droplet>
</dsp:form>
```

## PromotionDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.promotion.PromotionDroplet` |
| **Component(s)** | `/atg/commerce/promotion/PromotionDroplet` |

The `PromotionDroplet` servlet bean associates a promotion with a user profile. The promotion is added to the list of promotions in the `activePromotions` property of the user profile.

### *Input Parameters*

**promotion** (Required)
The promotion to be associated with the profile. The value of this parameter must be of type `RepositoryItem`.

**profile**
The user profile associated with the promotion. If this parameter is not supplied or is not an instance of `Profile`, then it is resolved from Nucleus.

### *Output Parameters*

**error**
An exception that occurred while associating the promotion the user profile.

### Open Parameters

**error**
This parameter is rendered if an error occurs.

### Example

```
<dsp:droplet name="/atg/commerce/promotion/PromotionDroplet">
  <dsp:param param="someIncomingPromotion" name="promotion"/>
</dsp:droplet>
```

## ReanimateAbandonedOrderDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.abandoned.ReanimateAbandonedOrderDroplet` |
| **Component(s)** | `/atg/commerce/order/abandoned/ReanimateAbandonedOrderDroplet`<br>(Abandoned Order Services module only) |

The `ReanimateAbandonedOrderDroplet` servlet bean 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 was 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.

See the *Using Abandoned Order Services* chapter in the *ATG Commerce Programming Guide* for detailed information on the Abandoned Order Services module.

### Input Parameters

**orderId** (Required)
The ID of the current order.

### *Output Parameters*

None.

### *Open Parameters*

**output**
This parameter is rendered when an order is reanimated.

**error**
This parameter is rendered if an error occurs.

### *Example*

```
<dsp:droplet name="ReanimateAbandonedOrderDroplet">
  <dsp:param name="orderId"
   bean="/atg/commerce/ShoppingCart.current.id"/>…
</dsp:droplet>
```

## RemoveBusinessProcessStage

| | |
|---|---|
| **Class Name** | `atg.markers.bp.RemoveBusinessProcessStage` |
| **Component** | `/atg/commerce/bp/droplet/RemoveShoppingProcessStageDroplet` |

This servlet bean removes existing shopping process stage markers that match the stage specified. It is an instance of `RemoveBusinessProcessStage` with the `businessProcessName` property set to `ShoppingProcess`.

### *Input Parameters*

**businessProcessName**
The name of the business process. If not specified, then we use the value of the servlet bean’s `defaultBusinessProcessName` property, which is `ShoppingProcess` by default.

**businessProcessStage** (Required)
The stage within the business process. If you use the value `!_anyvalue_!`, then all business process stage markers will be removed.

### *Output Parameters*

**errorMsg**
The error message describing a failure.

**markerCount**
The number of stage reached markers removed

### *Open Parameters*

**output**
This parameter is rendered on successful completion

**error**
This parameter is rendered on error.

### *Example*

This example removes a `ShippingPriceViewed` stage:

```
<dsp:droplet name="RemoveShoppingProcessStageDroplet">
  <dsp:param name="businessProcessStage" value="ShippingPriceViewed"/>
</dsp:droplet>
```

This example removes all shopping process stage markers that are found:

```
<dsp:droplet name="RemoveShoppingProcessStageDroplet">
  <dsp:param name="businessProcessStage" value="!_anyvalue_!"/>
</dsp:droplet>
```

## RepriceOrder

| | |
|---|---|
| **Class Name** | `atg.commerce.order.purchase.RepriceOrder` |
| **Component(s)** | `/atg/commerce/order/purchase/RepriceOrderDroplet` |

The `RepriceOrderDroplet` servlet bean is an instance of `atg.commerce.order.purchase.RepriceOrder`, which extends `atg.service.pipeline.servlet.PipelineChainInvocation`. The `RepriceOrder` class provides the objects that are needed to execute a repricing pipeline chain as convenient properties. Typically, execution of a repricing pipeline chain requires the `Order`, the `Profile`, the `OrderManager`, and the user's `PricingModelHolder`. While the `PipelineChainInvocation` class is flexible enough to handle these requirements (and enable you to configure the properties as required in a Map), `RepriceOrder` is conveniently configured to reference these objects. This means the page developer doesn't need to supply them as input parameters every time `RepriceOrderDroplet` is invoked.

By default, `RepriceOrderDroplet` is configured to invoke the `repriceOrder` pipeline chain to reprice an order. As such, it provides a mechanism for updating the price of an order every time a customer accesses the shopping cart page. This is useful if your sites enable customers to access their shopping carts through non-form actions, such as standard hyperlinks. Because of dynamic pricing, customers could potentially view inaccurate prices in their shopping carts when accessing the shopping cart through a hyperlink.

For more information on repricing an order, refer to the *Repricing Orders* section and the *Checking Out an Order* section in the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*. Also refer to the Repricing Shopping Carts section in the *Implementing Shopping Carts* chapter. For more information on the `repriceOrder` pipeline chain, refer to the *Commerce Processor Chains* section of the *ATG Commerce Programming Guide*.

### *Input Parameters*

**pricingOp** (Required)
The pricing operation to be executed. Acceptable pricing operations are defined in the `atg.commerce.pricing.PricingConstants` interface and are listed in the following table.

| 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` |
| `ORDER` | `PricingConstants.OP_REPRICE_ORDER` |
| `TAX` | `PricingConstants.OP_REPRICE_TAX` |
| `NO_REPRICE` | `PricingConstants.OP_NO_REPRICE` |

**priceList**
The ID of the pricelist to use. If you don’t specify this parameter, but you use pricelists to determine the order price, the pricelist specified for the active user is used. This parameter lets you provide an alternate pricelist.

**chainId**
The ID of the pipeline chain to execute. If this parameter is not set, then the servlet bean looks for a configured pipeline chain ID in the `defaultChainId` property. By default, `RepriceOrderDroplet.defaultChainId` is set to `repriceOrder`.

**paramObject**
A parameter Object used as an argument to the `runProcess` method of the `PipelineManager` during pipeline chain execution. If this parameter is not set, then the servlet bean looks for a configured value in

the `extraParametersMap` property. It uses the configured value to construct a HashMap to use as an argument to the `PipelineManager` during pipeline chain execution. `extraParametersMap` represents a mapping of the new Hash Map's keys to the request parameters that are bound to the new Hash Map's values. By default, `RepriceOrderDroplet.extraParametersMap` is empty.

**pipelineManager**
The `PipelineManager` instance to use for pipeline chain execution. If this parameter is not set, then the servlet bean looks for a configured `PipelineManager` in the `defaultPipelineManager` property. By default, `RepriceOrderDroplet.defaultPipelineManager` is set to `/atg/commerce/PipelineManager`.

### *Output Parameters*

**exception**
Any exception that occurred during the repricing process.

**pipelineResult**
The `PipelineResult` instance returned by the `PipelineManager` after a successful pipeline chain execution.

### *Open Parameters*

**success**
The parameter rendered after successful execution of the pipeline chain.

**successWithErrors**
This parameter is rendered after a successful pipeline chain execution that contains error messages in the `PipelineResult` object.

**failure**
The parameter rendered after an unsuccessful attempt to execute the pipeline chain.

### *Example*

```
<dsp:droplet name="RepriceOrderDroplet">
  <dsp:param value="ORDER_SUBTOTAL" name="pricingOp"/>
</dsp:droplet>
```

## SetLastUpdatedDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.abandoned.SetLastUpdatedDroplet` |
| **Component(s)** | `/atg/commerce/order/abandoned/SetLastUpdatedDroplet` (Abandoned Order Services module only) |

The `SetLastUpdatedDroplet` servlet bean 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.

See the *Using Abandoned Order Services* chapter in the *ATG Commerce Programming Guide* for detailed information on the Abandoned Order Services module.

### *Input Parameters*

**orderId** (Required)
The ID of the current order.

### *Output Parameters*

None.

### *Open Parameters*

**output**
This parameter is rendered when the order has an `abandonmentInfo` item or one was created for the order by this servlet bean.

**error**
This parameter is rendered if an error occurs.

### *Example*

```
<dsp:droplet name="SetLastUpdatedDroplet">
  <dsp:param name="orderId"
   bean="/atg/commerce/ShoppingCart.current.id"/>
</dsp:droplet>
```

## ShipItemRelPrice

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.ShipItemRelPriceDroplet` |
| **Component(s)** | `/atg/commerce/pricing/ShipItemRelPrice` (ATG Business Commerce only) |

The `ShipItemRelPrice` servlet bean returns a price for a `ShippingGroupCommerceItemRelationship`. It looks at the range of the `ShippingGroupCommerceItemRelationship` and returns the sum of the amounts of the `DetailedItemPriceInfo` objects that apply to the range.

Refer to the *Working With Purchase Process Objects* chapter in the *ATG Commerce Programming Guide* for more information about `ShippingGroupCommerceItemRelationship` and `Range` objects. Refer to the *Using and Extending Pricing Services* chapter of the *ATG Commerce Programming Guide* for more information about the `DetailedItemPriceInfo` price holding class.

### *Input Parameters*

**shipItemRel** (Required)
The `ShippingGroupCommerceItemRelationship` for which you want a price.

**propertyName**
The property of the individual `DetailedItemPriceInfo` objects used to calculate the price of the `ShippingGroupCommerceItemRelationship`. The default property is `amount`.

### *Output Parameters*

**price**
The price of the `ShippingGroupCommerceItemRelationship`.

**error**
Any exception that may have occurred while processing the price of the `ShippingGroupCommerceItemRelationship`.

### *Open Parameters*

**output**
The open parameter rendered if the price is processed successfully.

**error**
The open parameter rendered if an error occurs.

### *Example*

```
<dsp:droplet name="/atg/commerce/pricing/ShipItemRelPrice">
<dsp:param param="shipItemRel" name="shipItemRel"/>
<dsp:oparam name="output">Price = <dsp:valueof param="price"
       converter="currency">no price</dsp:valueof>
</dsp:oparam>
</dsp:droplet>
```

## ShippableGroupsDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.fulfillment.ShippableGroupsDroplet` |
| **Component(s)** | `/atg/commerce/fulfillment/droplet/ShippableGroupsDroplet` |

The `ShippableGroupsDroplet` servlet bean displays all orders with "shippable" shipping groups, that is, all orders with shipping groups in a PENDING_SHIPMENT state.

### *Input Parameters*

None.

### *Output Parameters*

**orders**
The array of order IDs for the orders that contain one or more shippable shipping groups.

**shippingGroups**
The array of shipping group IDs for the shippable shipping groups.

**count**
The number of shippable shipping groups.

**error**
Any exception that may have occurred while retrieving the shippable groups.

### *Open Parameters*

**shipSchedule**
Renders information related to the `HardgoodShipper` scheduled service, including the current time, the time of the last run, and the schedule.

**output**
The open parameter rendered if there are orders with shippable shipping groups.

**empty**
The open parameter rendered if there are no orders with shippable shipping groups.

### *Example*

```
<dsp:droplet name="/atg/commerce/fulfillment/droplet/ShippableGroupsDroplet">
  <dsp:oparam name="shipSchedule">
    <table border="1">
      <tr>
        <td colspan=2>Shipper</td>
      </tr>
      <tr>
        <td>Current Time</td><td><dsp:valueof
            param="currentTime">NA</dsp:valueof></td>
      </tr>
      <tr>
        <td>Last Run</td><td><dsp:valueof param="lastRun">NA</dsp:valueof></td>
      </tr>
      <tr>
        <td>Schedule</td><td><dsp:valueof param="schedule">NA</dsp:valueof></td>
      </tr>
    </table>
  </dsp:oparam>

  <dsp:oparam name="output">
    <table border="1">
      <tr>
        <td><b>Order ID</b></td><td><b>Shipping Group Id</b></td>
      </tr>
    <dsp:droplet name="/atg/dynamo/droplet/For">
      <dsp:param param="count" name="howMany"/>
      <dsp:oparam name="output">
         <tr>
           <td><dsp:valueof param="orders[param:index]">
                             no orderId</dsp:valueof></td>
           <td><dsp:valueof param="shippingGroups[param:index]">no
                shippingGroupId</dsp:valueof></td>
         </tr>
      </dsp:oparam>
    </dsp:droplet>
    </table>
    <br>
  </dsp:oparam>

  <dsp:oparam name="empty">
    <table border="1">
      <tr>
        <td><b>Order ID</b></td><td><b>Shipping Group Id</b></td>
      </tr>
      <tr>
        <td colspan=2>There are no shippable groups.</td>
      </tr>
    </table>
  </dsp:oparam>

</dsp:droplet>
```

## ShippingDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.fulfillment.ShippingDroplet` |
| **Component(s)** | `/atg/commerce/fulfillment/droplet/ShippingDroplet` |

The `ShippingDroplet` servlet bean informs the fulfillment system when a given shipping group has shipped.

### *Input Parameters*

**orderId** (Required)
The ID of the order that contains the shipped shipping group.

**shippingGroupId** (Required)
The ID of the shipping group that has shipped.

### *Output Parameters*

**status**
This parameter is set to one of two values: `ShipCallSucceeded` or `ShipCallFailed`.

### *Open Parameters*

**output**
This open parameter is rendered if the fulfillment system is informed when a given shipping group has shipped.

### *Example*

```
<dsp:droplet name="ShippingDroplet">
  <dsp:param param="orderId" name="orderId"/>
  <dsp:param param="shippingGroupId" name="shippingGroupId"/>
  <dsp:oparam name="output">
    <dsp:valueof param="status">no status</dsp:valueof>
  </dsp:oparam>
</dsp:droplet>
```

## ShippingGroupDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.order.purchase.ShippingGroupDroplet` |
| **Component(s)** | `/atg/commerce/order/purchase/ShippingGroupDroplet` |

ShippingGroupDroplet is a servlet bean that initializes ShippingGroup objects and CommerceItemShippingInfo objects for use by the ShippingGroupFormHandler form handler. The ShippingGroupDroplet class is composed of the following containers:

- ShippingGroupMapContainer – a container for the ShippingGroup objects that represent the user's authorized shipping groups.
- CommerceItemShippingInfoContainer – a container for the CommerceItemShippingInfo objects for the commerce items in the user's order.

For more information on these containers, the ShippingGroupDroplet servlet bean, and the ShippingGroupFormHandler form handler, refer to the *Preparing a Complex Order for Checkout* section in the *Configuring Purchase Process Services* chapter in the *ATG Commerce Programming Guide*.

### *Input Parameters*

**clear**
When this parameter is set to True, ShippingGroupDroplet clears both the user's CommerceItemShippingInfoContainer and ShippingGroupMapContainer.

**clearShippingGroups**
When this parameter is set to True, ShippingGroupDroplet clears the user's ShippingGroupMapContainer.

**clearShippingInfos**
When this parameter is set to True, ShippingGroupDroplet clears the user's CommerceItemShippingInfoContainer. This should be done at least once per Order to create fresh CommerceItemShippingInfo objects that refer to the unique CommerceItem objects in each Order.

**createOneInfoPerUnit**
When set to True, ShippingGroupDroplet creates a CommerceItemShippingInfo object for each individual unit contained in each CommerceItem. For example, a CommerceItem with a quantity of five will have five CommerceItemShippingInfo objects created for it. If a user has a default ShippingGroup in his or her profile, each CommerceItemShippingInfo object is initialized with that ShippingGroup. Set to False by default.

**initBasedOnOrder**
When set to True, ShippingGroupDroplet creates a CommerceItemShippingInfo object for each ShippingGroupCommerceItemRelationship object in the Order. The CommerceItemShippingInfo is initialized with the ShippingGroup that exists in the ShippingCommerceItemRelationship. This option is provided for the scenario where a customer has already gone part way through the checkout process and the order already contains some ShippingGroupCommerceItemRelationship objects. Set to False by default.

**initShippingGroups**
When this parameter is set to True, the ShippingGroup types supplied in the shippingGroupTypes input parameter will be initialized.

**initShippingInfos**
When set to True, ShippingGroupDroplet creates a CommerceItemShippingInfo object for each

CommerceItem in the Order. If a user has a default ShippingGroup in his or her profile, the CommerceItemShippingInfo object is initialized with that ShippingGroup. Set to True by default.

**order**
This parameter may be used to override the component's default setting for the user's order.

**shippingGroupTypes**
A comma-separated list of ShippingGroup types, such as hardgoodShippingGroup or electronicShippingGroup, that is used to determine which ShippingGroupInitializer components are executed.

The ShippingGroupInitializer components are responsible for creating and initializing the appropriate ShippingGroup objects and adding them to the ShippingGroupMap container. Each possible ShippingGroup type is configured to reference a ShippingGroupInitializer component in the ShippingGroupInitializers ServiceMap. The keys into the map are the Strings supplied in this shippingGroupTypes input parameter; the values are the underlying ShippingGroupInitializer components that do the initialization work for that ShippingGroup type.

By default, the ShippingGroupDroplet servlet bean is configured with the following ShippingGroupInitializer components:

```
## ServiceMap of shippingGroupTypes to ShippingGroupInitializer Nucleus components

shippingGroupInitializers=\
hardgoodShippingGroup=/atg/commerce/order/purchase/
                      HardgoodShippingGroupInitializer,\
electronicShippingGroup=/atg/commerce/order/purchase/
                        ElectronicShippingGroupInitializer
```

### *Output Parameters*

**shippingGroups**
The Map referenced by the ShippingGroupMapContainer.

**order**
The Order object that represents the user's order.

### *Open Parameters*

**output**
The open parameter rendered always.

### *Example*

This example creates HardgoodShippingGroup objects based on their availability for the current user. Additionally, it creates CommerceItemShippingInfo objects, which facilitate the association between any CommerceItems in the Order and any of the user's HardgoodShippingGroup objects.

```
<dsp:droplet name="ShippingGroupDroplet">
  <dsp:param value="true" name="clear"/>
  <dsp:param value="hardgoodShippingGroup" name="shippingGroupTypes"/>
  <dsp:param value="true" name="initShippingGroups"/>
  <dsp:param value="true" name="initShippingInfos"/>
  <dsp:oparam name="output"> Manipulation of objects here…
  </dsp:output>
</dsp:droplet>
```

## SiteIdForCatalogItem

| | |
|---|---|
| **Class Name** | `atg.droplet.multisite.SiteIdForItemDroplet` |
| **Component(s)** | `/atg/commerce/multisite/SiteIdForCatalogItem` |

**Note:** This droplet is intended for use with ATG's multisite feature. See the.*ATG Multisite Administration Guide.*

The `SiteIdForItemDroplet` is described in the *ATG Page Developer's Guide*; see that document for information. The `SiteIdForCatalogItem` implementation sets the `shareableTypeId` to `atg.ShoppingCart` by default.

## UnitPriceDetailDroplet

| | |
|---|---|
| **Class Name** | `atg.commerce.pricing.UnitPriceDetailDroplet` |
| **Component(s)** | `/atg/commerce/pricing/UnitPriceDetailDroplet` |

The `UnitPriceDetailDroplet` can provide detailed price information for units in a given line item, along with discount information. This is useful for displaying information such as "3 @ $2.00" or "2 @ $10.00, 1 @ $0.00 (Buy 2 Get 1 Free)".

### *Input Parameters*

**item** (Required)
The line item for which you want to display details.

### Output Parameters

**UnitPriceBeans**
The droplet output is a list of `UnitPriceBean` objects.

### Open Parameters

**output**
The open parameter is always rendered.

### Example

The following example shows a portion of a JSP that uses the `UnitPriceDetailDroplet` servlet bean to display detailed information about line items in an order.

```
<dsp:droplet name="UnitPriceDetailDroplet">
       <dsp:param name="item" param="currentItem"/>
       <dsp:oparam name="output">

         <dsp:getvalueof var="unitPriceBeans" vartype="java.lang.Object"
param="unitPriceBeans"/>
         <c:forEach var="unitPriceBean" items="${unitPriceBeans}">

           <dsp:param name="unitPriceBean" value="${unitPriceBean}"/>
           <dsp:getvalueof var="quantity" vartype="java.lang.Double"
param="unitPriceBean.quantity"/>
           <p class="price">
             <fmt:formatNumber value="${quantity}" type="number"/>
             <fmt:message key="common.atRateOf"/>
             <dsp:getvalueof var="unitPrice" vartype="java.lang.Double"
param="unitPriceBean.unitPrice"/>
             <fmt:formatNumber value="${unitPrice}" type="currency"
currencyCode="${currencyCode}"/>
           </p>

           <dsp:getvalueof var="pricingModels" vartype="java.lang.Object"
param="unitPriceBean.pricingModels"/>
           <c:choose>
             <c:when test="${not empty pricingModels}">
               <c:forEach var="pricingModel" items="${pricingModels}">
                 <dsp:param name="pricingModel" value="${pricingModel}"/>
                  <p class="note">
                   (<dsp:valueof param="pricingModel.description">
                      <fmt:message key="common.promotionDescriptionDefault"/>
                    </dsp:valueof>)
                 </p>
               </c:forEach><%-- End for each promotion used to create the unit
price --%>
             </c:when>
```

```
              <c:otherwise>
                <dsp:getvalueof var="currentItemOnSale"
param="currentItem.priceInfo.onSale"/>
                <c:if test='${currentItemOnSale == "true"}'>
                  <p><fmt:message key="cart_detailedItemPrice.salePriceB"/></p>
                </c:if>
              </c:otherwise>
            </c:choose>

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

## ViewItemEventSender

| | |
|---|---|
| **Class Name** | `atg.userprofiling.ViewItemEventSender` |
| **Component(s)** | `/atg/commerce/catalog/CategoryBrowsed`<br>`/atg/commerce/catalog/ProductBrowsed` |

`CategoryBrowsed` and `ProductBrowsed` are two servlet beans instantiated from class `atg.userprofiling.ViewItemEventSender`. These servlet beans send JMS messages to the messaging system when a customer views items in the catalog.

### *Input Parameters*

**eventobject**
The repository item the customer has viewed, which will be sent within the event message. This is either a product or category repository item, depending on which servlet bean you use (`CategoryBrowsed` or `ProductBrowsed`).

### *Output Parameters*

None.

### *Open Parameters*

**error**
The open parameter rendered if an error occurs while sending the message to the messaging system.

### *Example*

The following example shows a portion of a JSP that uses the `ProductBrowsed` servlet bean to send a message when a product is viewed. The product's repository ID is passed to this page (via the `ItemId` parameter) from the page that links to it.

```
<dsp:droplet name="/atg/commerce/catalog/ProductLookup">
<dsp:param param="ItemId" name="id"/>

<dsp:oparam name="output">
   <dsp:droplet name="/atg/commerce/catalog/ProductBrowsed">
     <dsp:param param="element" name="eventobject"/>
   </dsp:droplet>
</dsp:oparam>
</dsp:droplet>
```

# Index

## A

Abandoned Order Converted event element, 107
Abandoned Order Lost event element, 107
Abandoned Order Reanimated event element, 107
abandoned order services, 99
  business user overview, 99
  creating abandonment scenarios, 103
  orders, abandoned, 100
  orders, converted, 100
  orders, lost, 100
  orders, reanimated, 100
  scenario action elements, 107
  scenario event elements, 106
  testing abandonment scenarios, 105
action elements
  Add Item to Order, 95
  creating custom elements, 94
  described, 94
  Fill Related Items to Slot, 95
  Give Promotion, 95
  Revoke Promotion, 95
Add Item to Order action element, 95
AddBusinessProcessStage servlet bean, 228
adding a new cost center, 70
AddItemToCartServlet, 226
AdminOrderLookup, 277
advanced searches, 123
Approval Complete Event event element, 86
Approval Required Event event element, 86
ApprovalFormHandler, 190, 218
ApprovalRequiredDroplet, 189, 217, 218, 229
ApprovalResolvedDroplet, 218, 229
ApprovedDroplet, 191, 231
assigning a default cost center to a user, 70
assigning price lists to users, 37
ATG Control Center
  using to search catalogs, 25
  using to view catalogs, 11
AvailableShippingMethods, 166, 233
AvailableShippingMethodsDroplet, 233
AvailableStoreCredits, 235

## B

B2BOrderLookup, 183, 208, 218, 236
B2BOrderStates, 207
BuyItemFromGiftlist, 264

## C

CartModifierFormHandler, 148, 149
carts. See shopping carts
catalog folders
  defined, 16
  editing, 20
catalog folderss
  creating, 16
catalog items
  deleting, 22
  duplicating, 21
  moving, 21
catalog servlet beans, 221
CatalogItemLookupDroplet, 110, 218, 240
CatalogLookup, 110
CatalogNavHistory, 114
CatalogNavHistoryCollector, 114, 275
CatalogPossibleValues, 243
catalogs
  adding a subcatalog, 20
  adding to categories, 20
  creating, 16
  defined, 16
  displaying items, 110
  editing, 20
  hierarchical navigation in, 112
  historical navigation, 114
  item types, 11
  looking up items in, 110
  organizational models, 9
  searching, 25, 118
  viewing as hierarchy, 12
  viewing as lists, 15
CatalogSearchFormHandler, 118
  configuring, 120
categories
  defined, 16
  editing, 20

CategoryBrowsed, 308
CategoryLookup, 110, 240, 272
child categories
  creating, 17
  displaying, 113
child products
  displaying, 113
claimable servlet beans, 222
ClosenessQualifierDroplet, 244
collection filtering servlet beans, 222
CollectionFilter, 246
CommerceIdentifierPaymentInfo, 170
comparison lists. See product comparison
ComplexPriceDroplet, 249
condition elements
  creating custom elements, 94
  described, 93
  Item Where, 94
  Order Where, 94
Convert Abandoned Order action element, 108
ConvertAbandonedOrderDroplet, 250
cost centers, managing in the ACC, 2
cost centers,managing in the ACC, 69
CostCenterDroplet, 251
CouponDroplet, 254
coupons
  adding the coupon to the repository, 67
  adding the promotion to the repository, 66
  setting up, 66
CreateCreditCardFormHandler, 167
CreateElectronicShippingGroupFormHandler, 155
CreateHardgoodShippingGroupFormHandler, 155
CreateInvoiceRequestFormHandler, 167
cross-selling products, 95
CrossSellProductsSlot scenario template, 97
CurrencyCodeDroplet, 255
customer history, collecting, 115
customer path, rendering, 116

**D**

discounts. See promotions
DisplaySkuProperties, 256
droplets. See servlet beans

**E**

elements. See scenario elements
event elements
  Approval Complete Event, 86
  Approval Required Event, 86
  Approval Update Event event element, 87
  described, 85
  FulfillOrderFragment, 87
  Gift Purchased, 87
  Inventory Threshold Reached, 87
  Invoice Is Created, 88
  Invoice Is Removed, 88
  Invoice Is Updated, 88
  Item Added to Order, 88
  Item Quantity Changed in Order, 89
  Item Removed from Order, 89
  ModifyOrder, 89
  ModifyOrderNotification, 89
  Order Changes, 89
  Order Submitted, 90
  Orders Merged, 90
  Payment Group Changes, 90
  Promotion Revoked, 91, 92
  Scenario Added an Item to an Order, 92
  Scheduled Order Event, 92
  Shipping Group Changes, 92
  UpdateInventory, 93
  Uses Promotion, 93
event messages, sending when items are viewed, 111
ExcludeItemsInCartFilterDroplet, 257

**F**

Fill Related Items to Slot action element, 95
folders, creating, 22
ForEachItemInCatalog, 259
  page developer example, 111
fulfillment servlet beans, 223
FulfillOrderFragment event element, 87

**G**

GearConfigFormHandler, 207, 217
gears
  Order Approval, 209
  Order Status, 201
  servlet beans, 223
GetApplicablePromotions, 261
gift list servlet beans, 223
gift lists
  filtering, 196
Gift Purchased event element, 87
GiftCertificateAmountAvailable, 263
GiftitemDroplet, 264
GiftitemLookupDroplet, 272
GiftlistDroplet, 265
GiftlistLookupDroplet, 272
GiftShippingGroupDroplet, 266
GiftShippingGroups, 267
GiftShippingGroupsDroplet, 267
Give Promotion action element, 53, 95

**H**

HasBusinessProcessStage servlet bean, 268
HasFunction, 217
hierarchical searches, 123

**I**

images
  adding, 23
  specifying, 24

INCOMPLETE order state, 145
inventory framework servlet beans, 223
Inventory Threshold Reached event element, 87
InventoryDroplet, 269
InventoryLookup, 269
Invoice Is Created event element, 88
Invoice Is Removed event element, 88
Invoice Is Updated event element, 88
IsGiftShippingGroup, 266
IsHardGoods, 271
IsHardGoodsDroplet, 271
Item Added to Order event element, 88
Item Quantity Changed in Order event element, 89
Item Removed from Order event element, 89
Item Where condition element, 94
ItemLookupDroplet, 272
ItemPricingDroplet, 273

## K

keyword searches, 121, 125

## L

location jumps, tracking, 117
Log Promotion Information action element, 108

## M

managing cost centers using the ACC, 2, 69
MediaLookup, 110, 272
ModifyOrder event element, 89
ModifyOrderNotification event element, 89
MostRecentBusinessProcessStage servlet bean, 274
multisite catalog searching, 125

## N

NavHistoryCollector, 275

## O

Order Abandoned event element, 106
Order Approval gear
  components, 217
  configuring, 213
  implementation, 216
  implementation, display modes, 216
  implementation, gear modes, 216
  setting up, 209
  using, 210
order approval process
  ApprovalFormHandler, 190
  ApprovalRequiredDroplet, 189
  ApprovedDroplet, 191
  displaying approved and rejected orders, 191
  displaying orders requiring approval, 189
  implementing, 189
  processing approvals and rejections, 190
  servlet beans, 221
Order Changes event element, 89
order management servlet beans, 223
Order Status gear
  configuring, 204
  implementation, 206
  implementation, components, 206
  implementation, display modes, 206
  implementation, gear modes, 206
  setting up, 201
  using, 202
Order Submitted event element, 90
Order Where condition element, 94
OrderHolder, 145
OrderLookup, 148, 183, 208, 236, 277
orders
  abandoned. See abandoned order services
  adding items, 149
Orders Merged event element, 90
OrderStates, 207
OrderStatusFormHandler, 208
OrderStatusGear, 208

## P

parentCategory property
  using in catalogs, 109
Payment Group Changes event element, 90
PaymentGroupDroplet, 167, 281
PaymentGroupFormHandler, 177
PMDL rules
  creating with the discount rule editor (business users), 49
  examples (business users), 51
  using (business users), 49
portlets. See gears
PossibleValues, 285
price lists, managing in the ACC, 31
PriceDroplet, 285
PriceEachItem, 287
PriceEachItemDroplet, 287
PriceItem, 289
PriceItemDroplet, 289
PriceRangeDroplet, 290
pricing servlet beans, 224
product comparison
  comparison lists, managing, 137
  comparison lists, querying, 136
  described, 133
  example JSPs, 139
  implementing, 133
  localization, 137
  ProductList, 133
  ProductListContains, 136
  ProductListHandler, 137
  TableInfo, 135, 140
ProductBrowsed, 308
ProductList, 133
ProductListContains, 136, 291
  localization, 137

ProductListHandler, 137
ProductLookup, 110, 218, 240, 272
products
  creating, 17
  cross-selling, 95
  defined, 16
  editing, 20
  up-selling, 95
Promotion Revoked event element, 91, 92
PromotionDroplet, 293
promotions, 261
  behavior, 40
  coupons, 66
  creating (business users), 40
  creating scenarios for, 52
  delivering via URLs, 68
  described, 39
  disabling, 65
  discount rules, creating, 49
  discount rules, examples, 51
  media for, 65
  properties, 43
  servlet beans, 225
  types of promotions, 42
  withdrawing, 55
PromotionServlet, 68
purchase process servlet beans, 224

## R

Reanimate Abandoned Order action element, 107
ReanimateAbandonedOrderDroplet, 294
RelatedItemsOfCart slot, 95
RelatedItemsSlot scenario, 95
RemoveBusinessProcessStage, 295
RemoveItemFromGiftlist, 264
RepositoryValues, 285
RepriceOrder, 296
RepriceOrderDroplet, 182, 296
Revoke Promotion action element, 55, 95
root categories
  creating, 16
  defined, 16
  displaying, 112

## S

SaveOrderFormHandler, 183
Scenario Added an Item to an Order event element, 92
scenario elements
  action. See action elements
  condition. See condition elements
  event. See event elements
scenarios
  RelatedItemsSlot, 95
  using to cross-sell, 95
  using to up-sell, 95
Scheduled Order Event event element, 92
search components
  catalogs, 119
search criteria, specifying, 121
search results
  displaying, 125
  processing, 125
search types
  advanced, 123
  combining, 124
  configuring, 121
  hierarchical, 123
  keyword, 121
  text, 122
searches
  displaying results, 127
  in preview mode on CA, 131
  processing, 125
  restricting by catalog, 125
  restricting by site, 125
SearchFormHandler
  using with internationalized catalogs, 132
servlet beans
  AddBusinessProcessStageDroplet, 228
  AddItemToCartServlet, 226
  AdminOrderLookup, 277
  ApprovalRequiredDroplet, 189, 229
  ApprovalResolvedDroplet, 229
  Approved Droplet, 231
  ApprovedDroplet, 191
  AvailableShippingMethods, 233
  AvailableStoreCredits, 235
  BuyItemFromGiftlist, 264
  CartSharingFilterDroplet, 246
  CatalogNavHistoryCollector, 275
  CatalogPossibleValues, 243
  CategoryBrowsed, 308
  CategoryLookup, 240, 272
  ClosenessQualifierDroplet, 244
  ComplexPriceDroplet, 249
  ConvertAbandonedOrderDroplet, 250
  CostCenterDroplet, 251
  CouponDroplet, 254
  CurrencyCodeDroplet, 255
  DisplaySkuProperties, 256
  ExcludeItemInCartFilterDroplet, 246
  ExcludeItemsInCartFilterDroplet, 257
  ForEachItemInCatalog, 259
  GiftCertificateAmountAvailable, 263
  GiftitemLookupDroplet, 272
  GiftlistDroplet, 265
  GiftlistLookupDroplet, 272
  GiftListSiteFilterDroplet, 246
  GiftShippingGroups, 267
  HasBusinessProcessStageDroplet, 268
  InventoryFilterDroplet, 246
  InventoryLookup, 269
  IsGiftShippingGroup, 266
  IsHardGoods, 271
  list by functional area, 221
  MediaLookup, 272
  MostRecentBusinessProcessStageDroplet, 274
  OrderLookup, 236, 277
  PaymentGroupDroplet, 281
  PriceDroplet, 285

PriceEachItem, 287
PriceItem, 289
PriceRangeDroplet, 290
ProductBrowsed, 308
ProductFilterDroplet, 246
ProductListContains, 291
ProductLookup, 240, 272
PromotionDroplet, 293
ReanimateAbandonedOrderDroplet, 294
RemoveBusinessProcessStageDroplet, 295
RemoveItemFromGiftlist, 264
RepriceOrderDroplet, 296
SetLastUpdatedDroplet, 299
ShipItemRelPrice, 299
ShippableGroupsDroplet, 301
ShippingDroplet, 303
ShippingGroupDroplet, 304
SKULookup, 240, 272
UnitPriceDetailDroplet, 306
Set Order's Last Updated Date action element, 107
SetLastUpdatedDroplet, 299
ShipItemRelPrice, 299
ShippableGroupsDroplet, 301
Shipping Group Changes event element, 92
ShippingDroplet, 303
ShippingGroupDroplet, 155, 304
ShippingGroupFormHandler, 161
shopping carts
adding items, 149
adding payment information, 167
adding shipping information, 154
creating, 147
cross-selling, 95
handling custom commerce item properties, 154
implementing, 145
managing, 147
overriding the default commerce item type, 153
repricing, 182
retrieving, 147
saving, 183
up-selling, 95
shopping process tracking
adding stages, 228
checking stages, 268, 274
removing stages, 295
ShoppingCart, 145
SKU links, creating, 19
SKULookup, 240, 272
SKUs
creating, 18
creating bundled, 19
creating configurable, 18
defined, 16
editing, 20
SKUtLookup, 110
step pricing, 35
subcatalogs. See catalogs

T

TableInfo, 135, 140
templates
adding, 24
specifying, 24
text searches, 122
tiered pricing, 35

U

UnitPriceDetailDroplet, 306
UpdateInventory event element, 93
up-selling products, 95
Uses Promotion event element, 93

V

version conflict, preventing, 26
viewing existing cost centers, 69
ViewItemEventSender, 308
volume pricing, 35