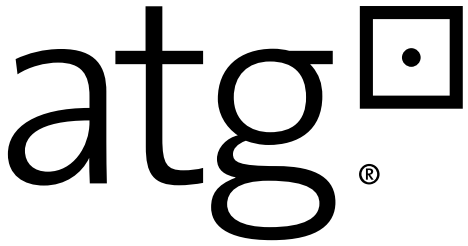


Version 10.0

ATG Business Commerce Reference Application Guide

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

ATG Business Commerce Reference Application Guide

Document Version

Doc10.0 BCOMMREFv1 12/21/10

Copyright

Trademark and Copyright Information

Patent Protection

No Warranty

Limitation of Liability

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

Contents

1 Introduction

The ATG Business Commerce Reference Application has three purposes:

- To demonstrate features and functionality of ATG Business Commerce that you can install and experiment with before you begin work on your own B2B site (or B2B and B2C site).
- To act as a template that you can adapt to build your own site.
- To be a guide on best practices in implementing a commerce site with ATG Business Commerce.

For the ATG Business Commerce Reference Application, we built a personalized business commerce site for a fictional company named Motorprise. We used the rich B2B functionality of ATG Business Commerce to provide a foundation and a guide to help you quickly and easily create your own personalized, innovative, commerce-intensive Web sites.

Motorprise is made up of pre-configured Java components and page templates that provide the core functionality you need to quickly implement a broad range of merchandising and order processing activities typical of an online-commerce site. We used the basic functionality available in ATG Business Commerce and extended it in places to implement specific features.

We demonstrate core features of ATG Business Commerce in the Motorprise site, including invoice/purchase order payment method, approvals, cost centers, and recurring and scheduled requisitions.

We designed Motorprise as an example of some of the ways that ATG Business Commerce can help you create a better and more effective e-commerce Web site. You will want to evaluate your business goals and use the ATG product features that are appropriate for your strategy. The processes and functionality that this manual discusses can be customized to fit the specific needs of your Web application.

Before exploring Motorprise, you should familiarize yourself with the core ATG Business Commerce concepts explained in the *ATG Commerce Programming Guide* and *ATG Commerce Guide to Setting Up a Store*.

The rest of this chapter provides you with the basic information you need to understand and get started with the Motorprise site.

Getting Started

Motorprise is part of the standard implementation of ATG Business Commerce. (If you chose a custom installation, Motorprise may not have been installed.) The standard ATG Business Commerce implementation also includes the ATG Control Center and SOLID Embedded Engine, the default database server used for Motorprise.

The instructions for installing the Motorprise site of the ATG Business Commerce Reference Application are provided in the *ATG Commerce Programming Guide*. For instructions on assembling an application that includes ATG Business Commerce, Motorprise, and the ATG Control Center, consult the *ATG Programming Guide*.

Setting Up E-mail

To configure the Motorprise site so that customers receive e-mails from Motorprise, you must set the e-mail host to a valid server and enable several scenarios.

First, specify the e-mail server:

1. In the Pages and Components > Components by path menu of the ACC, go to `/atg/dynamo/service/SMTPEmail`.
2. Set the `emailHandlerHostName` property to a valid mail server.

Second, enable the scenarios that send e-mail when an order is fulfilled and delivered.

1. Go to the Scenarios > Scenarios menu.
2. Select and enable the following Scenarios:

   DCS/Fulfillment

   DCS/ReceiveOrder

   B2B/25OverAverage

   B2B/ApprovalNotification

   B2B/NewUserRegistered

   B2B/OneMonth10Percent

Note: If your mail server checks to see if sender e-mail addresses are valid, you must change the sender mail address in all e-mail templates, which are located at `<ATG10dir>/MotorpriseJSP/j2ee-apps/Motorprise/web-app/en/email` and `<ATG10dir>/MotorpriseJSP/j2ee-apps/Motorprise/web-app/de/email`.

If you want to receive and read the messages sent by Motorprise, you should also change the e-mail addresses in the user profiles to use real e-mail addresses instead of `<user>@example.com`.

Viewing the Site

Once you assemble and deploy an application that includes the ATG platform, ATG Commerce and Motorprise, you can access it using the following URL:

```
http://hostname:port/Motorprise
```

The port you use depends on your application server configuration. For example, the default URL for JBoss is:

```
http://hostname:8080/Motorprise
```

For more information on application assembly and ATG modules, see the *ATG Programming Guide*. To learn about default ports, see the *ATG Installation and Configuration Guide*.

About this Guide

In this guide, we demonstrate the best practices for building a business commerce site such as Motorprise and explain how each feature is implemented. We have included code samples to help you understand the functionality and use it in your own projects by making small customizations. This guide also describes any additions or extensions made to the core Business Commerce functionality. The commerce areas of the site are built in a modular fashion, with a minimum of interdependencies—so that the site can be broken apart and re-implemented in pieces.

Page Developers, programmers, and business users should use this manual to see the features available to them and then make decisions about the most useful ways that they can customize these features for their site. This guide contains the following chapters.

- Business User's Overview
- Exploring the Motorprise Site as a Buying Organization
- Defining Motorprise User and Organization Profiles
- Company Administration
- My Account
- Displaying and Accessing the Product Catalog
- B2B Personalization
- Processing Orders
- SOAP Support
- Merchandising
- Creating ACC Users and Setting Privileges
- Exploring Motorprise as a Selling Organization
- Request-Handling Pipeline Servlets

2 Business User's Overview

The ATG Business Commerce Reference Application is a fictional business-to-business commerce Web site. Motorprise Inc. is a large distributor of automotive parts. The Motorprise site was designed to offer a full set of commerce features to its business customers and partners using ATG Business Commerce to enable easy catalog management, merchandising, profiling, content targeting, promotions, and analysis.

The Motorprise site includes:

- the ability to create organizational hierarchies, individuals, roles, and inheritance.
- an approval process that uses the ATG Business Commerce approval workflow.
- multiple customer-specific catalogs and price lists based on contracts.
- support for complex purchasing requirements such as volume pricing and order restrictions.
- purchase orders, requisitions, and credit cards as payment methods.
- the ability to use multiple payment methods, shipping addresses, and cost centers per line item.
- support for saved and scheduled orders.
- robust product searching and comparing.
- the ability for Motorprise customers to administer their users and approvers via a Web browser.
- Integration point via SOAP and XML.
- Scenarios and analysis.

For a technical description of how each of these features was implemented, refer to later chapters.

Setting Up the Catalog

The Motorprise site is targeted at its business partners, customers who purchase large volumes of automotive parts from Motorprise.

We set up our catalog:

- to easily show certain groups of products to certain customers.

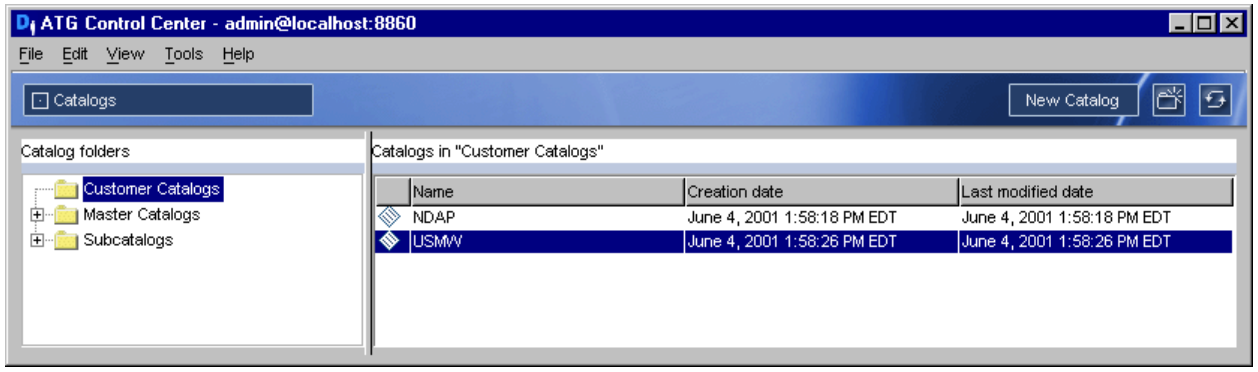
- to be able to share categories among customers so that we wouldn't have to create a separate catalog for each customer.
- to use the same categories to build our master catalog.
- to allow Motorprise product specialists to manage categories.

We wanted Motorprise customers to have catalogs that meet their specific business requirements. Not all customers can purchase all products that Motorprise sells. Motorprise negotiates contracts with its customers that specify the items in their catalog and their prices.

First, in the Catalog Management > Catalogs section of the ACC, we created subcatalogs for the different categories of stock that Motorprise sells.

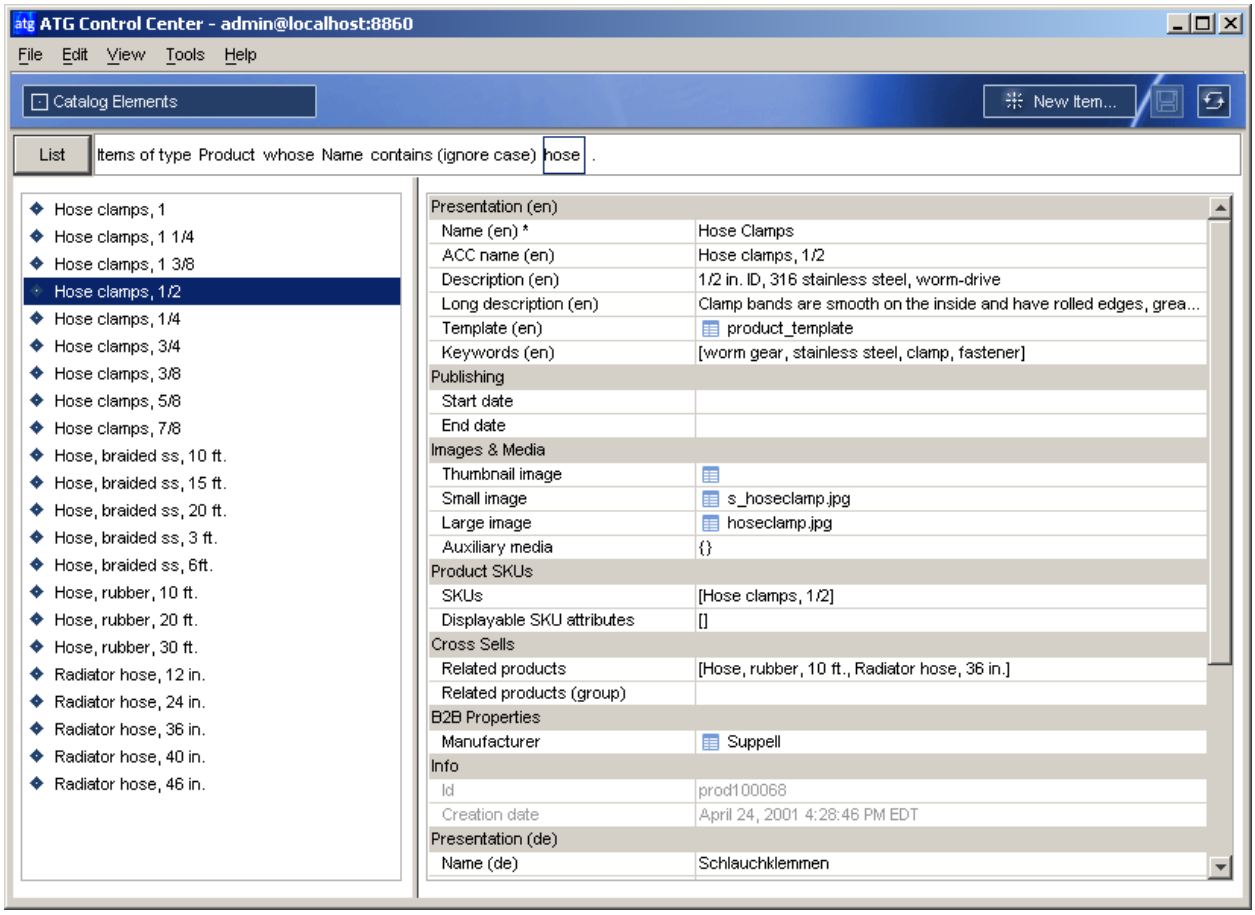
Then, we created catalogs for each customer built out of those subcatalogs. Finally, we created a base catalog that includes all of the subcatalogs. Anonymous users can also browse the site. The base catalog is the default; anonymous users, who are not assigned a catalog, see the base catalog.

These business needs are specific to Motorprise. You can create and manage a catalog structure that specifically fits your business strategy using ATG Business Commerce's flexible catalog management functionality.



Setting up the catalog structure in the ACC.

In the Catalog Management > Catalog Elements section of the ACC, you can view items by type, such as Category, Product, or SKU and browse their properties. You can also use the query editor to find an exact item more easily.



Browsing hose products in Catalog Management > Catalog Elements.

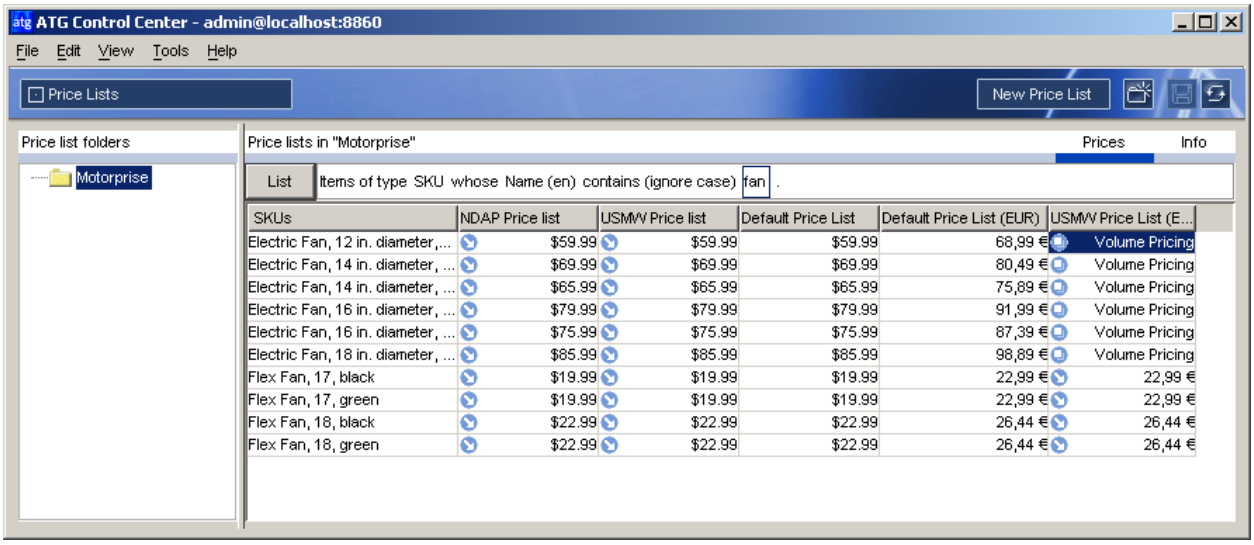
We extended the standard ATG Business Commerce functionality to create the Motorprise catalog. For more information, see the *ATG Commerce Programming Guide*.

Pricing

In Motorprise, we implemented price lists with volume pricing. Because Motorprise is a B2B site, its customers often place large orders and special pricing is often negotiated based on customer relationships.

We wanted to offer each Motorprise customer custom pricing. ATG Business Commerce provides the ability to set up custom price lists. We created a default price list to be used if there is no other price in a customer price list. We then set up custom price lists for our customers to manage their special pricing.

ATG Business Commerce provides two kinds of volume pricing: bulk pricing and tiered pricing. For bulk pricing, the price of a product is calculated based on the minimum quantity that is ordered. For tiered pricing, the price of a product is calculated using a fixed quantity or weight at different pricing levels.



Viewing Motorprise price lists .

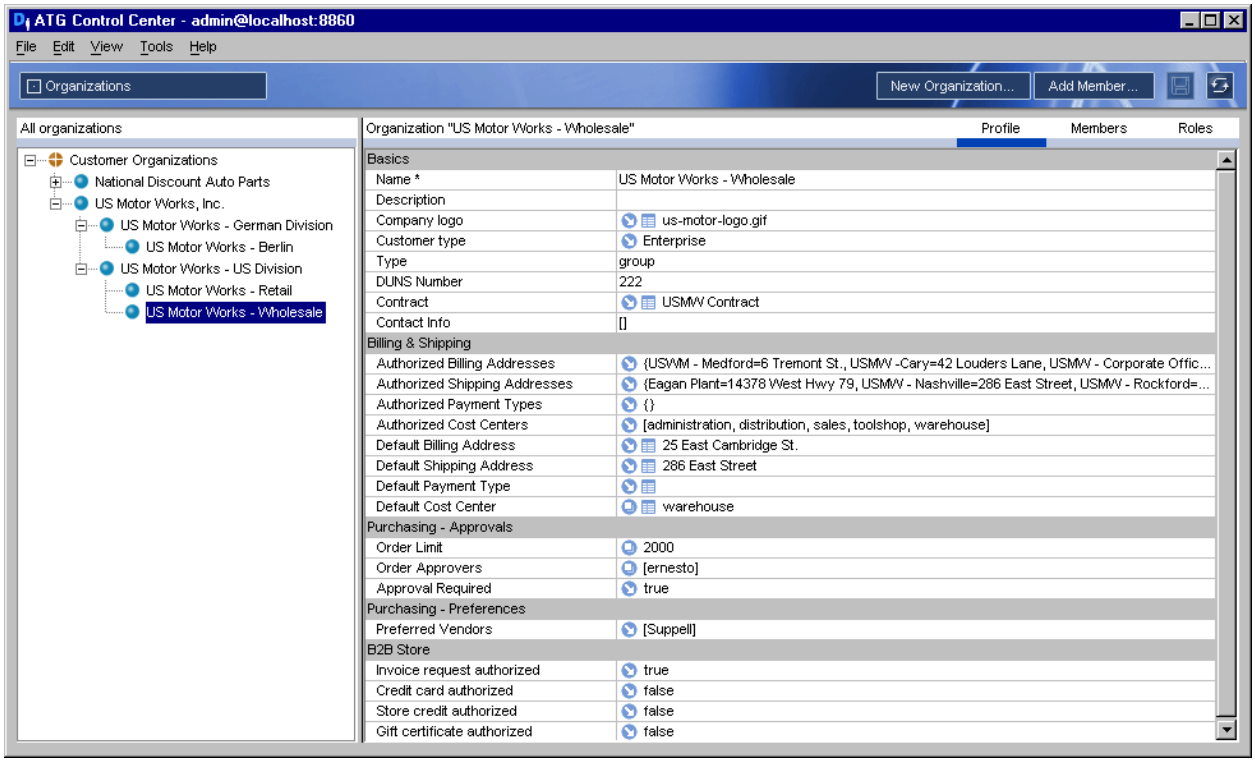
Creating Organizational Hierarchies, Profiles, and Roles

Because Motorprise is a business-to-business site, rather than business to consumer, it must track and record information not only about each individual user, but also about each customer organization and each of its subgroups.

We created multiple tiers of organizational structure for Motorprise:

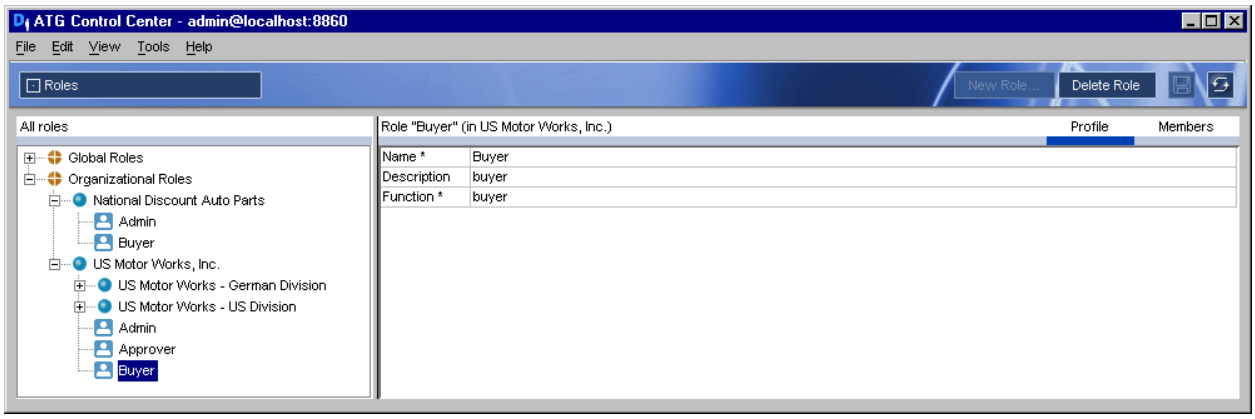
- Overall Company Organization
- Division/Business Unit
- Group (usually a department)
- Individual

The chapter Defining Motorprise User and Organization Profiles explains in detail how we did this.



Motorprise customer organizations in the ACC.

We also created various customer roles in Motorprise for buyers, approvers, and administrators. Individuals from buying organizations are assigned roles and are associated with a parent organization. They inherit various properties from that parent organization. You can view individual profiles in the People and Organizations > Users section in the ACC.



Viewing an individual's roles in the ACC.

Creating ACC Groups and Roles

We also created ACC groups and roles for Motorprise users; these are employees of Motorprise who have certain access privileges. For example, we created groups for Motorprise Account Managers; they have editing privileges to their customer's catalog and price list and read-only privileges for the base catalog and default price list.

Order Processing

We offered Motorprise customers a variety of options during the ordering process:

- the option to save orders for a later time.
- the ability to schedule a recurring order.
- choices of payment methods such as purchase orders, requisitions, and credit cards.
- support for multiple payment methods, shipping addresses, and cost centers per line item.

3 Exploring the Motorprise Site as a Buying Organization

Motorprise is a fictional distributor of automotive parts. The Motorprise Web site was created to serve its business customers, such as automotive manufacturers, retailers, and other distributors.

You can explore the Motorprise Web site several ways.

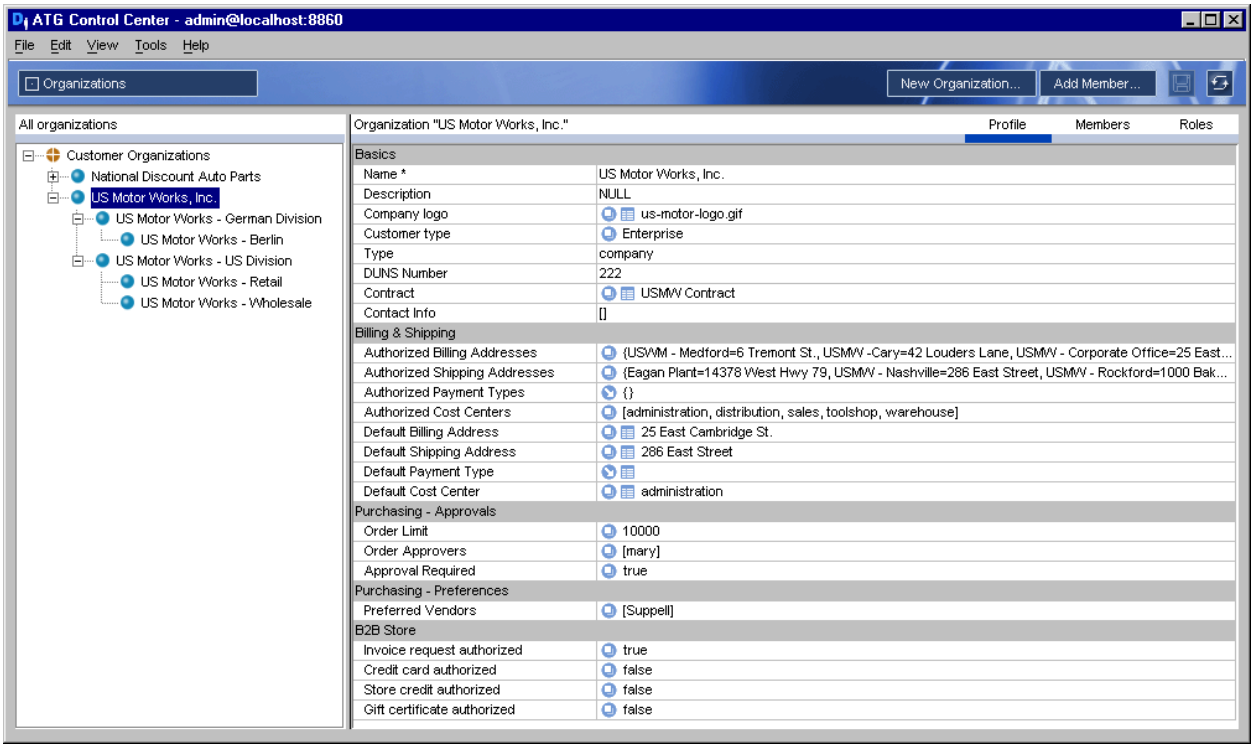
You can explore the Web site as a Motorprise customer. The Motorprise Reference Application comes with two sets of buying organizations and users. These business entities include different division levels and different roles such as buyers, approvers, and administrators. You can log in as any user and experience how the site is customized to his business needs.

You can log into the ACC as a developer and examine the JSPs and components we used to build the site, and view and modify the catalog, user and organizational profiles, scenarios, and reporting tools.

You also can explore Motorprise by logging in to the ACC as a Motorprise administrator. For more information, see the Exploring Motorprise as a Selling Organization chapter in this guide.

Motorprise Buying Organizations

We created two different types of buying organizations for the Motorprise Reference Application to demonstrate how ATG Business Commerce lets you build a B2B commerce site that can meet the needs of diverse customers. You can look at the organizational structures of the Motorprise customers in the ACC in the People and Organization > Roles section.



Motorprise organizational profiles in the ACC.

National Discount Auto Parts

National Discount Auto Parts (NDAP) is a national retail chain that sells automotive parts. Motorprise categorizes it as a preferred customer.

Buying profile

NDAP is cost conscious but more concerned about providing its customers with superior customer service. For example, they consider it a priority to have products in stock when needed.

NDAP Users

User	Organization	Role
Stuart Lee	National Discount Auto Parts	Administrator Buyer
Louis Veloso	National Discount Auto Parts	Buyer

US Motor Works, Inc.

US Motor Works, Inc. (USMW) is a large international retailer of automobile parts. USMW has global business concerns. Motorprise categorizes it as an enterprise customer.

Buying profile

USMW is considered by Motorprise to be one of its most strategic customers. The Motorprise site is designed to let USMW users see the actual amount of inventory in stock. Other customers only see inventory status.

USMW has multiple divisions. In this Reference Application, two divisions, US and Germany, are used as examples. The US division has Wholesale and Retail groups and the German division has a Berlin group.

NDAP Users

User	Organization	Role
Mary Granger	USMW, Inc. (corporate level)	Administrator Approver Buyer
Tim Hartwell	USMW, Inc. (corporate level)	Buyer
Ernesto Hernandez	US Division, Wholesale	Approver Buyer
Meredith Chin	US Division, Wholesale	Buyer
Ron Blooming	US Division, Wholesale	Buyer
Nicole Hsu	US Division, Wholesale	Buyer
Blair Parrish	US Division, Retail	Approver Buyer
Lorna Perman	US Division, Retail	Buyer
Peter Grün	US Motor Works – German Division	Admin Approver Buyer
Udo Bauer	US Motor Works – German Division US Motor Works – Berlin	Approver Buyer
Ingrid Hünä	US Motor Works – German Division US Motor Works – Berlin	Buyer

Exploring Motorprise as a Buyer

To log in to the Motorprise Web site, use the following URL:

```
http://hostname:port/Motorprise
```

The port you use depends on your application server and how it's configured. For example, by default the JBoss URL for Motorprise is accessible here:

```
http://hostname:8080/Motorprise
```

Log in as a Motorprise user, by entering the first name of any user as the username and password on the login page.

(Note: "admin" is not a valid username and password for Motorprise.)

Tim Hartwell is a registered customer of Motorprise. He is employed as a centralized buyer by US Motor Works at the corporate level.

Log in as Tim Hartwell, using "tim" for your username and password.

Tim's page is personalized and co-branded for him: the US Motor Works logo appears on the top right above his login information. Specials from USMW-designated preferred vendors are displayed on the right. These vendors are defined in USMW's organizational profile; a merchandising scenario (PreferredSupplierSpecials) highlights products from these vendors on the home page.

Tim also sees USMW's catalog, which has the following categories: Cooling System, Brakes, Ignition and Tune-Up, Tools, Exhaust, Air and Fuel, and Electrical. The promotions in the center of the page are targeted at him. He can also see the status of his open, fulfilled, and rejected orders.

Click on My Account.

Exploring My Account

The My Account page provides users with a self-service area. Here, Tim has a number of options. He can:

- check the status of his open and fulfilled orders.
- store and create purchase lists -- lists of frequently purchased products.
- schedule future orders – both one time and recurring.
- look at saved orders.
- edit his profile information.

Tim Hartwell's My Account page.

Viewing Orders

Click on the **Open orders** link. Then, click on the link to a specific order number.

Tim can view the details of his entire order. He can also cancel it from this page.

Click on the breadcrumbs My Account > Open Orders > Order ######## at the top of the page to return to the My Account page.

Tim can also review his fulfilled and rejected orders the same way.

Tim can also view and create scheduled orders, create purchase lists, and save orders from this page. These features are also available to him in other areas of the site such as when he is browsing the catalog and during checkout.

Editing Profile Information

Click on the **My profile** link to go to Tim's profile.

Tim's profile includes personal Contact Information, which is stored in his user profile, and Company Information, which is stored in the organization profiles of USMW and its business units. The company information includes attributes such as business units, cost centers, and shipping addresses. Users can inherit such profile attributes from their parent organization.

The Motorprise site allows Tim to edit his personal Contact Information or change his password. He can edit his Company information only to the extent of selecting defaults from lists of cost centers and shipping addresses. He is not a Company Admin so he cannot add or edit company information.

If you want to receive any e-mail intended for Tim, change Tim's e-mail address in his profile to your own. You can do this for any user.

Browsing the Catalog

Click on Product Catalog. This catalog has been custom-created for US Motor Works. The left panel shows the links to the seven subcatalogs in the USMW catalog. Click on the Air and Fuel category. Note the different subcategories available within Air and Fuel: filters, sensors, fuel pumps, PCV valves, and carburetors.

Click on **Fuel Pumps**, then click on the first Electric Fuel Pump, part #KAW-1761.

This page provides information on the product such as pricing, manufacturer, and description. The product availability status shows a simple "in stock" vs. "out of stock" and the stock level indicates the actual number of products available. Because USMW is an enterprise customer, Tim can actually see the stock level for this product. (If you log in as a user such as Stuart Lee from National Discount Auto Parts, a preferred customer, you will only see the stock availability status, not the actual quantity available.)

From the product page, Tim has a number of options:

- Add one or more of this product to his order
- Add one or more of this product to a purchase list
- Create a new purchase list
- Add this product to a comparison list
- View related items

Exploring Purchase Lists

Tim can add an item to one of his purchase lists or create a new one. Users can create purchase lists for groups of items that they order frequently instead of having to add each of those items from the product catalog every time they place an order.

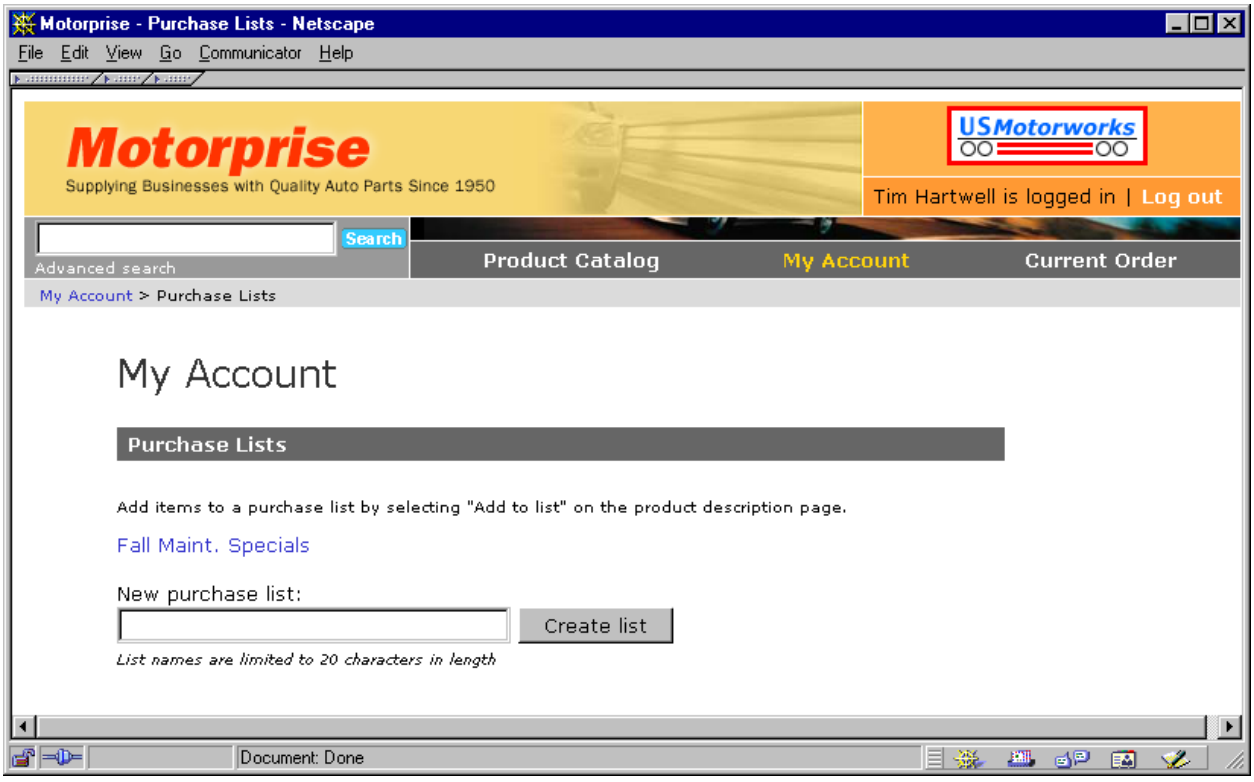
Adding an Item to a Purchase List

Tim can add any number of items to one of his purchase lists simply by selecting a previously created list from the pull-down menu and clicking the **Add to list** button. The item is automatically added and he can continue on in the product catalog.

Creating a Purchase List

Tim can create a new purchase list by clicking on the corresponding link, which brings him to the Purchase Lists page in My Account. Here Tim can add or modify purchase lists. Purchase lists persist over multiple user sessions so the buyer can add items to his current order over and over again.

To return to the product page for the fuel pump, you can hit the back button on your browser, or, use the breadcrumbs on the header: Product Catalog> Air and Fuel> Fuel Pumps> Electric Fuel Pump.



Managing purchase lists in My Account.(Navigation back to catalog is provided by "breadcrumbs."

Placing an Order

Under Add to Order, add 120 units of the KAW-1761 Fuel pump to Tim's order by entering 120 in the quantity field and clicking the **Add to order** button.

The Current Order page allows buyers to save an order to work on at a later time, change the quantity of items and update the order, or check out.

Click the **Checkout** button.

Shipping

The Shipping page allows buyers to select addresses or ship to multiple addresses. Tim is a centralized buyer making purchases for multiple retail locations. He can place a large single order and ship parts of it to multiple locations.

Click the **Ship to multiple addresses** link.

Split Tim's order by typing 60 in the "Quantity to move" field and selecting another address, US Motor Works – Nashville, from the dropdown list. The dropdown list displays the nicknames of the various shipping addresses. Then click the **Save** button. The order is now split between the two addresses. Click the **Continue** button.

Each shipping group has its own shipping method. Select a shipping method for each address using the dropdown list. Click the **Continue** button.

Billing

The Billing page is where users enter payment methods. As a USMW buyer, Tim is authorized to use purchase orders or requisitions (but not credit cards). He can use multiple payment methods for one order and split his order by dollar amount or by line item. Later, he will also be able to assign cost centers to parts of his order.

Click the **Split order by dollar amount** link. Enter a P.O. number and click **Add**. Note that Tim can change the billing address for this P.O. number if he wishes. Then add another P.O. number. After entering more than one P.O. number, users can specify one of them as the default payment method by selecting one in the dropdown list at the bottom of the page.

When you have finished entering P.O. numbers, click the **Continue** button. The full amount of the order has been assigned to the default P.O. Here, Tim can assign certain dollar amounts to each purchase order by entering an amount to move, selecting a P.O., and clicking **Save**.

Move a dollar amount to a different P.O. number and click **Save**. Then click the **Continue** button.

To facilitate accounting, Tim can also assign multiple cost centers to each line item in the order. This functionality works in the same way as the multiple payment methods. Select multiple cost centers if you wish and click the **Continue** button.

The order confirmation page allows Tim to create a recurring scheduled order, or simply execute the order immediately, or cancel the order. Click **Place order**.

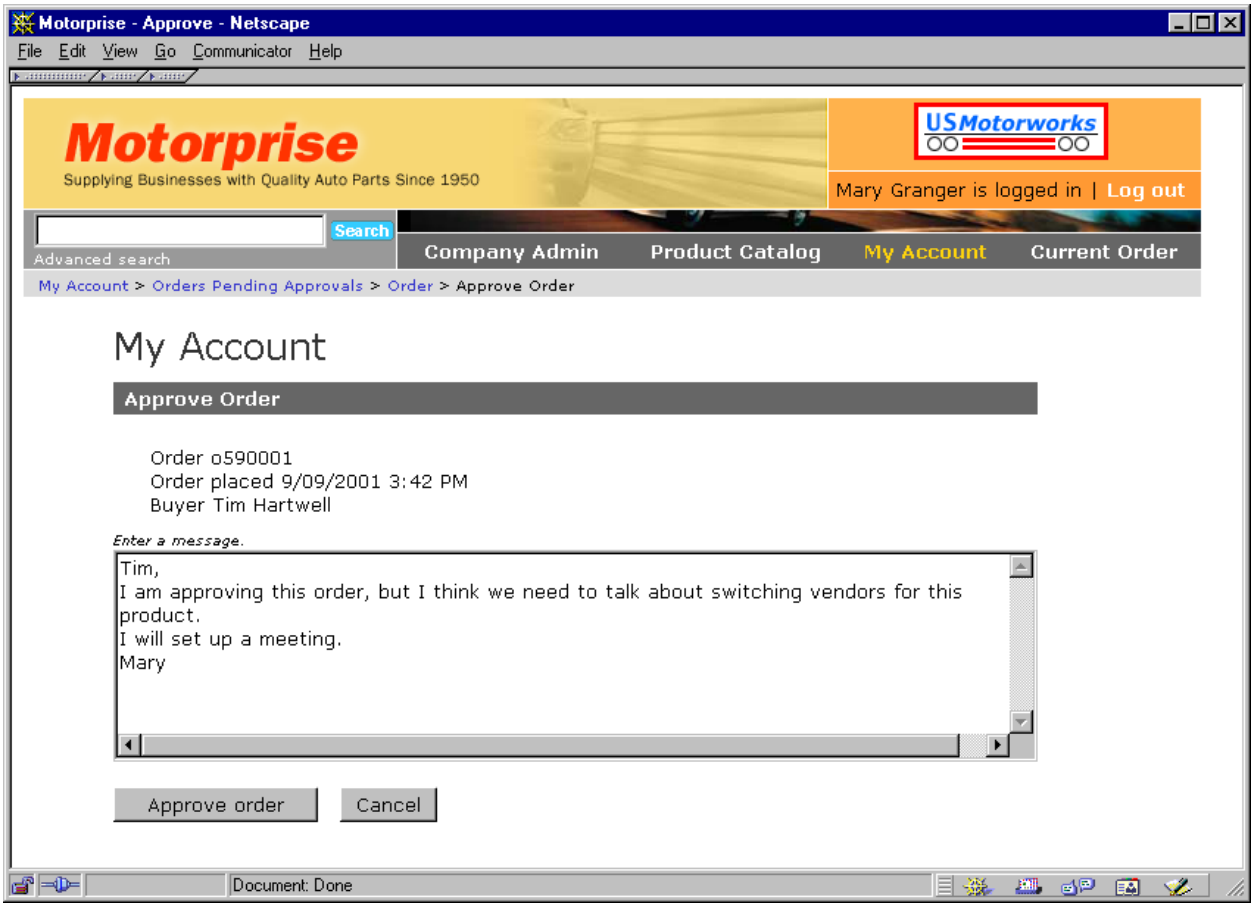
Tim has an order approval limit of $10,000. For all orders of larger amounts, he requires an order approval from his supervisor, Mary Granger. Tim's order has been submitted for approval. The ApprovalNotification scenario is set up to alert Mary by e-mail that an order requires her approval. After Mary receives this e-mail, she can log in to review the order.

Exploring Motorprise as an Approver

Click **Log out** and then log in as Mary Granger using "mary" for her username and password.

Mary is an approver and administrator for her company. (Note that she has a Company Admin area whereas Tim did not). She can see on her home page that there is an order that requires approval. Click on the **Approvals** link.

Mary is presented with all the orders needing her approval. Click on the order that Tim placed. Mary can review the pending order and approve or reject it. Click the **Approve order** link. Before she notifies Tim, she can add a message to the order.

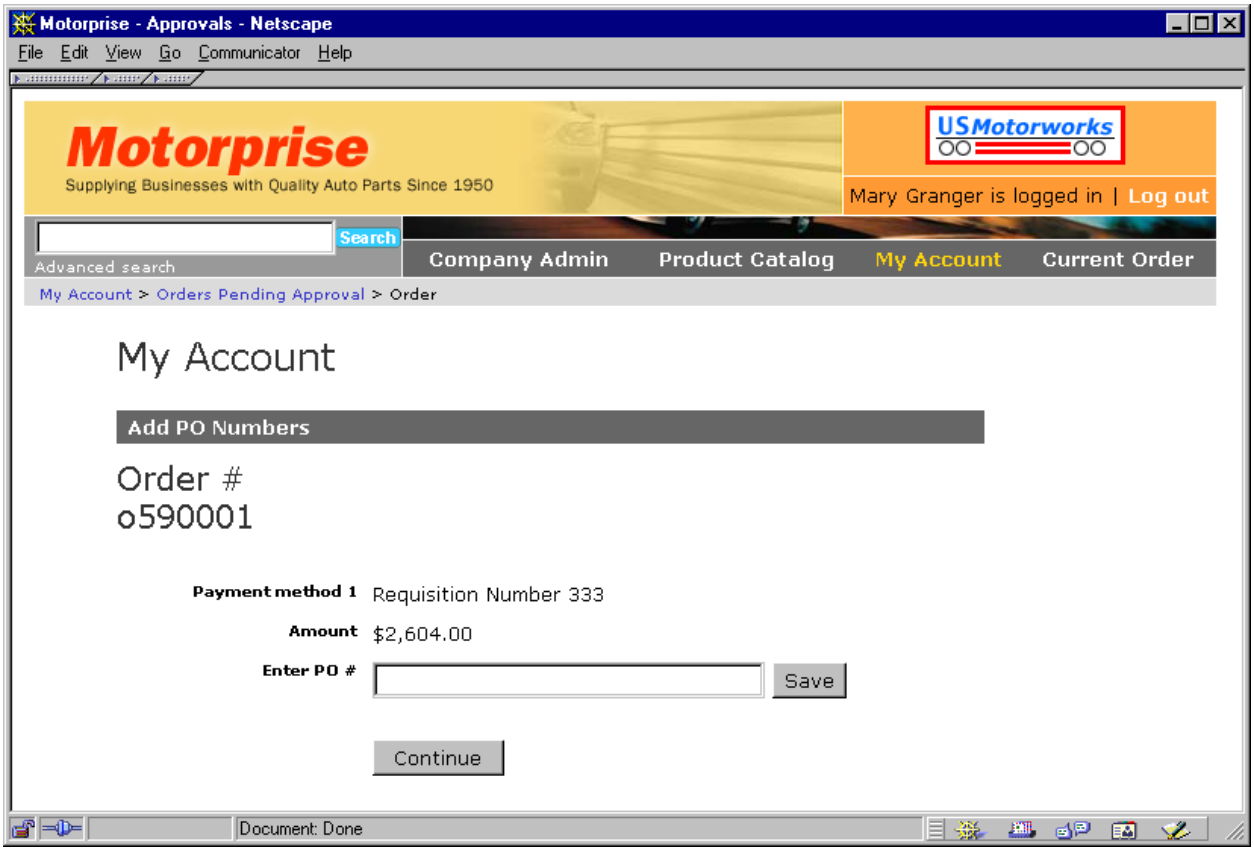


Approvers can enter messages when rejecting or approving orders.

Type in a comment to the order and click the **Approve order** button. When Mary approves the order, it is submitted to Motorprise for Fulfillment and another scenario is triggered that alerts Tim to the order's approval/rejection status.

Click on My Account. Note that because Mary is an approver, she can view her current and past approval information here.

If a user has submitted an order that requires approval, and that has a requisition number but no P.O. number, Mary will be prompted to add a P.O. number because Motorprise requires them for non-credit card orders.



If a buyer has submitted an order with only a requisition number for approval, the approver must enter a P.O. number.

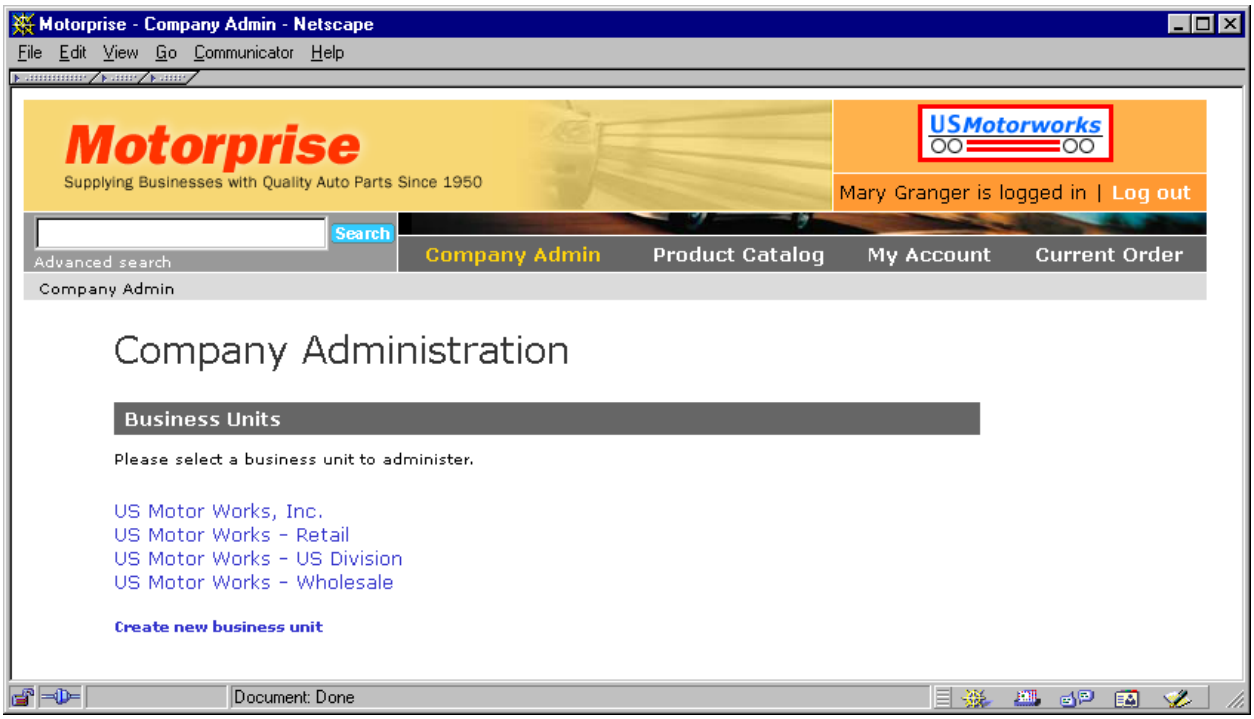
Exploring Motorprise as a Company Administrator

Motorprise allows buying organizations to administer their user and organizational profiles through a Company Admin JSP interface. This feature allows the buying organization to manage their own organizational information and roles through a Web browser, rather than relying on the supplying organization (in this case, Motorprise).

Because Mary is an administrator, she has access to a Company Admin area when she is logged into the Motorprise site. Click on the Company Admin section.

Listed on this page are all the business units for which Mary has administrator privileges. Because she is an admin at the corporate level, US Motor Works, Inc., she has access to the entire US division, and the groups that fall within its hierarchy. She can also create a new business unit from this page.

Select a business unit (for example, US Motor Works – Wholesale) from the list of business units.



Administrators select from their organization's business units on the Company Admin page.

Once she has selected a business unit, Mary can create and edit user profiles, and associate users with certain roles. She can also edit and add multiple user profiles at the same time. Admins can also edit business unit information such as addresses, payment methods, cost centers, and purchase limits.

Administrator options.

Creating Multiple Users

Click on the **Create Multiple Users** link. Enter the number of users you want to create and select "create users." Here, Mary can enter the individual information for each user, such as first, middle, and last name, login, password, and e-mail address. She can assign roles to each user from the available ones. In Motorprise, when business units are created, they have by default buyer, approver, and admin roles.

Click the **Save** button to create the new users. Click the **Cancel** button to return to the "Company Admin" page. The NewUser scenario (if enabled) then sends an e-mail to the new users with their user names and a link to the site.

Updating Multiple Users

Select the **Edit multiple users** link. Select the users you want to update. Here, Mary can select the attributes she wants to edit from the list and the values for each attribute. Click the **Change for selected users** button. From the confirmation screen, individual user profiles can be edited by clicking on their names.

Managing Shipping Addresses

Click on the **Shipping addresses** link.

The Shipping Address page in Company Admin allows to create, modify, and delete shipping addresses.

Create a new shipping address by clicking on the **Create new shipping address** link. Mary can enter information for the new shipping address and click the **Save** button.

She can edit an existing shipping address by clicking on the **Edit** link for the shipping address she wants to modify. Enter the new information and click the **Save** button. She can delete a shipping address by clicking on the **Delete** link for the shipping address you want to delete.

Managing Billing Addresses

Mary can create, edit, and delete billing addresses in the same way she does shipping addresses.

Managing Cost Centers

Click the **Cost centers** link.

Create a new cost center by selecting the **Add new cost center** link. Mary can enter information such as the ID number and the name of the cost center. Click the **Save** button. She can edit or delete an existing cost center by clicking on the appropriate link.

Approvals

Here Mary can say whether approvals are required by this business unit, and if so, what the purchase limit is.

Authorize Payment Methods

Select the **Authorize payment methods** link.

Mary can specify whether business units will be authorized to use invoices, credit cards, or both.

4 Defining Motorprise User and Organization Profiles

This chapter describes how we extended the standard user profile for Motorprise and includes the following sections:

Extending the Profile Repository
Explains how we extended the profile repository by adding Motorprise-specific properties to the `user` and `organization` item types.

Creating Table Definitions
Describes how we created the database schema to store the user profile data using a SQL script.

Extending the Profile Repository

This section describes the extensions we made to ATG Business Commerce and DPS for the Motorprise site. DPS performs the collection, storage, and retrieval of data specific to individual users of the Motorprise site. A profile is the data stored for an individual customer or organization based on information entered in the ACC or on actions by the company administrator or users on the Motorprise site. Collecting this data allows Web site developers, administrators, and business users the ability to show customers personalized products and content based on their profile information.

For Motorprise, we used personalization in a variety of ways to make a compelling B2B experience. DPS provides basic personalization capability, and ATG Business Commerce extends this functionality with B2B-specific properties. You can easily tailor this functionality to fit the needs of your Web application. Motorprise demonstrates several ways to extend this functionality.

Please refer to the *Setting up a Profile Repository* chapter in the *ATG Personalization Programming Guide* and the *SQL Content Repositories* chapter in the *ATG Repository Guide* for more detailed information about SQL Repository definitions.

DPS stores customer data in a repository, which consists of `user` and `organization` item types and other supporting item types. The default DPS repository contains item types with a set of properties that are needed by most personalized Web applications, such as name and address information. In ATG Business Commerce, additional properties are added to the `user` and `organization` item types to make possible many B2B features. On top of B2B Commerce additional properties have been added at Motorprise level as explained below.

Adding Motorprise Properties

For Motorprise, we added properties to the `user` and `organization` item types to provide functionality that is specific to its business model. These extensions are defined in `<ATG10dir>/Motorprise/config/atg/userprofiling/userProfile.xml`.

Many of the properties in the Motorprise `organization` and `user` repository items are derived from other properties. A derived property enables one repository item to obtain property values from another repository item or from another property in the same repository item. For more information on derived properties, please refer to the *Derived Properties* section of the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Programming Guide*.

For example, the following XML code in `<ATG10dir>/Motorprise/config/atg/userprofiling/userProfile.xml` illustrates how we used derived properties for the property `invoiceRequestAuthorized` in Motorprise.

Some Motorprise users are authorized to use purchase orders and request invoices, rather than using a corporate credit card. We added certain properties in Motorprise to check if a user or organization has that authorization.

First, we created a derived property, `invoiceRequestAuthorized`, in `organization` item-descriptor.

```
<item-descriptor name="organization">
    <table name="b2b_org_info" type="auxiliary" id-column-name="org_id">
      <property category-resource="categoryB2BStore"
        name="myInvoiceRequestAuthorized"  data-type="boolean"
        column-name="invoice_auth"
        display-name-resource="myInvoiceRequestAuthorized" expert="true">
      </property>
    </table>
    <!-- Derived properties -->
    <property category-resource="categoryB2BStore" name="invoiceRequestAuthorized"
      data-type="boolean" display-name-resource="invoiceRequestAuthorized">
      <derivation override-property="myInvoiceRequestAuthorized">
        <expression>parentOrganization.invoiceRequestAuthorized</expression>
      </derivation>
    </property>
</item-descriptor>
```

In Motorprise, all organizations have a Boolean property, `invoiceRequestAuthorized`, that indicates whether users from that organization are authorized to use purchase orders. Every organization checks for this property. If the property is not present, then the organization derives the value from its parent organization. As shown in the code sample above, we created an extra property `myInvoiceRequestAuthorized` in `organization` that contains the value. Then we created the required property, `invoiceRequestAuthorized`, that actually checks for the value in `myInvoiceRequestAuthorized` using the `<derivation override-property="myInvoiceRequestAuthorized">` tag. If this value is null, then we get the value from the

parent organization using the tag
<expression>parentOrganization.invoiceRequestAuthorized</expression>.

Based on the invoiceRequestAuthorized property of organization, we created another derived property in the user item-descriptor as shown in the below code:

```
<item-descriptor name="user">
    <table name="b2b_auth_pmnt" type="auxiliary" id-column-name="id">
      <property category-resource="categoryB2BStore"
        name="myInvoiceRequestAuthorized" data-type="boolean"
        column-name="invoice_auth" expert="true"
        display-name-resource="myInvoiceRequestAuthorized">
      </property>
    </table>
    <!--Derived Property -->
    <property category-resource="categoryB2BStore" name="invoiceRequestAuthorized"
      data-type="boolean" display-name-resource="invoiceRequestAuthorized">
      <derivation override-property="myInvoiceRequestAuthorized">
        <expression>parentOrganization.invoiceRequestAuthorized</expression>
      </derivation>
    </property>
</item-descriptor>
```

As shown above, when invoiceRequestAuthorized is retrieved for any user, we first check its value at the user level. If it is null, then we retrieve the value from the parent organization. This allows us to have most users inherit the setting of their parent organization; at the same time, we can override the default behavior for specific users when needed.

We created other derived properties in Motorprise as indicated below.

These are the properties we added to the organization item type:

Property	Description
companyLogo	A media item used to display the company logo.
myCompanyLogo	This property overrides the companyLogo property. It is used when the business unit's logo is not the same as the parent organization's logo.
invoiceRequestAuthorized	Indicates whether the user is authorized to use invoices as a payment method. The checkout pages use this property of the profile to determine whether to display the option to pay for an order with invoices.
creditCardAUthorized	Indicates whether a user is authorized to use credit cards as a payment option.

giftCertificateAuthorized	Indicates whether a user is authorized to use gift certificates as a payment option.
storeCreditAuthorized	Indicates whether a user is authorized to use store credit as a payment option.
myInvoiceRequestAuthorized	This is an override property for the invoiceRequestAuthorized property. If a user tries to access the invoiceRequestAuthorized property and there is no value in the myInvoiceRequestAuthorized property for the user type, this property is accessed.
myCreditCardAuthorized	This property works the same as myInvoiceRequestAuthorized.
myGiftCertificateAuthorized	This property works the same as myInvoiceRequestAuthorized.
myStoreCreditAuthorized	This property works the same as myInvoiceRequestAuthorized.

We also added the following properties to the user item type:

Property	Description
currentOrganization	Used in the Company Admin section of the site. When an administrator from a buying organization (such as Mary Granger from USMW) logs in, she selects a business unit and this property is set. This property references an organization repository item. For other users, this property is not set.
currentUser	A reference to a user repository item. It is used in the admin pages to work with multiple users. This property keeps track of which user is being updated at a given time. For example, if administrator Mary is editing the properties for user Stuart, then the currentUser property would get set to Stuart.
numOfOrders	Indicates the current number of orders placed by the user. This property is used in the 25OverAverage scenario. See the Merchandising chapter in this guide for more information.
avgOrderAmt	Tracks the average total of all the user's orders. This property is used in the 25OverAverage scenario. See the Merchandising chapter in this guide for more information.
invoiceRequestAuthorized	Indicates whether the user is authorized to use invoices as a payment method. The checkout pages use this property of the profile to determine whether to display the option to pay for an order with invoices.

creditCardAUthorized	Indicates whether a user is authorized to use credit cards as a payment option.
giftCertificateAuthorized	Indicates whether a user is authorized to use gift certificates as a payment option.
storeCreditAuthorized	Indicates whether a user is authorized to use store credit as a payment option.
myInvoiceRequestAuthorized	This is an override property for the invoiceRequestAuthorized property. If a user tries to access the invoiceRequestAuthorized property and there is no value in the myInvoiceRequestAuthorized property, the parent organization's property is accessed.
myCreditCardAuthorized	This property works the same as myInvoiceRequestAuthorized.
myGiftCertificateAuthorized	This property works the same as myInvoiceRequestAuthorized.
myStoreCreditAuthorized	This property works the same as myInvoiceRequestAuthorized.
currentLocation	This property is set to the page that a user requests. It is used to track which page the user is currently requesting.

For more information on how we set the transient currentLocation property, see the section on the SetCurrentLocation servlet in the Request-Handling Pipeline Servlets chapter.

For more information on profile properties, refer to the *Working with the Dynamo User Directory* chapter of the *ATG Personalization Programming Guide*.

Creating Table Definitions

The <ATG10dir>/Motorprise/config/atg/userprofiling/userProfile.xml file is combined with the userProfile.xml files from ATG Business Commerce, DSS, and DPS. These files are combined per the rules of XML combination to produce one XML file that is then parsed and used to describe the item types in the repository. (For more information about XML file combination, see the *XML File Combination* section of the *Nucleus: Organizing JavaBean Components* chapter in the *ATG Programming Guide*.) The combined file is reparsed each time the Profile Adapter Repository starts and it is never written out to disk, so we only need to maintain the separate files, not the combined one.

This approach allows users to define the user profile in a layered structure For example, the <ATG10dir>/Motorprise/config/atg/userprofiling/userProfile.xml defines only those properties, which are specific to Motorprise. It "adds on" to the already existing user profile definition.

The underlying storage of the user profile repository is a relational database. The SQL Profile Repository is used to expose that data via the Repository API. In the XML file, we describe the mapping of repository items to the relational database tables and columns.

We used the following SQL script to create the database schema to store the user profile data:

<ATG10dir>Motorprise/sql/db_components/solid/b2b_user_orddet_ddl.sql

```
-----------------------------------------------------
-- Profile extensions of B2BStore
--
-- The use_org_XXX columns are not used in the first release
-- of Dynamo 5.5 but are present to allow for planned changes
-- in the data model of the B2BStore solution set in a future
-- release By adding the columns now we can update the data
-- model without requiring any database migration.
--
-----------------------------------------------------
CREATE TABLE b2b_user_info (
  id        VARCHAR(40)     NOT NULL references dps_user(id),
     num_orders     INTEGER,
     avg_order_amt     DOUBLE PRECISION,
     use_org_approver     NUMERIC(1) CHECK(use_org_approver in (0,1)),
     use_org_costctr      NUMERIC(1) CHECK(use_org_costctr in (0,1)),
     use_org_billaddr     NUMERIC(1) CHECK(use_org_billaddr in (0,1)),
     use_org_shipaddr     NUMERIC(1) CHECK(use_org_shipaddr in (0,1)),
     use_org_payment      NUMERIC(1) CHECK(use_org_payment in (0,1)),
     use_org_vendors      NUMERIC(1) CHECK(use_org_vendors in (0,1)),
     use_org_purchlst     NUMERIC(1) CHECK(use_org_purchlst in (0,1)),
        PRIMARY KEY(id)
);
--
-- The table for the organization which contains logo image, and
-- authorization flags for credit cards, invoices, store credit
-- and gift certificates.
--
-- The use_prnt_XXX columns are not used in the first release
-- of Dynamo 5.5 but are present to allow for planned changes
-- in the data model of the B2BStore solution set in a future
-- release By adding the columns now we can update the data
-- model without requiring any database migration.
--
CREATE TABLE b2b_org_info (
  org_id       VARCHAR(40)     NOT NULL references dps_organization(org_id),
  logo            VARCHAR(40)     ,
  cc_auth               NUMERIC(1)            CHECK(cc_auth in (0,1)),
  invoice_auth          NUMERIC(1)            CHECK(invoice_auth in (0,1)),
  store_crdt_auth       NUMERIC(1)            CHECK(store_crdt_auth in (0,1)),
  gift_crt_auth         NUMERIC(1)            CHECK(gift_crt_auth in (0,1)),
```

```
  use_prnt_approver      NUMERIC(1) CHECK(use_prnt_approver in (0,1)),
  use_prnt_costctr       NUMERIC(1) CHECK(use_prnt_costctr in (0,1)),
  use_prnt_billaddr      NUMERIC(1) CHECK(use_prnt_billaddr in (0,1)),
  use_prnt_shipaddr      NUMERIC(1) CHECK(use_prnt_shipaddr in (0,1)),
  use_prnt_payment       NUMERIC(1) CHECK(use_prnt_payment in (0,1)),
  use_prnt_vendors       NUMERIC(1) CHECK(use_prnt_vendors in (0,1)),
  use_prnt_purchlst      NUMERIC(1) CHECK(use_prnt_purchlst in (0,1)),
  PRIMARY KEY (org_id)
);

COMMIT WORK;

GRANT SELECT ON b2b_user_info TO PUBLIC;
GRANT SELECT ON b2b_org_info TO PUBLIC;

ALTER TABLE b2b_user_info SET PESSIMISTIC;
ALTER TABLE b2b_org_info SET PESSIMISTIC;

COMMIT WORK;
```

After the profiles have been extended, an ACC user can set up profiles for customer organizations. Then, the administrator for that customer organization can create and edit users and suborganizations for that customer organization, using the Company Admin pages on the Motorprise web site.

5 Company Administration

The Motorprise Web site includes a self-service Company Admin interface that lets administrators in buying organizations access their accounts remotely.

Motorprise customers can use the Company Admin section of the site to manage their user and organizational profiles. Administrators can create and edit suborganizations and groups, associate users with certain user types, and edit user profiles. They can also add or update multiple user profiles at the same time. In essence, the Company Admin interface allows Motorprise customers to manage and control all information and access privileges for their company's users.

This chapter describes how we created the Company Admin section of the Motorprise site for buying organizations and includes the following sections:

Verifying Admin Access
Explains how a user is given access to the admin pages.

Creating Business Units and Roles
Explains how to create business units and their corresponding roles.

Selecting a Business Unit
Explains how admins select a business unit to administer.

Registering Multiple Users
Explains how to create multiple users simultaneously.

Editing Multiple User Profiles
Explains how common properties of multiple users can be edited simultaneously.

Deleting Users
Explains how users are deleted.

Using the B2BRepositoryFormHandler
Explains how we used the `B2BRepositoryFormHandler` to let administrators manage shipping and billing addresses, payment methods, cost centers, and purchase limits.

Changing the Scope of Form Handlers
Explains how to change the scope of `MultiUserAddFormHandler` and `MultiUserUpdateFormHandler`.

Verifying Admin Access

In Motorprise, only users who have admin rights within their organization can view and use the Company Admin pages. These users were assigned an admin role when they were created by a Motorprise administrator. When a user logs into Motorprise, we check to see if he or she has admin rights to the parent organization.

We used the `atg/userdirectory/droplet/HasFunction` component in the page fragment `MotorpriseJSP/j2ee-apps/motorprise/web-app/en/common/BrandNav.jsp` to check if the user who logged in is an admin. This fragment is used on `MotorpriseJSP/j2ee-apps/motorprise/web-app/en/home.jsp`.

Two input parameters are passed to `HasFunction`. It takes the ID of the user who is logged in as `userId`. It also checks to see if a `function` of `admin` exists for that user and renders a `true` output if it does.

Here is the code fragment from `BrandNav.jsp`:

```
<!--  display link if user has admin role -->
<dsp:droplet name="HasFunction">
  <dsp:param bean="Profile.id" name="userId"/>
  <dsp:param name="function" value="admin"/>
  <dsp:oparam name="true">
    <dsp:droplet name="Switch">
      <dsp:param bean="Profile.currentLocation" name="value"/>
      <dsp:oparam name="admin">
        <td align="center"><dsp:a href="../admin/business_units.jsp">
        <b><font color="#FDD30E" size=-1>Company
          Admin</font></b></dsp:a></td>
      </dsp:oparam>
      <dsp:oparam name="default">
        <td align="center"><dsp:a href="../admin/business_units.jsp">
        <b><font color="#FFFFFF" size=-1>Company
          Admin</font></b></dsp:a></td>
      </dsp:oparam>
    </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>
```

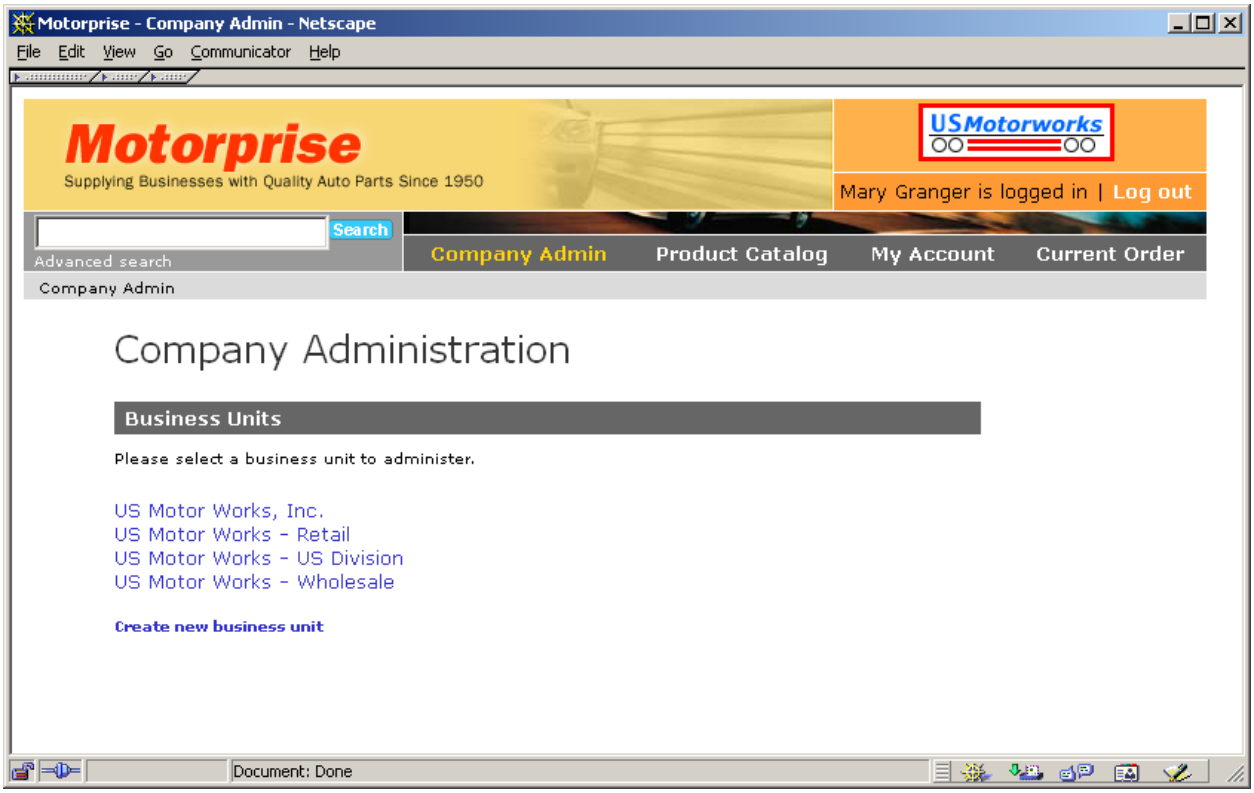
If the user has a `function` of `admin`, the "Company Admin" link is displayed on the home page, thereby giving him or her access to the admin pages. The following screen shot shows a home page for Mary Granger from USMW with a company admin link:

Company administrators see the Company Admin tab when they log in.

If the user is not an admin, the "Company Admin" link is not displayed and the user does not have access to the Company Admin pages. The following screen shot shows a home page for Ron Blooming from USMW without a company admin link:

Users who do not have admin privileges do not see the Company Admin tab when they log in.

After an administrator logs in with a valid user name and password and selects Company Admin, the Business Units page is displayed. This page lists all of the business units that the user has access to administer. He or she can either select one of these units or create a new business unit.



From the Company Administration page, Mary Granger of USMW can select a business unit or create a new one.

Creating Business Units and Roles

In Motorprise, Company admins create business units and roles for their company. We used the `atg/projects/b2bstore/userdirectory/CreateOrganizationFormHandler` to:

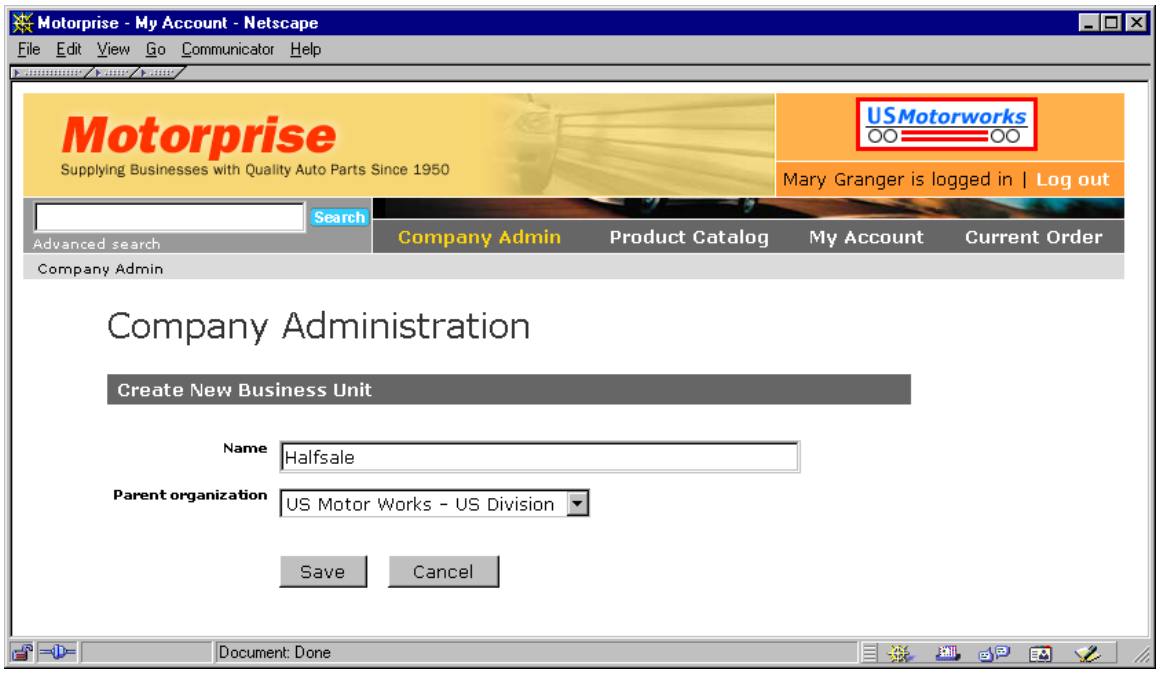
- create a new business unit
- set its name and parent organization
- create and assign roles, such as buyer, approver, and admin, for that business unit
- assign an admin role for that business unit to the user who creates it

In Motorprise, when an admin creates a new business unit, `CreateOrganizationFormHandler` does two things:

- creates relative roles of buyer, approver, and admin for that business unit
- assigns the role of admin to the user creating the business unit

Creating New Business Units

In the Company Admin section, the business units page has a **Create new business unit** link to `business_unit_new.jsp`. Clicking this link brings the administrator to the Create New Business Unit page.



Creating a new business unit.

Here, the administrator can enter the name of the new business unit and select an existing business unit as its parent organization. For example, Mary Granger could create a business unit called Halfsale with the US Motor Works – US Division as a parent.

We used the `CreateOrganizationFormHandler` on `<ATG10dir>/MotorpriseJSP/j2ee-apps/motorprise/web-app/en/admin/business_unit_new.jsp` to create the new business unit and to specify its name and parent organization.

We set the following properties on the profile of the new business unit:

`UserId`	The id of the current user. This user is assigned an admin role to the newly created organization.
`organizationName`	The name of the business unit that is going to be created.
`parentOrganizationId`	The id of the parent organization for the newly created business unit.

This is the form we used on business_unit_new.jsp. We created a hidden form field that sets the CreateOrganizationFormHandler.userId to the admin who accesses the page. Then we set the name of the organization, CreateOrganizationFormHandler.organizationName.

```
<dsp:form action="business_units.jsp" method="post">
<dsp:input bean="CreateOrganizationFormHandler.userId"
   beanvalue="Profile.id" type="hidden"/>
<table border=0 cellpadding=4 cellspacing=0>
   <tr valign=top>
      <td align=right><span class=smallb>Name</span></td>
      <td><dsp:input bean="CreateOrganizationFormHandler.organizationName"
         size="30" type="text"/></td>
   </tr>
```

Then we used TargetPrincipals to show the user a list of the organizations from which she or he can select a parent of the new business unit to set parentOrganizationId

```
   <tr valign=top>
      <td align=right><span class=smallb>Parent organization</td>
      <td>
      <dsp:select bean=
         "CreateOrganizationFormHandler.parentOrganizationId">
      <!--By default set the parent organization to current user's
          organization--!>
      <dsp:getvalueof id="parentId" idtype="java.lang.String"
         bean="Profile.parentOrganization.repositoryId">
      <dsp:option selected="<%=true%>" value="<%=parentId%>"/>Select
         Parent Organization
      </dsp:getvalueof>
      <!--Display all organizations available to this user--!>
      <dsp:droplet name="TargetPrincipals">
         <dsp:param bean="Profile.id" name="userId"/>
         <dsp:param name="roleName" value="admin"/>
         <dsp:oparam name="output">
         <dsp:droplet name="ForEach">
            <dsp:param name="array" param="principals"/>
            <dsp:param name="sortProperties" value="+name"/>
            <dsp:oparam name="output">
            <dsp:getvalueof id="parentId" idtype="java.lang.String"
               param="element.repositoryItem.repositoryId">
            <dsp:option value="<%=parentId%>"/>
            <dsp:valueof param="element.repositoryItem.name"/>
            </dsp:getvalueof>
            </dsp:oparam>
         </dsp:droplet>
         </dsp:oparam>
      </dsp:droplet>
```

```
      </dsp:select>
      </td>
   </tr>
```

Then we added **Save** and **Cancel** buttons.

```
   <tr valign=top>
      <td></td>
      <td><br>
      <dsp:input bean=
         "CreateOrganizationFormHandler.createOrganizationSuccessURL"
         type="hidden" value="business_units.jsp"/>

      <dsp:input bean=
         "CreateOrganizationFormHandler.createOrganizationErrorURL"
            type="hidden" value="business_unit_new.jsp"/>
      <dsp:input bean="CreateOrganizationFormHandler.createOrganization"
         type="submit" value=" Save "/>  
      <input type="submit" value=" Cancel "></td>

   </tr>

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

Setting Organizational Roles

`CreateOrganizationFormHandler` has the following properties, which are used to set roles in the new business unit:

`createRelativeRoles`	This property is set to true by default so that relative roles are created for the new organization.
`assignRelativeRoles`	This property is set to true by default so that the list of relative roles specified in `assignableFunctionNames` is assigned to a user.
`functionNames`	Sets the names of the relative roles of the new business unit For example, in Motorprise, these roles are admin, approver, buyer.
`assignableFunctionNames`	Specifies the relative roles for the user who is creating the new business unit. In Motorprise, the user is assigned an admin role for that business unit.
`userId`	The ID of the user to whom the relative roles listed in `assignableFunctionNames` is assigned.

The `functionNames` property of the `CreateOrganizationFormHandler` is used to set the names of the roles of the new organization. In Motorprise, these roles are admin, approver and buyer. This means that when a company admin creates a new business unit in Motorprise, three relative roles are created with these names:

- **Buyer:** a member of the buying organization who makes purchases using the application
- **Approver:** member of buying organization who can authorize another person's purchase
- **Admin:** member of the buying organization who can create and modify other users of the store

These roles fit the business model for Motorprise. You could create different roles for your business. See the section on `CreateOrganizationFormHandler` for further information.

The `assignableFunctionNames` property is a list of roles to assign the user specified in the `userId` property. These functions are used to extract a relative role from an organization and then assign it to the user. In this case, the user is assigned the role of `admin`.

Selecting a Business Unit

When an administrator such as Mary Granger of USMW logs in and selects the Company Admin pages, she is brought to `business_units.jsp` where a list of all the business units that she is authorized to administer is displayed. Each business unit in the list is a link; clicking on any of these links brings the user to `company_admin.jsp` for that business unit. On that page, the admin sees a list of various options for administration.

The Company Administration page for the USMW - Wholesale business unit.

Any specific action performed thereafter is related to that business unit.

We used the `atg/userdirectory/droplet/TargetPrincipals` component on `company_admin.jsp` to determine the organization that the user selected in `business_units.jsp`. When the user selects the link of the business unit, the ID of that organization is passed as a parameter. `TargetPrincipals` retrieves an array of organizations for which the user is assigned an admin role. We compare the ID of each one to the ID of the organization that was passed in as a parameter. When the correct match is found, that organization is assigned to the `Profile.currentOrganization` property.

This snippet from `company_admin.jsp` illustrates how we used `TargetPrincipals`:

```
<%-- Set the Profile.currentOrganization property to the organization
which was selected in the business_units.jsp page --%>

<dsp:droplet name="IsEmpty">
  <dsp:param name="value" param="organizationId"/>
  <dsp:oparam name="false">
    <dsp:droplet name="TargetPrincipals">
      <dsp:param bean="Profile.id" name="userId"/>
      <dsp:param name="roleName" value="admin"/>
```

```
      <dsp:oparam name="output">
        <dsp:droplet name="ForEach">
        <dsp:param name="array" param="principals"/>
        <dsp:oparam name="output">
          <dsp:droplet name="Compare">
            <dsp:param name="obj1" param="organizationId"/>
              <dsp:param name="obj2" param=
               "element.repositoryItem.repositoryid"/>
            <dsp:oparam name="equal">
            <dsp:setvalue bean="Profile.currentOrganization"
              paramvalue="element.repositoryItem"/>

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

          </dsp:oparam>
        </dsp:droplet>  <%-- End of ForEach --%>
      </dsp:oparam>
    </dsp:droplet>  <%-- End of TargetPricipals --%>
  </dsp:oparam>
</dsp:droplet> <%-- End of IsEmpty --%>
```

Registering Multiple Users

We wanted administrators to be able to register a number of users with common organizational information simultaneously. For example, an admin could create five users who are all buyers, from the US Motor Works – Wholesale group.

These users all inherit various properties of the parent organization such as shipping addresses, billing addresses, and cost centers.

Inheriting Properties

When users are created, by default they are given the selected business unit as their parent organization. Users, like suborganizations, can inherit properties from parent organizations. Some properties in the user profile are inherited from this parent organization by default if they are not set at the user level or at the suborganization level. Some of these properties include shipping address, billing addresses, credit cards, cost centers, and purchase limits.

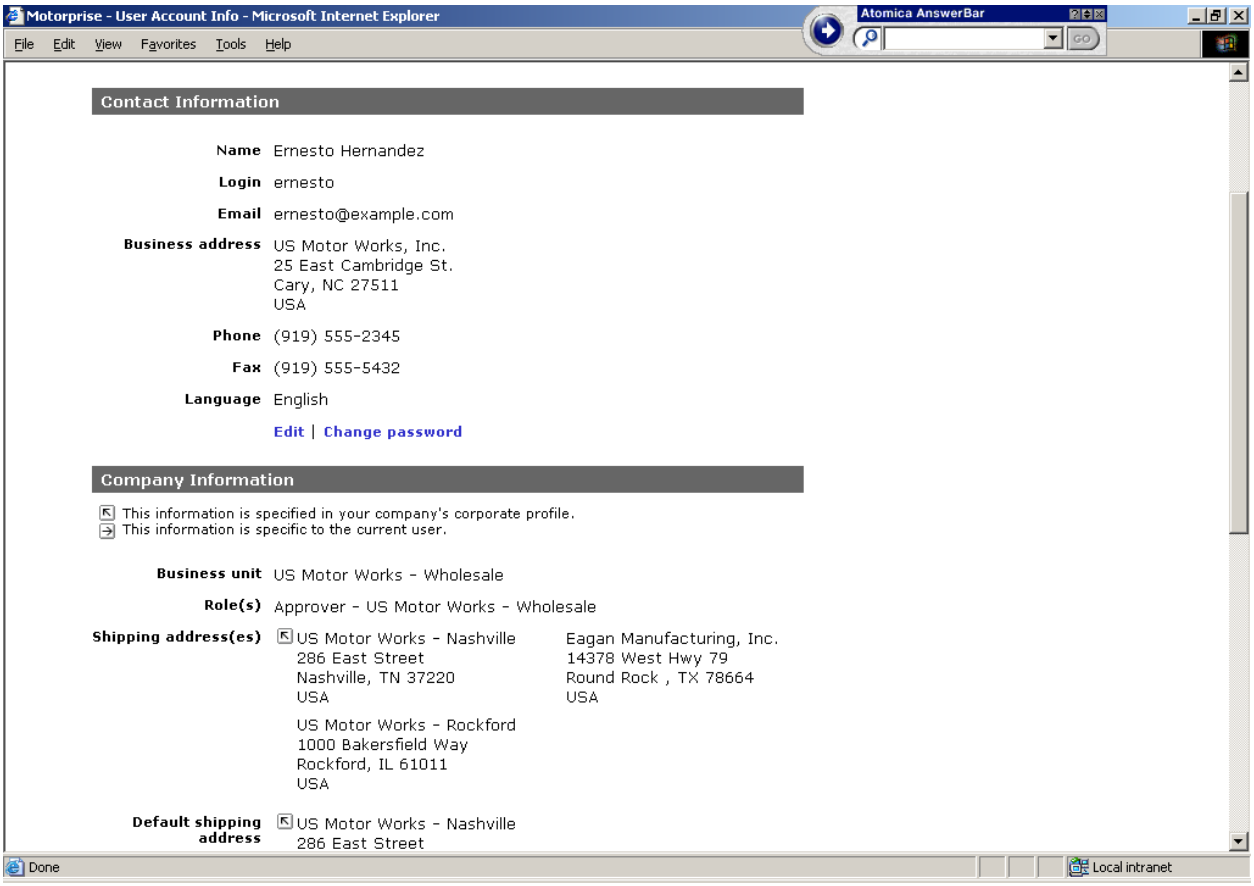
When you view a user's profile or the information of the selected business unit on the Motorprise site, you see the following symbols next to various fields:

 This information is specified in your company's corporate profile.

In this case, the property is not set at the current user or suborganization level; it is being inherited from the parent organization. This is true if the property being displayed does not have any value at the current user or suborganization level.

This information is specific to the current user.

In this case, the property has been set in the current user and is not being inherited from the parent organization. This is true if the property being displayed has a value at the current user or suborganization level. Users can later modify their profiles, thus overriding the properties set at the parent organization level. When they do, the symbol reflects that change.



Properties inherited from the parent organization as displayed in a Motorprise user profile.

After these users are created, they each receive an e-mail with their username.

Creating New Users

The following is a list of properties that must be entered when creating new users:

- First Name

- Last Name
- Login Id
- Password
- Confirm Password

The following properties are optional:

- Role(s)
- Email
- Language (defaults to English)

In Motorprise, the following properties are inherited from the parent organization when the user is created:

- invoiceRequestAuthorized
- creditCardAuthorized
- giftCertificateAuthorized
- storeCreditAuthorized

We used the `MultiUserAddFormHandler` component to create the group registration pages. The `MultiUserAddFormHandler` allows us to create new users, assign them to an organization, and assign roles to them.

Once the admin chooses the business unit, he or she clicks the Create Multiple Users link to start adding multiple users. First, the admin must enter the number of users he or she wants to create. The relevant code of this page, `create_multiple_users.jsp`, is as follows:

```
<dsp:setvalue bean="MultiUserAddFormHandler.clear" value=""/>

<table border=0 cellpadding=0 cellspacing=0 width=800>
    <tr valign=top>
    <td width=55><dsp:img src="../images/d.gif" hspace="27"/></td>

    <%--  main content area --%>
    <td valign="top" width=745>
    <dsp:form action="create_users2.jsp" method="post">
    <input type="hidden" name="b2bOp" value="add">
...

    <tr>
        <td>Create <dsp:input bean="MultiUserAddFormHandler.count"
          maxlength="1"
          size="1" type="text" value=""/> users.
        <p>
        <input type="submit" value="Create users">
```

```
        </td>
      </tr>
```

As shown in the above code, the admin enters the number of users to create and we initialize `MultiUserAddFormHandler.count` with it. We also use this variable in the next page, looping through it to provide the input fields to enter this number of users.

Once the admin submits the above page (`create_multiple_users.jsp`), he or she is directed to `create_users2.jsp` to enter information of all the users to create. We used the `MultiUserAddFormHandler` to create a user array based on the `MultiUserAddFormHandler.count` variable. The code below shows how we loop through `MultiUserAddFormHandler.count`, accessing each user in the `MultiUserAddFormHandler.users` array to provide input fields to set each user's properties:

```
<dsp:form action="create_multiple_users.jsp" method="post">

  <dsp:input bean="MultiUserAddFormHandler.confirmPassword" type="hidden"
    value="true"/>
  <dsp:input bean="MultiUserAddFormHandler.createErrorURL" type="hidden"
    value="create_users2.jsp"/>
  <dsp:input bean="MultiUserAddFormHandler.createSuccessURL" type="hidden"
    value="create_users3.jsp"/>
  <dsp:input bean="MultiUserAddFormHandler.organizationId" beanvalue=
    "Profile.currentOrganization.repositoryid" type="hidden"/>
  <dsp:input bean="MultiUserAddFormHandler.value.member" type="hidden"
    value="true"/>

  <table border=0 cellpadding=4 width=100%>
    <tr><td><dsp:img src="../images/d.gif" vspace="0"/></td></tr>
    <tr valign=top>
      <td colspan=2><span class=big>Company Administration</span><br><span
        class=little></span></td>
    </tr>
    <tr><td><dsp:img src="../images/d.gif" vspace="0"/></td></tr>

    <tr valign=top>
      <td colspan=2>
      <table width=100% cellpadding=3 cellspacing=0 border=0>
        <tr><td class=box-top> Create New Users</td></tr></table>
      </td>
    </tr>
    <tr><td><dsp:img src="../images/d.gif" vspace="0"/></td></tr>

     <!--Display all the users--!>
     <dsp:droplet name="/atg/dynamo/droplet/For">
      <dsp:param bean="MultiUserAddFormHandler.count" name="howMany"/>
       <dsp:oparam name="output">
         <tr>
```

```
  <td align=right><span class=smallb>User <dsp:valueof
    param="count"/>
    </span></td>
</tr>
<tr>
  <td align=right><span class=smallb>Name</span></td>
  <td width=75%><dsp:input bean=
    "MultiUserAddFormHandler.users[param:index].value.firstName"
    size="15" type="text" required="<%=true%>"/>
   <dsp:input bean=
    "MultiUserAddFormHandler.users[param:index].value.middleName"
       size="4" type="text"/>
   <dsp:input bean=
     "MultiUserAddFormHandler.users[param:index].value.lastName"
       size="15" type="text" required="<%=true%>"/></td>
</tr>
<tr>
  <td align=right><span class=smallb>Login</span></td>
  <td><dsp:input bean=
    "MultiUserAddFormHandler.users[param:index].value.login"
     size="30"
      type="text"/></td>
</tr>
<tr>
  <td align=right><span class=smallb>Password</span></td>
  <td><dsp:input bean=
    "MultiUserAddFormHandler.users[param:index].value.Password"
     size="30"
      type="password" value=""/></td>
</tr>
<tr>
  <td align=right><span class=smallb>Confirm</span></td>
  <td>
    <dsp:input bean=
      "MultiUserAddFormHandler.users[param:index].
       value.CONFIRMPASSWORD"
       size="30" type="password" value=""/>
  </td>
</tr>
<tr>
  <td align=right><span class=smallb>Email</span></td>
  <td><dsp:input bean=
   "MultiUserAddFormHandler.users[param:index].value.email"
     size="30"
      type="text"/></td>
</tr>
<tr valign=top>
  <td align=right><span class=smallb>Role</span></td>
  <td>
```

```
<%/* Check if the roleIds already exist.  This property will not
 be set when the page is displayed for the first time.  In this
 case, display all the roles as unchecked boxes.  If this property
 is set, that means this page has been displayed already and that
 an error has occured.  In this case, each role in the
 currentOrganization is traversed to see if any of their ids exist
 in the roleIds property and if so, display that role as a checked
 box. */ %>

<dsp:droplet name="IsEmpty">
<dsp:param bean=
  "MultiUserAddFormHandler.users[param:index].roleIds"
  name="value"/>
<dsp:oparam name="true">

<%/* List organization roles, allowing admin to check off
     roles for each new user */%>
   <dsp:droplet name="ForEach">
   <dsp:param bean=
    "Profile.currentOrganization.relativeRoles"
     name="array"/>
   <dsp:param name="elementName" value="role"/>
   <dsp:param name="indexName" value="roleIndex"/>
   <dsp:oparam name="output">

      <dsp:input bean=
      "MultiUserAddFormHandler.users[param:index].roleIds"
        paramvalue="role.repositoryId" type="checkbox" />
        <dsp:valueof param="role.name">No name
      </dsp:valueof>
   </BR>
   </dsp:oparam>
   </dsp:droplet>
</dsp:oparam>

<dsp:oparam name="false">

   <dsp:droplet name="ForEach">
   <dsp:param bean="Profile.currentOrganization.
    relativeRoles"
     name="array"/>
   <dsp:param name="elementName" value="role"/>
   <dsp:param name="indexName" value="roleIndex"/>
   <dsp:oparam name="output">

      <dsp:droplet name="ArrayIncludesValue">
      <dsp:param bean=
      "MultiUserAddFormHandler.users[param:index].roleIds"
        name="array"/>
      <dsp:param name="value" param="role.repositoryId"/>
```

```
                        <dsp:oparam name="true">
                          <dsp:input bean=
                        "MultiUserAddFormHandler.users[param:index].roleIds"
                            paramvalue="role.repositoryId" type="checkbox"
                            checked="<%=true%>"/>  <dsp:valueof param=
                            "role.name">No
                            name</dsp:valueof>
                        </dsp:oparam>
                        <dsp:oparam name="false">
                             <dsp:input bean=
                        "MultiUserAddFormHandler.users[param:index].roleIds"
                           paramvalue="role.repositoryId" type="checkbox" />
                        <dsp:valueof param="role.name">No name</dsp:valueof>
                        </dsp:oparam>
                        </dsp:droplet>
                      </br>
                      </dsp:oparam>
                      </dsp:droplet>

                      </dsp:oparam>
                      </dsp:droplet>
              </td>
            </tr>
            <tr>
              <td align=right><span class=smallb>Language</span></td>
              <td><dsp:select bean=
                "MultiUserAddFormHandler.users[param:index].value.locale">
                    <dsp:option value="en_US"/>English
                    <dsp:option value="de_DE"/> German
                  </dsp:select></td>
            </tr>

            <tr>
              <td colspan=2><hr size=1 color="#666666"></td>
            </tr>
            <tr><td><dsp:img src="../images/d.gif" vspace="6"/></td></tr>
          </dsp:oparam>
        </dsp:droplet>
        <tr>
          <td></td>
          <td><b><dsp:input bean="MultiUserAddFormHandler.create"
            type="submit"
            value=" Save "/>  
                   <input type="submit" value=" Cancel " ></td>
        </tr>
        <%--  End of add new user action --%>

    </table>

    </dsp:form>
```

Setting Single-Value Properties

The code above shows how we set a property value that is a single value in the user profile, such as a first name. We used the format `MultiUserAddFormHandler.users[param:index].value.propertyName`. Here, `index` is a numeric value that indicates the current user in the array of users being created. For example, `MultiUserAddFormHandler.users[0].value.firstName` sets the `firstName` of the first user.

For example, if the `MultiUserAddFormHandler.count` property is set to 10, then the `users[ ]` array has 10 entries and we can set the properties for 10 users during the same form submission.

The `create` operation is invoked by a `submit` button that submits to `MultiUserAddFormHandler.create`. Administrators can also use the group registration pages to register a single user.

Setting Properties That are Arrays, Lists, or Maps of Other Items

If a property is an array, a list, or a map of other items, we must set it differently, because we cannot assign a repository item from a Web page. To set properties that are arrays, lists, or maps of other items, we used an ID to refer to an item that already exists. We set the special sub-property called `repositoryIds` of the property.

For example, the `roles` property of `user`is a map, so we set it like this:

```
<dsp:input type="checkbox" value="role0001"
   bean="MultiUserAddFormHandler.users[0].value.roles.repositoryIds"/>
<dsp:input type="checkbox" value="role0002"
   bean="MultiUserAddFormHandler.users[0].value.roles.repositoryIds"/>
```

In the above code, repository items of type `role` with IDs `role001` and `role002` exist in the repository. We set these role IDs as the `repositoryIds` sub-property of the `roles` property of the `user`, which is the map, and the form handler automatically retrieves the corresponding repository items and adds them to the map.

Editing Multiple User Profiles

Administrators can update properties for multiple buyers using the Edit Multiple Users pages. This is useful if you want to set a profile property of multiple users with the same value. For example, you could set the purchase limit for 20 users to $5000. Using the Edit Multiple Users features, you could do that with a single form submission.

We used the `atg/userprofiling/MultiUserUpdateFormHandler` component for group updates in Motorprise. This component requires the repository IDs of the users whose profiles are being updated. They are set in the `repositoryids` property.

For more information on the `MultiUserUpdateFormHandler`, see the API.

Profile properties are updated using the `MultiUserUpdateFormHandler.value` dictionary. The values in this dictionary are applied to all the users whose user IDs are included in the `repositoryIds` property. For example, to update `orderPriceLimit` for all the users whose repository IDs are set in `MultiUserUpdateFormHandler.repositoryIds`, the following format can be used: `MultiUserUpdateFormHandler.value.orderPriceLimit`.

For arrays, sets, lists, and maps of other items, we set them by using an ID to refer to an item that already exists. We do this by setting the special sub-property called `repositoryIds` of the property. For example, to set the roles of the user, which is a map, we use input tags like this:

```
<dsp:input type="checkbox" bean="MultiProfileUpdateFormHandler.
  value.roles.repositoryIds" value="r001"/>
```

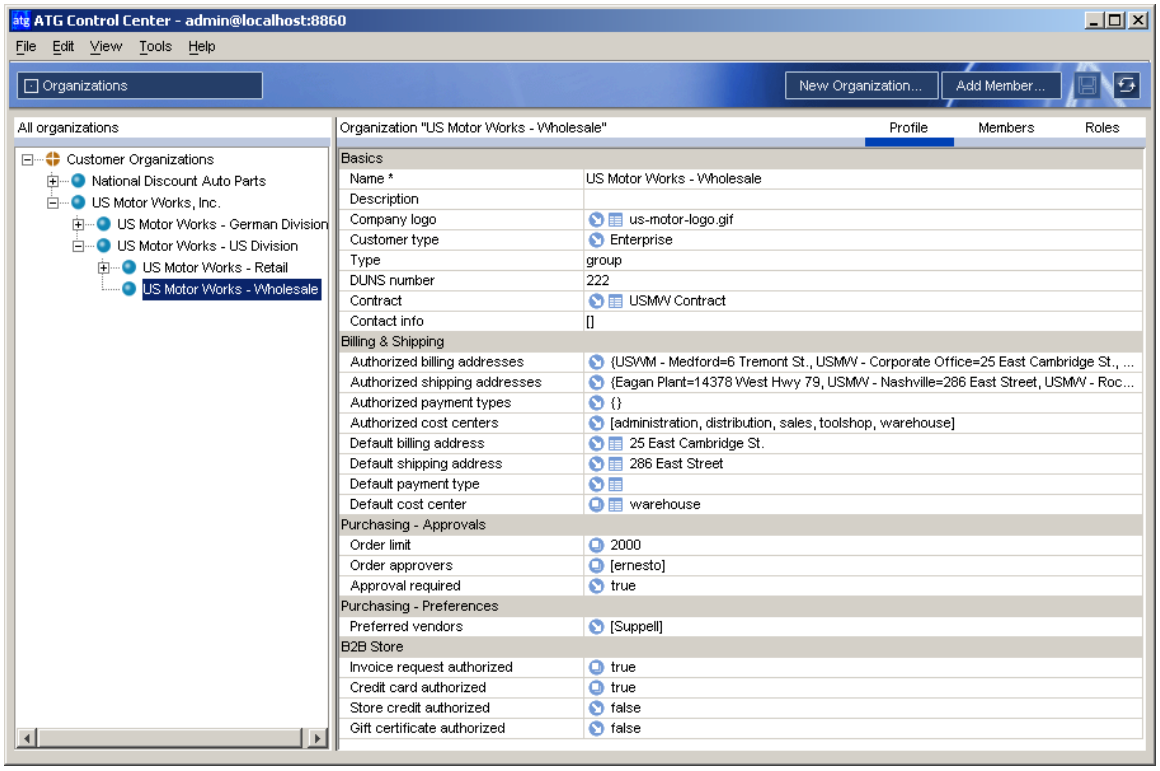
The update operation is called using a submit button that submits to `MultiProfileUpdateFormHandler.update`.

The code below demonstrates how to set the `OrderPriceLimit` property and the roles of users whose user IDs are `Id001` and `Id002`:

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

<dsp:form action="xxx" method="post">
   <dsp:input bean="MultiUserUpdateFormHandler.repositoryIds"
      type="checkbox" value="Id001"/>
   <dsp:input bean="MultiUserUpdateFormHandler.repositoryIds"
      type="checkbox" value="Id002"/>
   <dsp:input bean="MultiUserUpdateFormHandler.value.OrderPriceLimit" type="text"
      value=""/>
   <dsp:input bean="MultiUserUpdateFormHandler.value.roles.repositoryIds"
      name="Admin" type="checkbox" value="R001"/>
   <dsp:input bean="MultiUserUpdateFormHandler.value.roles.repositoryIds"
      name="Buyer" type="checkbox" value="R002"/>
   <dsp:input bean="MultiUserUpdateFormHandler.update" type="submit"
      value="submit"/>
</dsp:form>
</dsp:page>
```

Deleting Users

Because Motorprise users can have roles such as approvers or admins, they may be referenced in the profiles of their parent organizations. For example, Ernesto Hernandez is listed as an approver in the profile of USMW - Wholesale.



Ernesto Hernandez is listed as an order approver in the USMW - Wholesale profile.

When users are deleted, all such references to them must also be deleted. Failing to remove these references when deleting the users would result in referential integrity constraint violations.

To avoid this, we created a new class, `atg.projects.b2bstore.userprofiling.B2BDeleteFormHandler`, that extends `MultiProfileUpdateFormHandler` to override the behavior when user profiles are deleted. We remove any references to the user item in `B2BDeleteFormHandler` before handing it over to the base class for removing the user item. Using the `B2BDeleteFormHandler` is no different than using `MultiProfileUpdateFormHandler`. In Motorprise, users can be deleted from the Company Administration page using the Delete Users link (`users_delete.jsp`). We display a list of all the users with checkboxes and a Delete button to delete the selected users.

The following describes the code snippets from `users_delete.jsp`:

First, the repository ID of the organization is passed to the `atg/userdirectory/droplet/UserList` component to get the list of users for the current organizations. The ID of the admin who is accessing the page is passed into `excludeId` so that the admin is excluded from the returned list of users. This ensures that the admin may not delete his or her own account while logged into it:

```
<dsp:droplet name="UserList">
   <dsp:param bean="Profile.currentOrganization.repositoryid"
      name="organizationId"/>
```

```
        <dsp:param bean="Profile.id" name="excludeId"/>
           <dsp:oparam name="output">
```

Next, we set the name of the user repository item in the form handler to delete:

```
        <dsp:form action="users_delete_confirm.jsp" method="post">
           <dsp:input bean="DeleteProfileFormHandler.itemDescriptorName"
              type="hidden" value="user"/>
```

We used the Range droplet to get 10 items each time and display each user:

```
<dsp:droplet name="Range">
   <dsp:param name="array" param="users"/>
   <dsp:param name="start" param="startIndex"/>
   <dsp:param name="howMany" value="10"/>
   <dsp:param name="sortProperties" value="+name"/>
```

Next, we display a checkbox. When the admin selects this box on the page, the repository id of the corresponding user is added to `DeleteProfileFormHandler.repositoryIds` so that this user item is deleted from the repository:

```
<dsp:oparam name="output">
<dsp:input bean="DeleteProfileFormHandler.repositoryIds"
   paramvalue="element.repositoryItem.id" type="checkbox"/>
   <dsp:valueof param="element.repositoryItem.firstName"/> <dsp:valueof
      param="element.repositoryItem.lastName"/><BR>
</dsp:oparam>

<dsp:oparam name="outputEnd">
   <dsp:droplet name="IsEmpty">
     <dsp:param name="value" param="nextHowMany"/>
     <dsp:oparam name="false"><BR>
       <dsp:a href="users_delete.jsp">  Next 10 <dsp:param name="startIndex"
         param="nextStart"/></dsp:a>
     </dsp:oparam>
   </dsp:droplet>
</dsp:oparam>
```

When the admin selects users displayed by the code above and hits the Delete button, he or she is redirected to the delete confirmation screen (`users_delete_confirm.jsp`), which shows the users and a Delete button to confirm.

We get the repository ids of all the users that were selected to delete in the previous screen and fetch their profiles using the `ProfileLookUp` component to display user info:

```
<dsp:form action="company_admin.jsp" method="post">

   <dsp:droplet name="ForEach">
      <dsp:param bean="DeleteProfileFormHandler.repositoryIds" name="array"/>
      <dsp:param name="elementName" value="userId"/>
      <dsp:oparam name="output">
         <dsp:droplet name="ProfileLookUp">
            <dsp:param name="id" param="userId"/>
            <dsp:param name="elementName" value="user"/>
            <dsp:oparam name="output">
            <dsp:valueof param="user.firstName"/>
             
            <dsp:valueof param="user.lastName"/><br>
            </dsp:oparam>
         </dsp:droplet>
      </dsp:oparam>
   </dsp:droplet>

<br>
   <dsp:input bean="DeleteProfileFormHandler.delete" type="submit"
      value="Delete"/>  
   <dsp:input bean="DeleteProfileFormHandler.cancel" type="submit"
      value="Cancel"/>
</dsp:form>
```

Using the B2BRepositoryFormHandler

ATG Business Commerce includes the `RepositoryFormHandler`, which allows you to create, update, and remove repository items from any repository. For Motorprise, we wanted to assign newly created repository items to properties of other items, so we extended this class to create `B2BRepositoryFormHandler`, located at `atg/projects/b2bstore/repository/`. `B2BRepositoryFormHandler` helps automatically manage the relationships between existing repository items and repository items that are created and removed by this form handler.

(For more information on the `RepositoryFormHandler`, see the *ATG Page Developer's Guide*.)

`B2BRepositoryFormHandler` provides two important features: adding and removing references to repository items.

Adding References to Repository Items

`B2BRepositoryFormHandler` can automatically add newly created repository items to properties of existing repository items. For example, you can use this form handler to manage the relationship between organizations and shipping addresses; if a company admin creates a new shipping address in Motorprise, `B2BRepositoryFormHandler` can assign it to whatever the selected business unit is.

Removing References to Repository Items

`B2BRepositoryFormHandler` can remove references to repository items when they are deleted from the repository. It overrides `deleteItem` to check for references to each repository item before deleting it. Only when the last reference has disappeared is the item actually deleted from the repository. In B2B Commerce, multiple organizations often refer to same items. For example, the USMW Retail and USMW Wholesale divisions might share a shipping address. Removing this shipping address from the Retail division should not remove it from the repository, because the Wholesale division still refers to it. However, when the address is removed from all divisions, it *should* be removed from the repository. `B2BRepositoryFormHandler` adds logic to the `deleteItem` method to ensure that repository items are not removed from the repository until all references to them have been removed.

Using B2BRepositoryFormHandler in Company Admin

We used the `B2BRepositoryFormHandler` to provide functionality in the Motorprise Company Admin. After administrators have logged in and selected a business unit, they can manage the following within that business unit:

Shipping Addresses	Create new ones. Edit or delete existing ones.
Default Shipping Address	Select a default shipping address from the list of existing shipping addresses.
Billing Addresses	Create new ones. Edit or delete existing ones.
Default Billing Address	Select a default billing address from the list of existing billing addresses.
Credit Cards	Create new ones. Edit or delete existing ones.
Cost Centers	Create new ones. Edit or delete existing ones.
Approvals	Make them required or not. Change the purchase limit.
Payment Methods	Authorize or restrict the use of invoices and credit cards.

For example, the following code, from `shipping_address_create.jsp`, creates a new shipping address and adds that shipping address to the organization to which the admin belongs. This code is explained below sequentially.

In the example, a new repository item of type contactInfo is created and is assigned to the shippingAddrs property of the repository item whose type is organization and whose repository ID is Profile.currentOrganization.repositoryid.

address1 is a contactInfo property and can be accessed using the value dictionary of the B2BRepositoryFormHandler. It is also a part of the requiredFields property.

We use the updateItemDescriptorName property to set the name of the item type to which we are adding the newly created shipping address. In this case , it is organization.

```
<%--  main content area --%>
<td valign="top" width=745>
<dsp:include page="../common/FormError.jsp" flush="true"></dsp:include>
  <dsp:form action="shipping_edit.jsp" method="post">
    <dsp:input bean="B2BRepositoryFormHandler.itemDescriptorName" type="hidden"
      value="contactInfo"/>
    <dsp:input bean="B2BRepositoryFormHandler.updateItemDescriptorName"
      type="hidden" value="organization"/>
```

We use the updateRepositoryId property to set the repository ID of the item to which the new address is being added. In the example below, this is the repository ID of the parent organization, Profile.currentOrganization.repositoryid.

updatePropertyName determines the property of the parent organization to which the newly created shipping address is to be added. In this example, it is set to shippingAddrs, which is the parent organization's list property that contains a list of shipping addresses.

```
    <dsp:input bean="B2BRepositoryFormHandler.updateRepositoryId"
      beanvalue="Profile.currentOrganization.repositoryid" type="hidden"/>
    <dsp:input bean="B2BRepositoryFormHandler.updatePropertyName" type="hidden"
      value="shippingAddrs"/>
```

If one or more fields in the form must be a non-null value, we set RequiredFields so that the user is always forced to enter a value before submitting the form.

If the property being modified is a map instead of a list, as in this case with shippingAddrs, then a key must be provided for the new map entry. This is specified by the updateKey property.

```
    <dsp:input bean="B2BRepositoryFormHandler.requireIdOnCreate" type="hidden"
      value="false"/>
    <dsp:input bean="B2BRepositoryFormHandler.requiredFields" name="hiddenFields"
      type="hidden" value="COMPANYNAME"/>
    <dsp:input bean="B2BRepositoryFormHandler.requiredFields" name="hiddenFields"
      type="hidden" value="ADDRESS1"/>
    <dsp:input bean="B2BRepositoryFormHandler.requiredFields" name="hiddenFields"
      type="hidden" value="CITY"/>
```

```
  <dsp:input bean="B2BRepositoryFormHandler.requiredFields" name="hiddenFields"
    type="hidden" value="POSTALCODE"/>
  <dsp:input bean="B2BRepositoryFormHandler.createErrorURL" type="hidden"
    value="shipping_address_create.jsp"/>
  <dsp:input bean="B2BRepositoryFormHandler.createSuccessURL" type="hidden"
    value="shipping_edit.jsp"/>

 <table border=0 cellpadding=4 width=80%>
   <tr>
     <td colspan=2><dsp:img src="../images/d.gif" vspace="0"/></td>
   </tr>
   <tr>
     <td colspan=2 valign="top"><span class=big>Company Administration</span>
       <br><span class=little><dsp:valueof
       bean="Profile.currentOrganization.name" /></span></td>
   </tr>
   <tr>
     <td colspan=2><dsp:img src="../images/d.gif" vspace="0"/></td>
   </tr>

    <tr>
      <td colspan=2 valign="top">
        <table width=100% cellpadding=3 cellspacing=0 border=0>
          <tr>
            <td class=box-top> Create Shipping Address</td>
          </tr>
        </table>
      </td>
    </tr>
<tr>
  <td colspan=2><span class=small>The nickname field is used to identify this
    address when you don't have access to the full address. It should be unique
    from all other shipping address nicknames. It can be the same as the company
    name.</span></td>
</tr>

    <tr>
      <td colspan=2><dsp:img src="../images/d.gif" vspace="0"/></td>
    </tr>
    <tr>
      <td align=right><span class=smallb>Nickname</span></td>
      <td width=75%><dsp:input bean="B2BRepositoryFormHandler.updateKey"
```

Changing the Scope of Form Handlers

In Motorprise, we overrode the `$scope` of the `MultiUserAddFormHandler` and `MultiUserUpdateFormHandler` components from `request` to `session` to address our multi-page registration process in the Company Admin section of the store.

We wanted the information entered on various pages to be preserved throughout the entire process. Because this information is stored in the form handlers, we made them session-scoped

We chose to override the scope in the actual components because Motorprise is the only application that's running for the ATG Business Commerce Reference Application. You could also configure your own instances of these components by copying the standard configurations and changing the scope to session in your versions.

If you change the scope in the original components, when you upgrade to a new version of ATG Business Commerce, or install a patch that adds new properties to the form handlers or changes their default settings, you get the changes without having to do anything special.

However, if you have other applications running on the same server, and you override the scope in the original components, they will use session-scoped form handlers where they might expect request-scoped form handlers.

If you want to configure your own instances of these components, you must copy the standard configurations and change the scope to session in your versions.

To do this, you would copy the entire configuration of each default components to an application-specific path such as `/atg/projects/B2BStore/userprofiling/MultiProfileFormHandler`, change the scope of that component, and then use that component path in your JSPs.

You could set all of the property values using ^= to refer to the original component, like this:

```
trimProperties^=/atg/userprofiling/MultiProfileAddFormHandler.trimProperties
```

That way you don't have to worry about exactly what the values are, you just have to make sure that you copy all of them into your component's configuration. Using ^= copies a property value from one object to another. For more information on this command, see Using Nucleus, Linking Property Values in the *Nucleus: Organizing JavaBean Components* chapter of the *ATG Programming Guide*.

However, when you create your own objects, you must remember to check the configurations when you install a new version or patch and update them to account for any properties that might be added to or removed from the original components.

6 My Account

In the My Account section of the Motorprise site, users can review their open and fulfilled orders. They also have the ability to create purchase lists and edit their profile information. In addition, those Motorprise users who are approvers can manage their approvals in the My Account section.

This chapter describes how we created the My Account section and includes the following sections:

Reviewing Orders
Describes how open, fulfilled, and rejected orders are displayed.

Using Purchase Lists
Describes purchase list functionality.

Scheduling Orders
Describes how users schedule orders.

Saving Orders
Describes how users save orders.

Approving Orders
Describes the approval process.

Reviewing Orders

When buyers log into the Motorprise site, their open, fulfilled, and rejected orders are displayed on the home page.

Buyers see a summary of their open, fulfilled, and rejected orders when they log on to the home page.

Motorprise buyers can also view their orders in the My Account section of the store. Each order is displayed with its number, date, and status.

Open orders in the buyer's My Account section of Motorprise

Buyers can click on any order number to see a full view of that order.

We used the following JSP code fragment from `MotorpriseJSP/j22ee-apps/motorprise/web-app/en/users/orders_open.jsp` to display the list of orders:

```
<dsp:getvalueof id="profileId" idtype="java.lang.String" bean="Profile.id">
  <%-- profileId:  <%=profileId%> --%>
  <dsp:droplet name="OrderLookup">
    <dsp:param value="<%=profileId%>" name="userId"/>
    <dsp:param name="state" value="open"/>
    <dsp:param name="startIndex" param="startIndex"/>
    <dsp:oparam name="output">
      <dsp:droplet name="ForEach">
        <dsp:param name="array" param="result"/>
        <dsp:oparam name="outputStart">
          <table border=0 width=95%>
            <tr valign=top>
              <td width=33%><span class="small"><b>Order #</b></span></td>
              <td width=33%><span class="small"><b>Date ordered</b></span></td>
              <td width=33%><span class="small"><b>Status</b></span></td>
            </tr>
            <tr>
              <td colspan=3><hr size=1 color="#666666"></td>
            </tr>
```

```
</dsp:oparam>

<dsp:oparam name="output">
  <tr>
    <dsp:droplet name="Switch">
    <dsp:param name="value" param="element.originOfOrder"/>
    <dsp:oparam name="default">
      <td><dsp:a href="order.jsp">
      <dsp:param name="orderId" param="element.id"/>
      <dsp:valueof param="element.id"/><dsp:param name="orderState"
        param="state"/>
      </dsp:a></td>
    </dsp:oparam>
    <dsp:oparam name="scheduledOrder">
      <td><dsp:a href="order.jsp">
      <dsp:param name="orderId" param="element.id"/>
      <dsp:valueof param="element.id"/>
      </dsp:a>(ScheduledOrder)</td>
    </dsp:oparam>
    </dsp:droplet>
    <td><dsp:valueof date="MMMMM d, yyyy"
    param="element.submittedDate"/></td>
    <td><dsp:valueof param="element.stateAsString"/></td>
  </tr>

</dsp:oparam>
<dsp:oparam name="outputEnd">
  <tr></tr>
  <tr>
    <td colspan=3>
        <hr size=1 color="#666666">
        Now viewing orders
        <dsp:valueof param="startRange"/> -
        <dsp:valueof param="endRange"/> out of
        <dsp:valueof param="total_count"/>
    </td>
    <td>
        <dsp:droplet name="IsEmpty">
          <dsp:param name="value" param="previousIndex"/>
          <dsp:oparam name="false">
            <dsp:a href="orders_open.jsp">
            <dsp:param name="startIndex" param="previousIndex"/>
            Previous</dsp:a>
          </dsp:oparam>
        </dsp:droplet>

        <dsp:droplet name="IsEmpty">
          <dsp:param name="value" param="nextIndex"/>
          <dsp:oparam name="false">
            <dsp:a href="orders_open.jsp">
```

```
                    <dsp:param name="startIndex" param="nextIndex"/>
                    Next</dsp:a>
                  </dsp:oparam>
                </dsp:droplet>
              </td>
            </tr>
          </dsp:oparam>


      </dsp:droplet>
    </dsp:oparam>
    <dsp:oparam name="empty">
      <%/* no open orders for user */%>
        You have no open orders.
    </dsp:oparam>
    <dsp:oparam name="error">
      <dsp:valueof param="errorMsg"/>
    </dsp:oparam>
  </dsp:droplet>
</dsp:getvalueof>
```

We used the `OrderLookup` component to fetch the orders for the user. It takes the `userId`, `startIndex`, and `state` of each order to fetch and outputs a result array that contains `defaultNumOrders` if no `numOrders` parameter is given.

It's possible that users could have hundreds of orders, so we wanted to make them easy to browse them. We decided to display orders in groups of ten, with links to go to the previous or next group. We used the parameters of `OrderLookup` to provide this kind of page navigation of the orders.

The number of orders returned by `OrderLookup` is determined using the parameter `numOrders`. If this value is not set, then the property `defaultNumOrders` determines the number of orders to return. In addition, the `startIndex` parameter is used to specify the index of the order from which to return the orders. We use `OrderLookup` to set the parameters `previousIndex` and `nextIndex`.

We also use `OrderLookup` to output other parameters to display only a certain number of orders at a time. Based on these parameters, we provide links for the previous and next pages to display the set of orders as shown in the above code.

If you want to display all the orders on a single page, you would set the `defaultNumOrders` property of `OrderLookup` to –1 and the output parameter result array will contain all the orders with the given state for the given user.

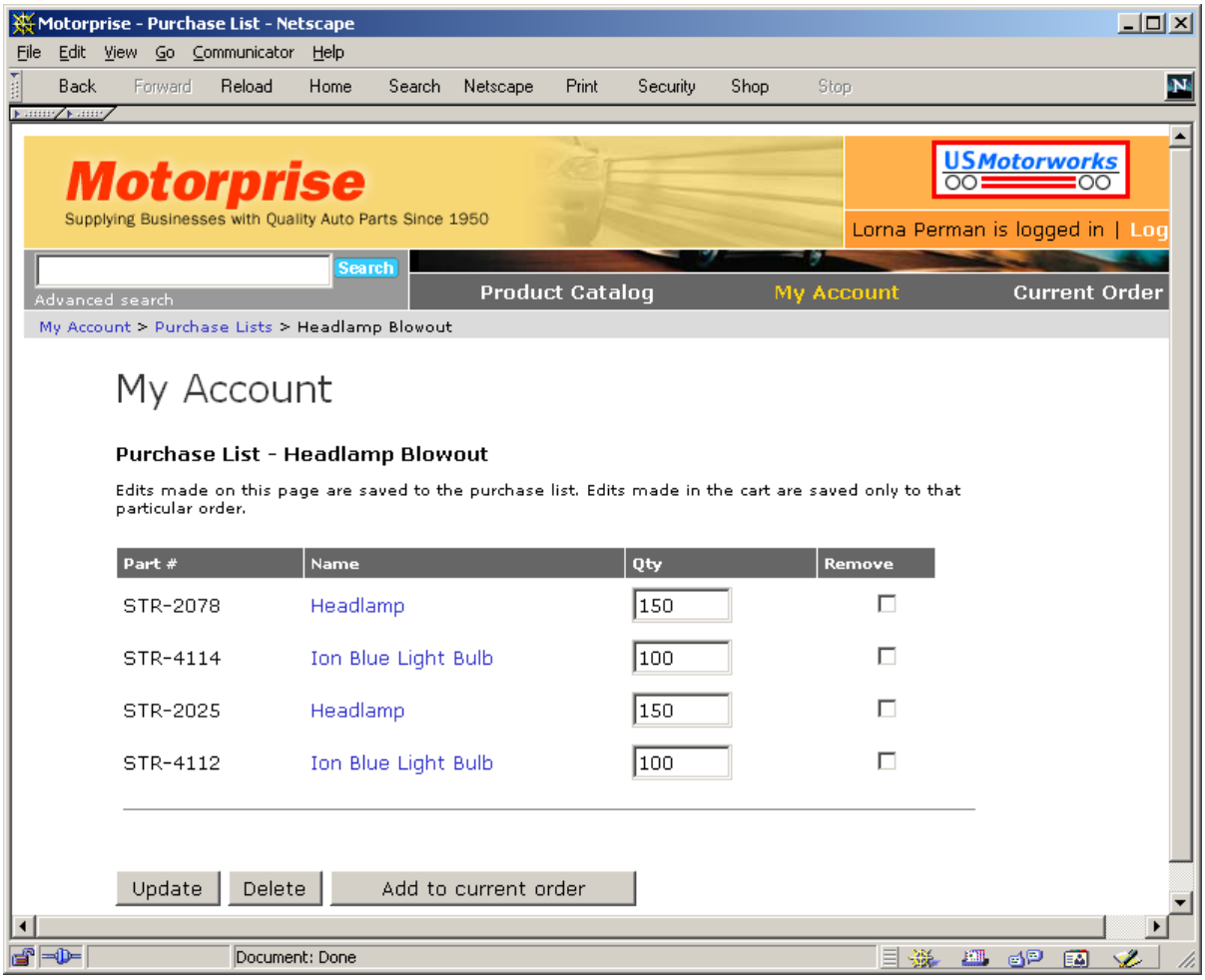
For more information on order states, see the ATG Commerce States section of the *Working With Purchase Process Objects* chapter in the *ATG Commerce Programming Guide*.

Using Purchase Lists

In Motorprise, a purchase list is a list of frequently ordered items that are typically ordered together. Customers can create and maintain these lists so that they do not have to look through the catalog for the same products each time they want to place the order. Instead, they can create a purchase list once. Then, whenever they want to order those items, they can select the purchase list and add all or some of the items in it to their order.

For example, Lorna Perman, a buyer at USMW US Division, Retail, created a purchase list called Headlamp Blowout. This is a set of headlamps and light bulbs that the Retail Division regularly features in a sale.

Lorna can add all the items in that purchase list to her current order. She can then choose additional items from the catalog or delete items from her order.



Lorna Perman's "Headlamp Blowout" purchase list.

Purchase lists are persistent and specific to a buyer. Also, each purchase list for a specific buyer must have a unique name. After Lorna adds a purchase list to her current order, the purchase list remains in her profile. She can use that list again or delete it later on the Purchase List page in My Account.

Buyers can edit, remove, or add purchase lists. Any changes made to a purchase list will not affect orders that have been already submitted using that purchase list. If the buyer edits a purchase list, she is prompted to save these changes permanently.

When a customer is looking at products in the catalog (viewing `<ATG10dir>/MotorpriseJSP/j2ee-apps/Motorprise/web-app/en/catalog/product.jsp`), she has the option to create a new purchase list or to add the displayed product to a previously saved purchase list.

If she clicks the **Add to List** button and selects an existing purchase list, the product is added to that purchase list. The following code shows how we implemented this feature in `MotorpriseJSP/j22ee-apps/motorprise/web-app/en/catalog/AddToList.jsp`.

```
<dsp:form action="product.jsp" method="post">
  <input name="id" type="hidden" value="<dsp:valueof
    param="product.repositoryId"/>">
  <dsp:input bean="PurchaselistFormHandler.addItemToPurchaselistErrorURL"
     type="hidden" value="product.jsp"/>
  <dsp:input bean="PurchaselistFormHandler.productId"
    paramvalue="product.repositoryId" type="hidden"/>
  <dsp:droplet name="/atg/dynamo/droplet/ForEach">
    <dsp:param name="array" param="product.childSKUs"/>
    <dsp:oparam name="output">
      <table border=0 cellpadding=3 width=100%>
        <tr>
         <td><dsp:input bean="PurchaselistFormHandler.catalogRefIds"
           paramvalue="element.repositoryId" type="hidden"/>
         <span class=smallb>Qty</span> 
         <dsp:input bean="PurchaselistFormHandler.quantity" size="2" type="text"
           value="1"/> 
    </dsp:oparam>
  </dsp:droplet>

  <dsp:select bean="PurchaselistFormHandler.purchaseListId">
    <dsp:droplet name="ForEach">
      <dsp:param bean="Profile.purchaselists" name="array"/>
      <dsp:oparam name="output">
        <dsp:getvalueof id="elem" idtype="atg.repository.RepositoryItem"
          param="element">
          <dsp:option value="<%=elem.getRepositoryId()%>"/>
          <dsp:valueof param="element.eventName">Unnamed Purchase
            List</dsp:valueof>
        </dsp:getvalueof>
      </dsp:oparam>
    </dsp:droplet>
  </dsp:select></td>
```

```
  </tr>
  <tr>
    <td><dsp:input bean="PurchaselistFormHandler.addItemToPurchaselist"
      type="submit" value="Add to list"/></td>
  </tr>
<tr>
    <td>
    <table border=0 cellpadding=3 width=100%>
      <tr>
        <td><span class=smallb><dsp:a href="../user/
          purchase_lists.jsp?noCrumbs=false"><dsp:param name="product"
          param="product.repositoryId"/><dsp:param name="noCrumbs"
          value="false"/>Create new purchase list</dsp:a></span></td>
      </tr>
    </table>
    </td>
  </tr>

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

If she clicks the **Create new purchase list** button, she is taken to the Purchase Lists page, `user/purchase_lists.jsp`, where all of her saved purchase lists are displayed. She has an option here to create a new purchase list. We used the following JSP code to achieve this:

```
<dsp:form action="purchase_lists.jsp" method="post">
<input name="noCrumbs" type="hidden" value="<dsp:valueof param='noCrumbs'/>">
New purchase list:<br>
<dsp:input bean="PurchaselistFormHandler.listName" maxlength="20" size="20"
   type="text"  value=""/>
      <dsp:input bean="PurchaselistFormHandler.savePurchaseList" type="hidden"
        value=""/>
<dsp:input bean="PurchaselistFormHandler.savePurchaseList" type="submit"
    value="Create list"/>
<br>
<span class="help">List names are limited to 20 characters in length</span>
</dsp:form>
```

In the above code, the user is provided an input box to suggest a name for the purchase list. If she doesn't have any purchase lists with that name, we create one for her with the given name and redirect her to the same page, which then displays the newly created purchase list. If the user has another purchase list with that name, we redirect her to the same page displaying the error message saying that a purchase list with that name already exists.

In Motorprise, we extended the ATG Business Commerce gift list functionality to create purchase lists. We extended the `GiftlistFormHandler` component to create `PurchaselistFormHandler`. This component is located at `atg/projects/b2bstore/purchaselists`.

PurchaselistFormHandler.properties:

```
$class=atg.projects.b2bstore.purchaselists.PurchaselistFormHandler
$scope=request
profile=/atg/userprofiling/Profile
defaultLocale^=/atg/commerce/pricing/PricingTools.defaultLocale

giftlistRepository=/atg/commerce/gifts/Giftlists
giftlistManager=/atg/commerce/gifts/GiftlistManager
orderManager=/atg/commerce/order/OrderManager
shoppingCart=/atg/commerce/ShoppingCart
giftlistTools=/atg/commerce/gifts/GiftlistTools
catalogTools=/atg/commerce/catalog/CatalogTools
profileTools=/atg/userprofiling/ProfileTools
```

We created a new property, `purchaseLists`, for users. This property differs from the gift list property in that it doesn't require event dates, names, and shipping address. We set all of these to null in the `PurchaseListFormHandler` and just set the name of the purchase list.

`PurchaselistFormHandler` extends `GiftlistFormHandler`. Before it passes an event, it sets the various properties of the gift list that are not required by purchase list to null. It also checks for duplicate purchase list names. Also, unlike `GiftlistFormHandler`, `PurchaselistFormHandler` does not decrease the quantity of items or remove them from the gift list when they are moved to current order.

We overrode the `personalGiftlistsProperty` of `GiftlistTools` in Motorprise to use `purchaseLists` as the property to hold the user purchase lists.

Scheduling Orders

Motorprise buyers can create scheduled orders to be placed on a pre-defined schedule that does not require any actions from the user. They can create scheduled orders during the checkout process or through the My Account pages.

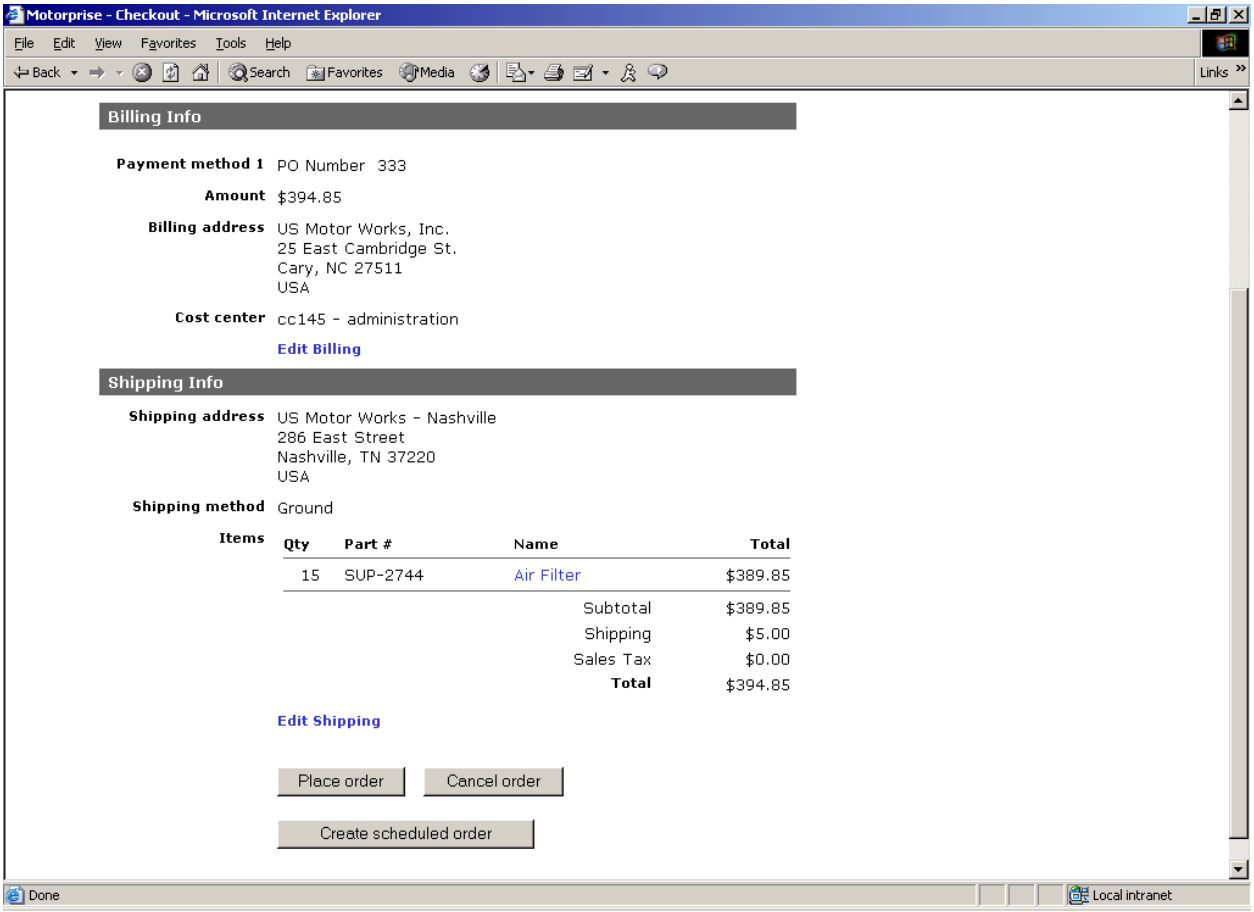
If a scheduled order requires approval, it is automatically sent to the approver at the specified interval with a "pending approval" status. Once it has been approved, the order is fulfilled automatically.

We used `/atg/commerce/order/scheduled/ScheduledOrderFormHandler` (an instance of `atg.b2bcommerce.order.scheduled.ScheduledOrderHandler`) to manage scheduled orders. This session-scoped component is responsible for verifying, creating, updating, and deleting the scheduled order.

Scheduling Orders during Checkout

When buyers go through the normal checkout process, instead of clicking the **Place order** button on the order confirmation screen, they can click the **Create scheduled order** button. They are then directed to

the Scheduled Order page (`user/scheduled_order_new.jsp`), where they can specify the name and type (daily, weekly or monthly) of the scheduled order.



Scheduling an order during checkout.

The following is the code in `scheduled_order_new.jsp` used to create a scheduled order.

First we find the ID number for the order by checking if `prototypeOrderId` is set or not:

```
<dsp:droplet name="IsNull">
        <dsp:param name="value" param="prototypeOrderId"/>
        <dsp:oparam name="true">
```

Then we use the `ScheduledOrderFormHandler` to let the user fill in the order information:

```
<tr valign=bottom>
  <td valign="top" align=right><span class=smallb>Name</span></td>
  <td><dsp:input bean="ScheduledOrderFormHandler.value.name" name="name" size="35"
```

```
    type="text"/><br>
  <span class=help>Enter a name for your scheduled order (i.e. Weekly Spark Plug
    Assortment).</span></td>
</tr>
<tr>
  <td align=right><span class=smallb>Schedule type</span></td>
  <td align="left">
    <dsp:select bean="ScheduledOrderFormHandler.moveToMode" name="select"
      size="1">
      <dsp:option value=""/>Choose schedule
      <dsp:option value="Daily"/>daily
      <dsp:option value="Weekly"/>weekly
      <dsp:option value="Monthly"/>monthly
    </dsp:select>
  </td>
</tr>

<tr>
  <td align="right" valign=top><span class=smallb>Order</span></td>
  <td>
    <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      templateOrderId.userInputFields.orderId">
      <dsp:droplet name="ForEach">
        <dsp:param name="array" param="result"/>
      <dsp:oparam name="outputStart">
        <dsp:option value=""/>  Select a fulfilled order
      </dsp:oparam>

      <dsp:oparam name="output">
        <dsp:getvalueof id="elementId" idtype="String" param="element.id">
          <dsp:option value="<%=elementId%>"/><dsp:valueof param="element.id"/>
        </dsp:getvalueof>
      </dsp:oparam>

      </dsp:droplet>
    </dsp:select>
<br>


  </td>
</tr>

<tr><td><img src="../images/d.gif"></td></tr>
<tr>
  <td></td>
    <input name="createNew" type="hidden" value="<dsp:valueof
      param="createNew"/>">
    <dsp:input bean="ScheduledOrderFormHandler.moveToErrorURL" type="hidden"
      value="scheduled_order_new.jsp"/>
    <dsp:input bean="ScheduledOrderFormHandler.moveToSuccessURL" type="hidden"
```

```
      value="scheduled_order_calendar.jsp?createNew=new"/>
      <td><dsp:input bean="ScheduledOrderFormHandler.moveToURL" type="submit"
        value="Continue"/></td>
</tr>
```

We used the same page, `scheduled_order_new.jsp`, to create scheduled orders from the checkout page and from My Account. (If the user creates a new scheduled order from the Scheduled Orders page in My Account, we display a list of all the user's fulfilled orders. See the *Scheduling Orders in My Account* section below for more information.) Based on the hidden form field `prototypeOrderId`, which is set in the order confirmation screen, we determine whether we are creating a scheduled order from the checkout process.

When the user chooses to create a scheduled order from the order confirmation page, we assign the ID of the order being created to `prototypeOrderId` and pass it to `scheduled_order_new.jsp`, which assigns the order ID to `ScheduledOrderFormHandler.complexScheduledOrderMap.templateOrderId.userInputFields.orderId`.

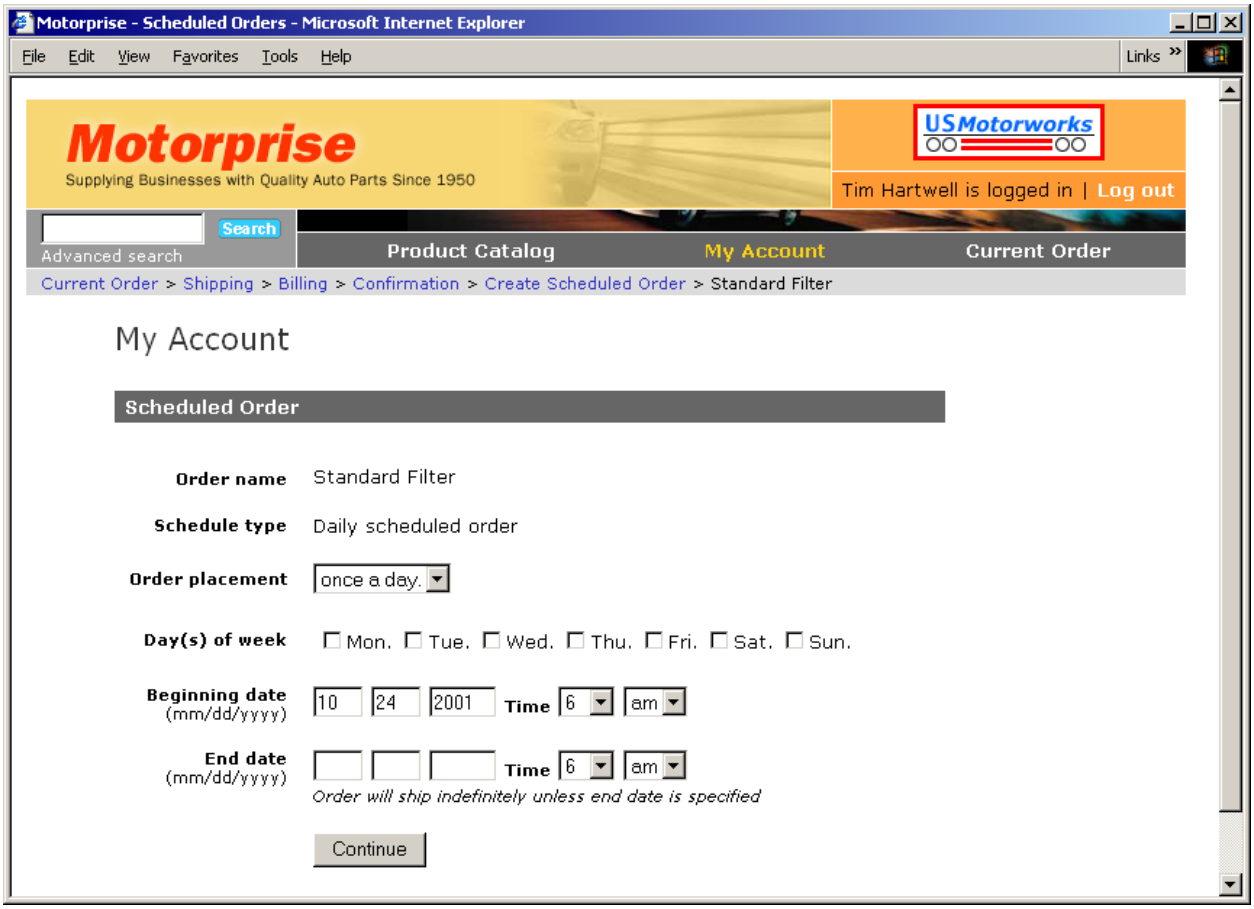
Once the buyer enters the name of the order and the schedule type, he or she is directed to `scheduled_order_calendar.jsp`, which displays a calendar for the order depending on its schedule type: daily, weekly or monthly. We used the following JSP code to display the appropriate calendar:

```
<dsp:form action="scheduled_order_preview.jsp" method="post">
<table border=0 cellpadding=8>
   <tr valign=bottom>
      <td align=right><span class=smallb>Order name</span></td>
      <td><dsp:valueof bean="ScheduledOrderFormHandler.value.name" /></td>
   </tr>
   <tr>
      <td align=right><span class=smallb>Schedule type</span></td>
      <td> <dsp:valueof bean="ScheduledOrderFormHandler.moveToMode" />
         scheduled order</td>
   </tr>

<dsp:droplet name="Switch">
   <dsp:param bean="ScheduledOrderFormHandler.moveToMode" name="value"/>

   <dsp:oparam name="Daily">
      <tr>
         <td align="right"><span class=smallb>Order placement</span></td>
         <td align="left">
            <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
               calendarSchedule.userInputFields.scheduleMode" name="select"
               size="1">
               <dsp:option value="onceDaily"/>once a day.
               <dsp:option value="twiceDaily"/>twice a day.
            </dsp:select>
         </td>
      </tr>
```

```
   <%@ include file="day_of_week.jspf" %>
</dsp:oparam>

<dsp:oparam name="Weekly">
   <dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      calendarSchedule.userInputFields.scheduleMode" type="hidden"
      value="Weekly"/>
   <%@ include file="day_of_week1.jspf"%>
   <tr>
      <td align="right" valign="top"><span class=smallb>Week(s) of
         month</span></td>
      <td align="left">
      <table border="0">
         <tr>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="1"/>1st. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="2"/>2nd. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="3"/>3rd. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
            complexScheduledOrderMap.calendarSchedule.userInputFields.
            occuranceInMonth" name="occuranceInMonth" type="Checkbox"
            value="4"/>4th. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="5"/>5th. </td>
         </tr>
         <tr><td colspan=5><span class=help>Uncheck to ship selected
            weeks</span></td></tr>
      </table>
   </tr>
</dsp:oparam>

<dsp:oparam name="Monthly">
   <tr>
      <td align="right"><span class=smallb>Order Placement</span></td>
      <td align="left">
         <dsp:select bean="ScheduledOrderFormHandler.
            complexScheduledOrderMap.calendarSchedule.userInputFields.
            scheduleMode" name="select" size="1">
            <dsp:option value="onceMonthly"/>once a month.
            <dsp:option value="biMonthly"/>every two months.
```

```
                <dsp:option value="quarterly"/>every quarter.
              </dsp:select>
            </td>
          </tr>
        </dsp:oparam>

        <dsp:oparam name="default">
        </dsp:oparam>
</dsp:droplet>
```

The above block of code shows the control flow we used to display the appropriate calendar chosen by the user and set in `ScheduleOrderFormHandler.moveToMode`. There are three schedule types: daily, weekly, and monthly. The calendar for each schedule type is explained below.



Calendar page for creating a daily scheduled order.

The code for Daily schedule is:

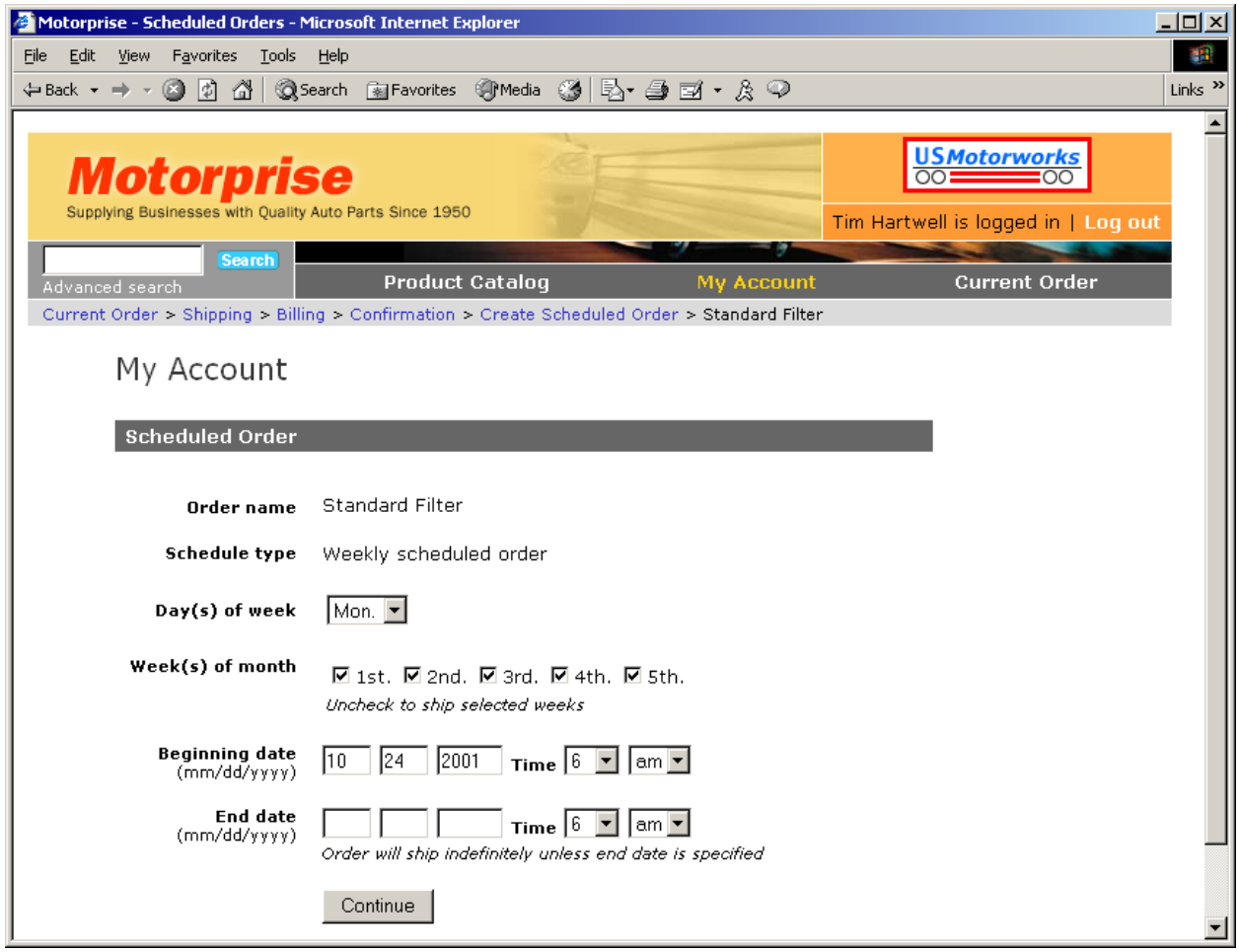
```
<dsp:oparam name="Daily">
   <tr>
      <td align="right"><span class=smallb>Order placement</span></td>
      <td align="left">
         <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
            calendarSchedule.userInputFields.scheduleMode" name="select"
            size="1">
            <dsp:option value="onceDaily"/>once a day.
            <dsp:option value="twiceDaily"/>twice a day.
         </dsp:select>
      </td>
   </tr>
   <%@ include file="day_of_week.jspf" %>
</dsp:oparam>
```

The day_of_week.jspf page fragment contains the following code that displays the days of the week and assigns them to complexScheduledOrderMap as shown below:

```
dsp:importbean bean="/atg/commerce/order/scheduled/ScheduledOrderFormHandler"/>
<!-- Selection of days for shipment including x times/week -->
<tr>
  <td align="right"><span class=smallb>Day(s) of week</span></td>
  <td align="left">
    <table>
      <tr>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
           type="checkbox" value="2"/>Mon. </td>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
           type="checkbox" value="3"/>Tue. </td>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
           type="checkbox" value="4"/>Wed. </td>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
           type="Checkbox" value="5"/>Thu. </td>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
              type="checkbox" value="6"/>Fri. </td>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
              type="Checkbox" value="7"/>Sat. </td>
        <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
           calendarSchedule.userInputFields.daysOfWeek" name="daysOfWeek"
           type="Checkbox" value="1"/>Sun. </td>
     </tr>
  </table>
```

```
</tr>
</dsp:page>
```

As shown above, we display the options for Daily schedule, once a day and twice a day, and the days of the week with corresponding checkboxes from which to choose.



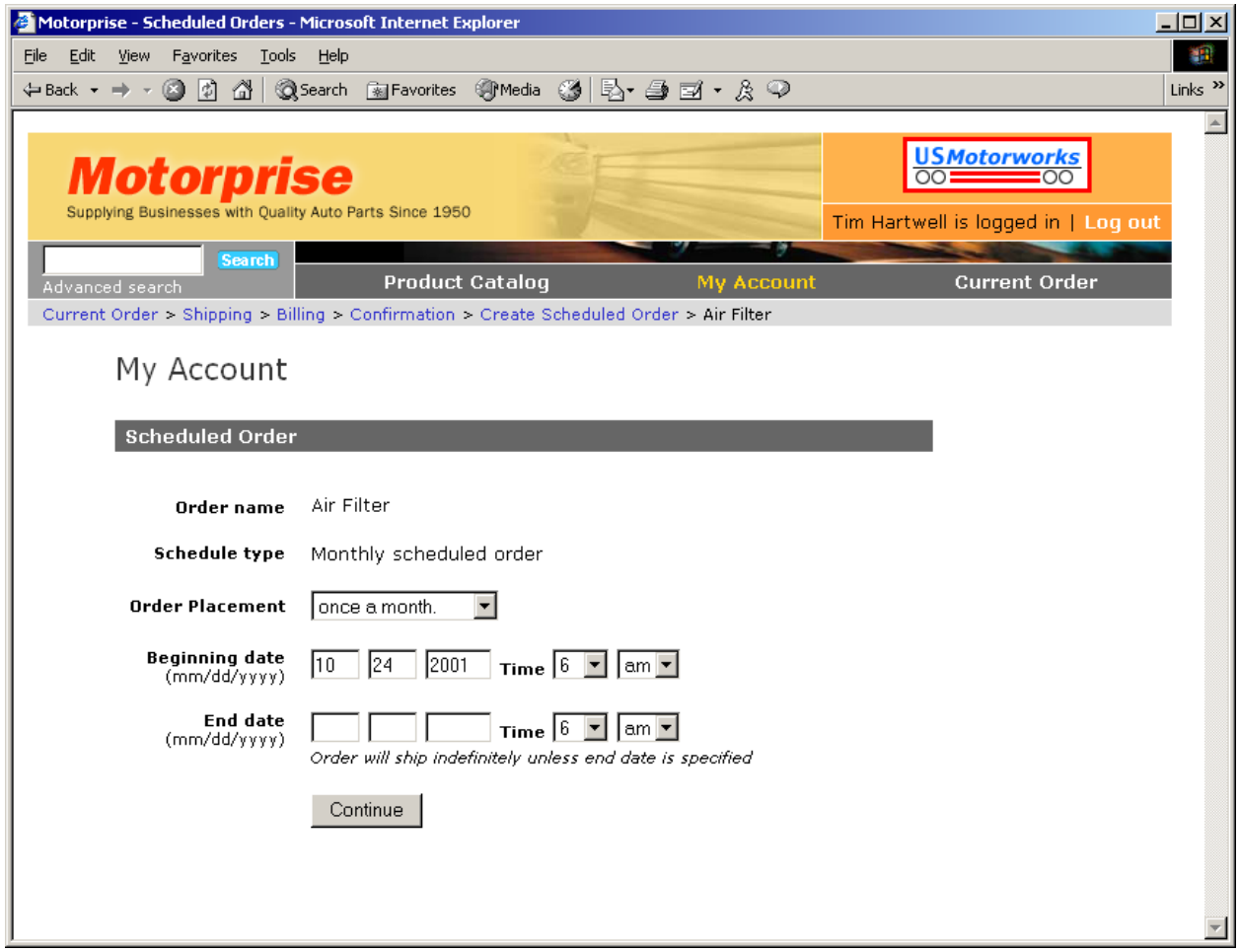
Calendar page for creating a weekly scheduled order.

The code for Weekly schedule is:

```
<dsp:oparam name="Weekly">
   <dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      calendarSchedule.userInputFields.scheduleMode" type="hidden"
         value="Weekly"/>
   <%@ include file="day_of_week1.jspf"%>
```

```
   <tr>
      <td align="right" valign="top"><span class=smallb>Week(s) of
         month</span></td>
      <td align="left">
      <table border="0">
         <tr>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="1"/>1st. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="2"/>2nd. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="checkbox"
               value="3"/>3rd. </td>
            <td><dsp:input bean="ScheduledOrderFormHandler.
               complexScheduledOrderMap.calendarSchedule.userInputFields.
               occuranceInMonth" name="occuranceInMonth" type="Checkbox"
               value="4"/>4th. </td>
           <td><dsp:input bean="ScheduledOrderFormHandler.
              complexScheduledOrderMap.calendarSchedule.userInputFields.
              occuranceInMonth" name="occuranceInMonth" type="checkbox"
              value="5"/>5th. </td>
        </tr>
        <tr><td colspan=5><span class=help>Uncheck to ship selected
           weeks</span></td></tr>
      </table>
   </tr>
</dsp:oparam>
```

The code above sets the `ScheduledOrderFormHandler.scheduledMode` to Weekly and displays the days of the week in a select box for the user to choose any day of the week. It also displays all the weeks in the month numerically and provides corresponding checkboxes so the user can choose a week to schedule the order.



Calendar page for creating a monthly scheduled order.

The following is the code to display Monthly schedule:

```
<dsp:oparam name="Monthly">
   <tr>
      <td align="right"><span class=smallb>Order Placement</span></td>
      <td align="left">
         <dsp:select bean="ScheduledOrderFormHandler.
            complexScheduledOrderMap.calendarSchedule.userInputFields.
            scheduleMode" name="select" size="1">
            <dsp:option value="onceMonthly"/>once a month.
            <dsp:option value="biMonthly"/>every two months.
            <dsp:option value="quarterly"/>every quarter.
         </dsp:select>
      </td>
```

```
   </tr>
</dsp:oparam>
```

There are three modes for the Monthly schedule from which the user can choose: once a month, every two months, every quarter.

Finally, for each calendar, we display fields for start and end dates. The following code snippet is from `start_end_date.jspf`, which is invoked in `scheduled_order_new.jsp`, to display the start and end date options:

```
<tr valign=top>
  <td align=right><span class=smallb>Beginning date</span><br>
  <span class="small">(mm/dd/yyyy)</span></td>
  <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
    startDate.userInputFields.month" maxlength="2" size="2" type="text"/>
  <dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
    startDate.userInputFields.day" maxlength="2" size="2" type="text"/>
  <dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
    startDate.userInputFields.year" maxlength="4" size="4" type="text"/>
  <span class=smallb>Time</span>

  <dsp:droplet name="Switch">
  <dsp:param bean="Profile.locale" name="value"/>
  <dsp:oparam name="en_US">
    <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      startDate.userInputFields.hour">
    <dsp:droplet name="For">
      <dsp:param name="howMany" value="12"/>
      <dsp:oparam name="output">
        <dsp:getvalueof id="numHoursEn" idtype="Integer" param="count">
          <dsp:option value="<%=numHoursEn.toString()%>"/> <dsp:valueof
            param="count"/>
        </dsp:getvalueof>
      </dsp:oparam>
    </dsp:droplet>
    </dsp:select>

    <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      startDate.userInputFields.ampm">
     <dsp:option value="am"/> am
     <dsp:option value="pm"/> pm
    </dsp:select>
  </dsp:oparam>
  <dsp:oparam name="de_DE">
    <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      startDate.userInputFields.hour">
    <dsp:droplet name="For">
      <dsp:param name="howMany" value="24"/>
```

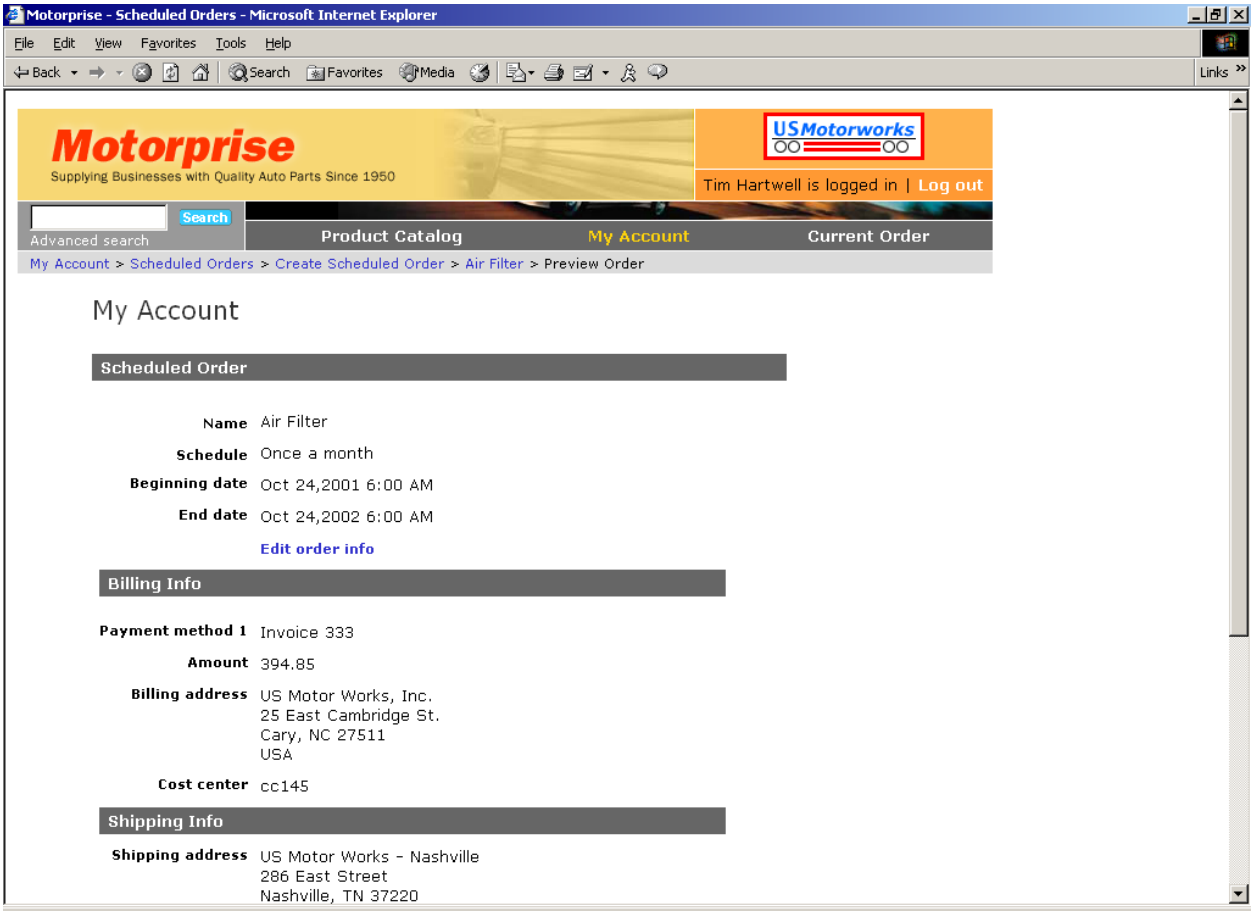
```
      <dsp:oparam name="output">
        <dsp:getvalueof id="numHoursDe" idtype="Integer" param="count">
          <dsp:option value="<%= numHoursDe.toString() %>"/> <dsp:valueof
            param="count"/>
        </dsp:getvalueof>
      </dsp:oparam>
    </dsp:droplet>
    </dsp:select>
  </dsp:oparam>
  </dsp:droplet>
  </td>
</tr>


<tr valign=top>
     <!--<dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
       endDate.userInputFields.mode" type="hidden" value="definite"/>-->
  <td align=right><span class=smallb>End date</span><br>
  <span class="small">(mm/dd/yyyy)</span></td>
  <td><dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
    endDate.userInputFields.month" maxlength="2" size="2" type="text"/>
  <dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
    endDate.userInputFields.day" maxlength="2" size="2" type="text"/>
  <dsp:input bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
    endDate.userInputFields.year" maxlength="4" size="4" type="text"/>
  <span class=smallb>Time</span>

  <dsp:droplet name="Switch">
  <dsp:param bean="Profile.locale" name="value"/>
  <dsp:oparam name="en_US">
    <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      endDate.userInputFields.hour">
    <dsp:droplet name="For">
      <dsp:param name="howMany" value="12"/>
        <dsp:oparam name="output">
        <dsp:getvalueof id="hourEndDe" idtype="Integer" param="count">
          <dsp:option value="<%=hourEndDe.toString()%>"/> <dsp:valueof
            param="count"/>
        </dsp:getvalueof>

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

   <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.endDate.
     userInputFields.ampm">
    <dsp:option value="am"/> am
    <dsp:option value="pm"/> pm
   </dsp:select>
  </dsp:oparam>
```

```
  <dsp:oparam name="de_DE">
    <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
      endDate.userInputFields.hour">
    <dsp:droplet name="For">
      <dsp:param name="howMany" value="24"/>
      <dsp:oparam name="output">
        <dsp:getvalueof id="hourEndEn" idtype="Integer" param="count">
          <dsp:option value="<%=hourEndEn.toString()%>"/> <dsp:valueof
            param="count"/>
        </dsp:getvalueof>
      </dsp:oparam>
    </dsp:droplet>
    </dsp:select>
  </dsp:oparam>
  </dsp:droplet>
```

We display submit buttons so that user can choose any of the options provided above and preview the order. The following is the relevant code from `scheduled_order_new.jsp`:

```
<tr>
  <td></td>
  <input name="createNew" type="hidden" value="<dsp:valueof param="createNew"/>">
  <dsp:input bean="ScheduledOrderFormHandler.moveToErrorURL" type="hidden"
    value="scheduled_order_new.jsp"/>
  <dsp:input bean="ScheduledOrderFormHandler.moveToSuccessURL" type="hidden"
    value="scheduled_order_calendar.jsp?createNew=new"/>
  <td><dsp:input bean="ScheduledOrderFormHandler.moveToURL" type="submit"
    value="Continue"/></td>
</tr>
```

Once the user chooses the calendar options and clicks the **Continue** button, he or she is directed to `scheduled_order_preview.jsp`, which displays the order information and a submit button to create the scheduled order.



Previewing the scheduled order.

The following is the code from `scheduled_order_preview.jsp` for displaying the submit button for creating or deleting the scheduled order:

```
<dsp:droplet name="IsNull">
  <dsp:param name="value" bean="ScheduledOrderFormHandler.repositoryId"/>
  <dsp:oparam name="true">
    <dsp:input bean="ScheduledOrderFormHandler.createErrorURL" type="hidden"
      value="scheduled_order_new.jsp"/>
    <dsp:input bean="ScheduledOrderFormHandler.createSuccessURL" type="hidden"
      value="scheduled_orders.jsp"/>
    <dsp:input bean="ScheduledOrderFormHandler.create" type="submit" value="Create
      scheduled order"/>
  </dsp:oparam>
  <dsp:oparam name="false">
    <dsp:input bean="ScheduledOrderFormHandler.deleteSuccessURL" type="hidden"
      value="scheduled_orders.jsp"/>
```

```
    <dsp:input bean="ScheduledOrderFormHandler.deleteErrorURL" type="hidden"
      value="scheduled_order_preview.jsp"/>
    <dsp:input bean="ScheduledOrderFormHandler.delete" type="submit" value="Delete
      scheduled order"/><p>
    <span class=smallb><dsp:a href="scheduled_orders.jsp">Return to scheduled
      orders</dsp:a></span>
  </dsp:oparam>
</dsp:droplet>
```

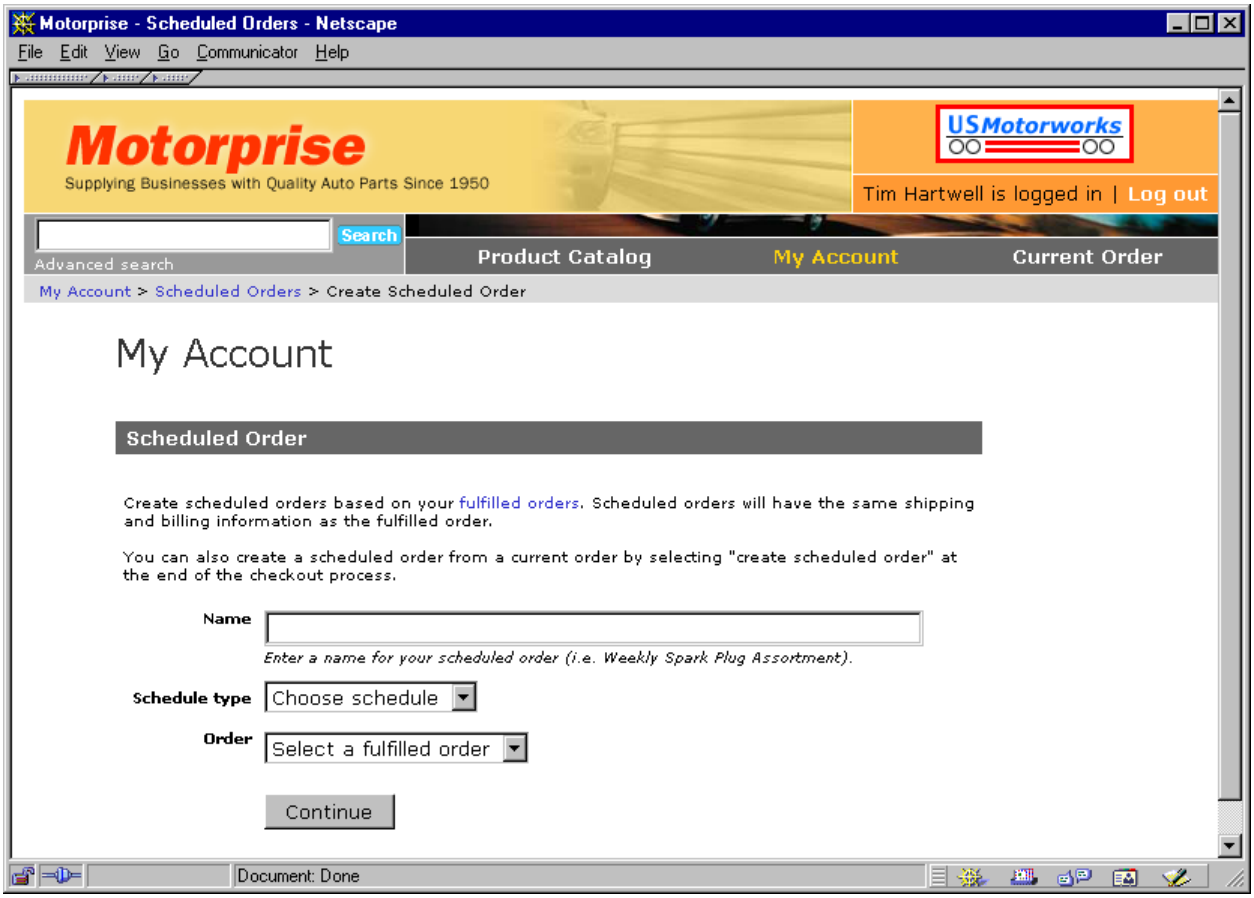
The user is then directed to `scheduled_orders.jsp`, which actually displays all his or her scheduled orders. The following is the code used to fetch and display the scheduled orders for a given user:

```
<dsp:droplet name="RQLQueryForEach">
   <dsp:param bean="/atg/dynamo/transaction/TransactionManager"
      name="transactionManager"/>
   <dsp:param bean="/atg/commerce/order/OrderRepository" name="repository"/>
   <dsp:param name="itemDescriptor" value="scheduledOrder"/>
   <dsp:param name="queryRQL" value="profileId=:profileId"/>
   <dsp:param bean="Profile.repositoryId" name="profileId"/>
   <dsp:oparam name="output">
      <dsp:a href="scheduled_order_preview.jsp">
      <dsp:param name="scheduledOrderId" param="element.repositoryId"/>
      <dsp:param name="source" value="scheduledOrder"/>
      <dsp:valueof param="element.name"/>
      </dsp:a><br>
   </dsp:oparam>
   <dsp:oparam name="empty">
      You have no scheduled orders.
   </dsp:oparam>
</dsp:droplet>
```

We used the `RQLQueryForEach` servlet bean to fetch the scheduled orders for a given user and provide a link to `scheduled_order_preview.jsp` for each order. It assigns the repository ID of each scheduled order to the `scheduledOrderId` parameter and passes it to the page. For more information on the `RQLQueryForEach` servlet bean, please refer to the *ATG Page Developer's Guide*.

Scheduling Orders in My Account

Users can also create scheduled orders by clicking the **Create scheduled order** link on the Scheduled Orders page (`user/scheduled_orders.jsp`) in My Account.



Setting up a scheduled order in My Account.

Creating scheduled orders from My Account is similar to creating them from the order confirmation page. However, the template order is passed to `ScheduledOrderFormHandler` differently. When the user chooses to create a scheduled order from the order confirmation page, we assign the ID of the order being created to `prototypeOrderId` and pass it to `scheduled_order_new.jsp`, which assigns the order ID to `ScheduledOrderFormHandler.complexScheduledOrderMap.templateOrderId.userInputFields.orderId`. If the user clicks the **Create new scheduled order** link from the Scheduled Orders page in My Account, we display a list of all the user's fulfilled orders. The user can select an order from this list that is assigned to `ScheduledOrderFormHandler.complexScheduledOrderMap.templateOrderId.userInputFields.orderId`.

The following is the code in `scheduled_order_new.jsp` that displays the fulfilled orders and submit buttons for creating a scheduled order from My Account:

```
<dsp:droplet name="OrderLookup">
   <dsp:param name="userId" bean="Profile.repositoryid"/>
   <dsp:param name="state" value="closed"/>
```

```
   <dsp:oparam name="output">
      <dsp:setvalue param="result" paramvalue="result"/>
   </dsp:oparam>
</dsp:droplet>

<dsp:droplet name="IsNull">
   <dsp:param name="value" param="prototypeOrderId"/>
   <dsp:oparam name="true">
      <dsp:droplet name="OrderLookup">
         <dsp:param name="userId" bean="Profile.repositoryId"/>
         <dsp:param name="state" value="closed"/>
            <dsp:oparam name="output">
               <dsp:form action="scheduled_order_new.jsp" method="post">

<table border=0 cellpadding=3 width=100%>
   <tr>
      <td colspan=2>
         <dsp:droplet name="IsNull">
            <dsp:param bean="ScheduledOrderFormHandler.repositoryId"
               name="value"/>
            <dsp:oparam name="true"><span class=small>Create scheduled orders
               based on your <dsp:a href="orders_filled.jsp">fulfilled
               orders</dsp:a>. Scheduled orders will have the same shipping and
               billing information as the fulfilled order. <p>You can also create
               a scheduled order from a current order by selecting "create
               scheduled order" at the end of the checkout process.</span>
            </dsp:oparam>
            <dsp:oparam name="false"><span class=small>You can edit the name of
               your order, schedule type, and order placement information.</span>
            </dsp:oparam>
         </dsp:droplet>
      </td>
   </tr>
   <tr><td><dsp:img src="../images/d.gif"/></td></tr>
   <tr valign=bottom>
      <td valign="top" align=right><span class=smallb>Name</span></td>
      <td><dsp:input bean="ScheduledOrderFormHandler.value.name" name="name"
         size="35" type="text"/><br>
      <span class=help>Enter a name for your scheduled order (i.e. Weekly Spark
         Plug Assortment).</span></td>
   </tr>
   <tr>
      <td align=right><span class=smallb>Schedule type</span></td>
      <td align="left">
         <dsp:select bean="ScheduledOrderFormHandler.moveToMode" name="select"
            size="1">
            <dsp:option value=""/>Choose schedule
            <dsp:option value="Daily"/>daily
            <dsp:option value="Weekly"/>weekly
            <dsp:option value="Monthly"/>monthly
```

```
         </dsp:select>
      </td>
   </tr>

<dsp:droplet name="IsNull">
   <dsp:param bean="ScheduledOrderFormHandler.repositoryId" name="value"/>
   <dsp:oparam name="true">
   <tr>
      <td align="right" valign=top><span class=smallb>Order</span></td>
      <td>
         <dsp:select bean="ScheduledOrderFormHandler.complexScheduledOrderMap.
            templateOrderId.userInputFields.orderId">
            <dsp:droplet name="ForEach">
               <dsp:param name="array" param="result"/>
               <dsp:oparam name="outputStart">
                  <dsp:option value=""/>  Select a fulfilled order
               </dsp:oparam>
               <dsp:oparam name="output">
                  <dsp:getvalueof id="option384" param="element.id"
                     idtype="java.lang.String">
                     <dsp:option value="<%=option384%>"/>
                  </dsp:getvalueof><dsp:valueof param="element.id"/>
               </dsp:oparam>
            </dsp:droplet>
         </dsp:select><br>
      </td>
   </tr>
   </dsp:oparam>
   <dsp:oparam name="false">
   </dsp:oparam>
</dsp:droplet>

   <tr><td><dsp:img src="../images/d.gif"/></td></tr>
   <tr>
      <td></td>
      <input name="createNew" type="hidden" value='<dsp:valueof
         param="createNew"/>'>
      <dsp:input bean="ScheduledOrderFormHandler.moveToErrorURL" type="hidden"
         value="scheduled_order_new.jsp"/>
      <dsp:input bean="ScheduledOrderFormHandler.moveToSuccessURL" type="hidden"
         value="scheduled_order_calendar.jsp?createNew=new"/>
      <td><dsp:input bean="ScheduledOrderFormHandler.moveToURL" type="submit"
         value="Continue"/></td>
   </tr>
   <!-- vertical space -->
   <tr><td><dsp:img vspace="0" src="../images/d.gif"/></td></tr>
</table>
</dsp:form>
</dsp:oparam>
<dsp:oparam name="empty">
```

```
<dsp:param name="value" param="result"/>
<dsp:oparam name="true">
   You have no <dsp:a href="orders_filled.jsp">fulfilled orders</dsp:a> at this
   time to create a scheduled order.  <p>You can create a scheduled order from a
   current order by selecting "create scheduled order" at the end of the checkout
   process.
</dsp:oparam>
   You have no <dsp:a href="orders_filled.jsp">fulfilled orders</dsp:a> at this
   time to create a scheduled order.  <p>You can create a scheduled order from a
   current order by selecting "create scheduled order" at the end of the checkout
   process.

</dsp:oparam>
</dsp:droplet><!--end OrderLookup-->
```

As shown above, we used the `OrderLookup` component to fetch all the user's orders with the order state "closed" and display them so that the user can select one. After the user chooses the schedule type, scheduled order name, and the order, he or she is directed to `schedule_order_calendar.jsp` to choose the calendar. The rest of the process is identical to creating a scheduled order from the checkout process.

Deleting a Scheduled Order

Users can delete a scheduled order by selecting it on the Scheduled Orders page in My Account and clicking the **Delete scheduled order** button at the end of the page. The user is directed to `scheduled_order_preview.jsp`, and the order ID of the selected order is passed as `scheduledOrderId`, which displays the order information, and provides options to edit or delete the order. The `scheduledOrderId` is assigned to `ScheduledOrderFormHandler.repositoryId`.

The JSP code from `scheduled_order_preview.jsp` to delete a scheduled order is:

```
<dsp:input bean="ScheduledOrderFormHandler.deleteSuccessURL" type="hidden"
   value="scheduled_orders.jsp"/>
<dsp:input bean="ScheduledOrderFormHandler.deleteErrorURL" type="hidden"
   value="scheduled_order_preview.jsp"/>
<dsp:input bean="ScheduledOrderFormHandler.delete" type="submit" value="Delete
   scheduled order"/>
```

Updating a Scheduled Order

Users can select scheduled orders on the Scheduled Orders page in My Account and click the **Edit order info** link. They can then change any information for that scheduled order. After making the necessary changes, they click the **Update** button. This process is similar to that of creating a new scheduled order, except that we display an option to update the order and the existing scheduled order information.

First, we used the following JSP code from `scheduled_order_calendar` to verify the validity of the user inputs:

```
<dsp:input bean="ScheduledOrderFormHandler.verifySuccessURL" type="hidden"
   value="scheduled_order_preview.jsp?source=scheduledOrder&createNew=new"/>
<dsp:input bean="ScheduledOrderFormHandler.verifyErrorURL" type="hidden"
   value="scheduled_order_calendar.jsp"/>
<dsp:input bean="ScheduledOrderFormHandler.verify" type="submit"
   value="Continue"/>
```

We used this JSP code snippet to update the scheduled order:

```
<dsp:input bean="ScheduledOrderFormHandler.updateSuccessURL" type="hidden"
   value="scheduled_orders.jsp"/>
<dsp:input bean="ScheduledOrderFormHandler.updateErrorURL" type="hidden"
   value="scheduled_order_calendar.jsp"/>
<dsp:input bean="ScheduledOrderFormHandler.update" type="submit"
   value="Update"/><p>
<span class=smallb><dsp:a href="scheduled_orders.jsp">Return to scheduled
   orders</dsp:a></span>
```

The verification is done when the user clicks the **Preview** button. If any invalid data are found, such as a missing order name, an error message is displayed at the top of the screen.

ATG Commerce has a back-end service called `ScheduledOrderService` that polls the Order Repository at a periodic interval and submits scheduled Orders according to their schedules. It has a property called `schedule` that defines how often to run the `placeScheduledOrders` task. Because Motorprise is a demo, we wanted the scheduled orders to be checked often, so we set the `schedule` property in `<ATG10dir>\MotorpriseJSP\j2ee-apps\motorprise\config\atg\commerce\order\scheduled\ScheduledOrderService.properties` to "every 10 seconds" so that you can see scheduled order functionality immediately.

For more information, see the Scheduling Recurring Orders section of the *Configuring Purchase Process Services* chapter of the *ATG Commerce Programming Guide*.

Saving Orders

Buyers can save selected orders from one session to the next. This feature is comparable to the concept of having multiple saved shopping carts.

`checkout/cart.jsp` and `checkout/save_order.jsp` allow users to save orders under a description they provide.

On the Current Order page (`checkout/cart.jsp`), the buyer can check out, cancel the order, or save the order. If he clicks the **Save Order** button, he is taken to `save_order.jsp`, where he is asked to provide a description of the saved order. If the user does not provide a name, the order is saved with the current date/time stamp, according to the user's locale.

We used /atg/commerce/order/purchase/SaveOrderFormHandler to save the user's current order based on the name that the user specifies. Each saved order for a specific buyer must have a unique name; an error is displayed if the buyer enters a name that already exists on a saved order in their saved order list. We used the following block of JSP code in user/save_order.jsp to provide this:

```
<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" type="submit" value="Save
      order"/>
   <dsp:input bean="SaveOrderFormHandler.saveOrder" type="hidden" value="save"/>
   <dsp:input bean="SaveOrderFormHandler.saveOrderSuccessURL" type="hidden"
      value="../user/saved_orders.jsp"/>
   <dsp:input bean="SaveOrderFormHandler.saveOrderErrorURL" type="hidden"
     value="../user/save_order.jsp"/>
```

Users have a link for their saved orders on the My Account page (order_management.jsp). They can browse through a list their of saved orders by clicking this link, which takes them to saved_orders.jsp. Once on this page, users can choose any saved order as their current order.

The code to display the list of saved orders is:

```
<dsp:droplet name="IsEmpty">
  <dsp:param name="value" bean="ShoppingCart.saved"/>
  <dsp:oparam name="true">
    <tr><td>You have no saved orders.</td></tr>
  </dsp:oparam>
  <dsp:oparam name="false">
    <tr valign=top>
      <td>
      <table border=0 cellpadding=4 width=100%>
        <tr>
          <td><b><span class=small> Order name</span></b></td>
          <td>  </td>
          <td><b><span class=small> Order #</span></b></td>
          <td>  </td>
          <td><b><span class=small>Date saved</span></b></td>
        </tr>
        <dsp:droplet name="ForEach">
          <dsp:param name="array" bean="ShoppingCart.saved"/>
          <dsp:oparam name="output">
            <tr>
              <td><dsp:a href="saved_order.jsp"><dsp:param name="orderId"
                param="element.id"/><dsp:valueof
                param="element.description"/></dsp:a></td>
              <td></td>
              <td><dsp:valueof  param="element.id"/></td>
```

```
              <td></td>
              <td><dsp:valueof date="MMMM d, yyyy h:mm a" param=
               "element.creationDate"/></td>
            </tr>
          </dsp:oparam>
        </dsp:droplet>
      </table>
      </td>
    </tr>
  </dsp:oparam>
</dsp:droplet><!--end IsEmpty-->
```

All the saved orders of the user are stored in saved array property of the Shopping Cart. We iterate through it displaying each saved order with a link, using which an user can access the saved order and he can choose it as his current Shopping Cart. The following code from `saved_order.jsp` demonstrates this:

```
<td colspan=4 align=right>
   <dsp:form action="../checkout/cart.jsp" method="post">
   <dsp:input bean="ShoppingCart.handlerOrderId" paramvalue="order.id"
      type="hidden"/>
   <dsp:input bean="ShoppingCart.switch" type="submit" value="Make this your
      current order"/>  
   </dsp:form>
</td>
<td colspan=2 align=left>
   <dsp:form action="../user/saved_orders.jsp" method="post">
   <dsp:input bean="ShoppingCart.handlerOrderId" paramvalue="order.id"
      type="hidden"/>
   <dsp:input bean="ShoppingCart.delete" type="submit" value="Delete"/>  
   </dsp:form>
```

Once a saved order is made the current order, it is removed from the saved orders list.

Note: saved orders are different than purchase lists. Users create purchase lists while navigating the catalog. While they are browsing the catalog, they can add any product to their purchase lists or create new purchase lists and add items to them. Purchase lists are completely editable; users can add or delete items from them and they can delete the entire purchase lists. Purchase lists can be used again and again. When a purchase list is added to the current order, all the items in purchase list are actually added to the order. The purchase list itself is not removed after adding it to the order. Saved orders, however, can only be used once they are made the current order. We actually put the saved order into the checkout process. It becomes the current order and then no longer exists as a saved order.

Approving Orders

Motorprise demonstrates how you can create your own custom approval conditions using ATG Business Commerce. Two steps are involved in the approval verification process, which occurs when the user submits the order for confirmation:

1. We check the `approvalRequired` property of the user placing the order to determine whether the user should go through the entire approval verification process.

   For example, orders from Louis Veloso, buyer for NDAP, do not require approval so we don't check for the approval conditions for orders he places. However, Nicole Hsu of USMW – US Division - Wholesale does require approvals for some orders. Her `approvalRequired` property is `true`, so we check for approval conditions for any orders she places.

2. If the `approvalRequired` property is `true` for the user, then we check all the approval conditions for any order placed by that user. You can customize approval behavior by creating your own specific approval conditions. We created one such condition for Motorprise that restricts the maximum amount of the orders that can be placed.

   For example, Tim Hartwell of USMW has an order price limit of $10,000. If Tim places an order whose total is more than $10,000, then the order requires approval from Mary Granger, the approver for his division at USMW.

Approval Condition

By default, in ATG Business Commerce, the `approvalRequired` property of the user is checked. If it is true, then all the orders placed by the user are set to the "approval required" state. We overrode this behavior in Motorprise by adding an extra condition that checks the `orderPriceLimit` property of the user if his or her `approvalRequired` property is true.

We implemented this new approval condition for Motorprise by creating a new component, `/atg/commerce/approval/processor/CheckOrderLimitApprovalRequirements`, based on the class `atg.commerce.expression.ProcPropertyRestriction`. This component checks to see whether the total amount of the order placed by the user is greater than the `orderPriceLimit` of the user.

For more information on the `ProcPropertyRestriction` class, see the *Setting Restrictions on Orders* section of the *Configuring Purchase Process Services* chapter of the *ATG Commerce Programming Guide*.

The properties of `CheckOrderLimitApprovalRequirements` are:

```
$class=atg.commerce.expression.ProcPropertyRestriction
$description=This component creates new approval rule based on Order.
$scope=global
errorMessage=Order's total is greater than the approved order limit of user.
expressionParser=/atg/commerce/util/ExpressionParser
pipelineResultErrorMessageKey=checkOrderLimitApprovalRequirements
returnValueForFalseEvaluation=1
returnValueForTrueEvaluation=0
```

```
ruleEvaluator=/atg/commerce/util/RuleEvaluator
ruleExpression=Order.priceInfo.amount > Profile.orderPriceLimit
```

The `ruleExpression` property defines the condition for approval. For Motorprise, this rule determines if the order total of any order created by the user, `Order.priceInfo.amount`, exceeds his approval limit, `Profile.orderPriceLimit`. If the order placed by the user exceeds their order limit, then the order is marked as requiring approval.

The `returnValueForTrueEvaluation` and `returnValueForFalseEvaluation` properties define what should be returned by this component if `ruleExpression` is true or false. Later, in the approval pipeline we use these return values to determine which pipeline path to take depending on the approval condition.

We also created another approval condition that checks for requisition numbers in invoice payment methods. Users can enter either a P.O. number or requisition number when they choose invoice as their payment method. If the user uses a requisition number, then the order is placed in an "approval required" state if his or her `approvalRequired` property is set to true. For more information, see the *Payment Information* section of the Processing Orders chapter.

We implemented this approval condition for Motorprise by creating a new component, `/atg/commerce/approval/processor/CheckRequisitionNumbers`, based on a new pipeline processor `atg.projects.b2bstore.approval.ProcCheckRequisitionNumbers`. This pipeline processor returns one of two values, depending on whether or not an order contains any payment group with an associated requisition number. For more information, please refer to the API documentation for this class.

The properties of `CheckRequisitionNumbers` are:

```
$class=atg.projects.b2bstore.approval.ProcCheckRequisitionNumbers

# If requisitions were used, add an error to the pipeline result and
# return STOP_CHAIN_EXECUTION_AND_COMMIT so the approval pipeline will
# mark the order as requiring approval

requisitionUsedValue=0
requisitionUsedAddsPipelineError=true
requisitionUsedPipelineMessage=Requisition number found - order requires approval

# Otherwise just return a value we can use in a transition link to
# proceed to the next test for possible approval conditions.

requisitionNotUsedValue=1
```

Once we have the approval condition, we have to include it in the `approveOrder` pipeline chain, which is invoked when the order is submitted for checkout. Most of the approval pipeline framework has been developed and configured in the core ATG Business Commerce layer; we just have to include our new custom approval condition in the approval pipeline framework. In ATG Business Commerce, the `approveOrder` pipeline chain is invoked during the order confirmation phase, which in turn invokes the

checkRequiresApproval chain to verify the order for the approval conditions. The following is the relevant approval pipeline code in <ATG10dir>/B2BCommerce/config/atg/commerce/approval/approvalPipeline.xml

```
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<!-- PipelineChain used to determine if a users order requires    -->
<!-- approval or not.                                             -->
<!-- ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ -->
<pipelinechain name="checkRequiresApproval" transaction="TX_REQUIRES_NEW"
   headlink="checkProfileApprovalRequirements">
   <pipelinelink name="checkProfileApprovalRequirements"
      transaction="TX_MANDATORY">
      <processor jndi="/atg/commerce/approval/processor/
         CheckProfileApprovalRequirements"/>
    </pipelinelink>
</pipelinechain>
```

As shown above, the checkRequiredApproval chain determines whether or not the order requires approval. We overrode this chain in the Motorprise layer to include our new approval condition in <ATG10dir>/Motorprise/config/atg/commerce/approval/approvalPipeline.xml as shown below:

```
<pipelinechain name="checkRequiresApproval"
               transaction="TX_REQUIRES_NEW"
               headlink="checkProfileApprovalRequirements"
               xml-combine="replace">
  <pipelinelink name="checkProfileApprovalRequirements"
                transaction="TX_MANDATORY">
    <processor
    jndi="/atg/commerce/approval/processor/CheckProfileApprovalRequirements"/>
   <transition returnvalue="1"
               link="checkOrderLimitApprovalRequirements"/>
  </pipelinelink>
  <pipelinelink name="checkOrderLimitApprovalRequirements"
                transaction="TX_MANDATORY">
    <processor
    jndi="/atg/commerce/approval/processor/CheckOrderLimitApprovalRequirements"/>
  </pipelinelink>
</pipelinechain>
```

```
<pipelinechain name="checkRequiresApproval" transaction="TX_REQUIRES_NEW"
   headlink="checkProfileApprovalRequirements" xml-combine="replace">
      <pipelinelink name="checkProfileApprovalRequirements" transaction=
        "TX_MANDATORY">
      <processor jndi="/atg/commerce/approval/processor/
         CheckProfileApprovalRequirements"/>
```

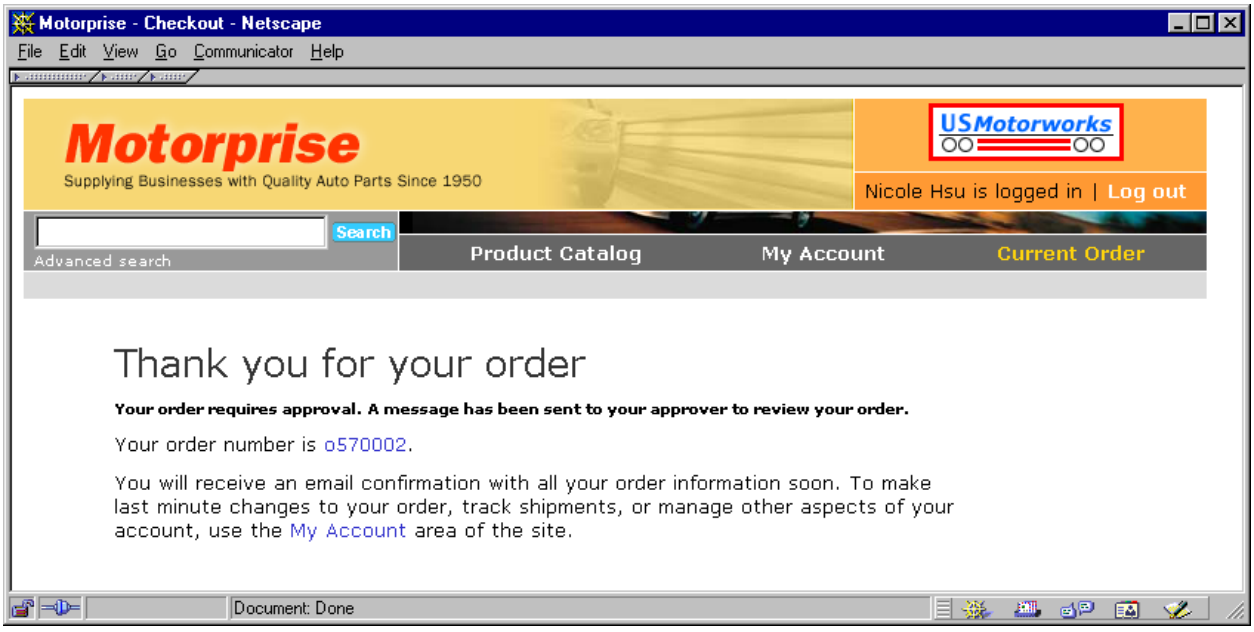
```
            <transition returnvalue="1" link="checkRequisitionNumbers"/>
      </pipelinelink>

      <pipelinelink name="checkRequisitionNumbers" transaction="TX_MANDATORY">
         <processor jndi="/atg/commerce/approval/processor/
            CheckRequisitionNumbers"/>
         <transition returnvalue="1" link="checkOrderLimitApprovalRequirements"/>
      </pipelinelink>

      <pipelinelink name="checkOrderLimitApprovalRequirements" transaction=
         "TX_MANDATORY">
         <processor jndi="/atg/commerce/approval/processor/
            CheckOrderLimitApprovalRequirements"/>
      </pipelinelink>
</pipelinechain>
```

The first chain in `approvalPipeline.xml`, `checkRequiresApproval`, is invoked during the checkout process and the new approval conditions, `checkRequisitionNumbers` and `checkOrderLimitApprovalRequirements`, are invoked in this chain. First, the chain checks to see if the user requires approval for orders; if he does, then the `checkRequisitionNumbers` condition is checked. If the order uses a requisition number in the invoice payment method, the pipeline is stopped and the order is marked as requiring approval. If the order doesn't use a requisition number, then the control goes to `checkOrderLimitApprovalRequirements`, which checks whether the order price is above or below the user's approved order limit. If the order requires an approval, then a message is displayed to the user on the order confirmation page, `thank_you.jsp`, indicating that the order requires approval.

The following is the code in `thank_you.jsp` to determine whether the order requires approval or not:

```
<dsp:droplet name="Switch">
  <dsp:param bean="ShoppingCart.last.state" name="value"/>
  <dsp:oparam name="5000">
    <tr>
      <td><span class=smallb>Your order requires approval. A message has been sent
         to your approver to review your order.</span><p></td>
    </tr>
  </dsp:oparam>
  <dsp:oparam name="default">
  </dsp:oparam>
</dsp:droplet>
```

Whenever the user submits an order, it is moved into `ShoppingCart.last`, so that its state property is checked to determine whether it requires approval or not as shown above.



The order confirmation page, `thank_you.jsp`, displays a message indicating that the order requires approval

We also created another approval condition that checks for requisition numbers in invoice payment methods. Users can enter either PO number or requisition number when they choose invoice as their payment method. If the user chooses requisition number, then the order is placed in an "approval required" state if his or her `approvalRequired` property is set to true. For more information, see the *Payment Information* section of the Processing Orders chapter.

Approver Information

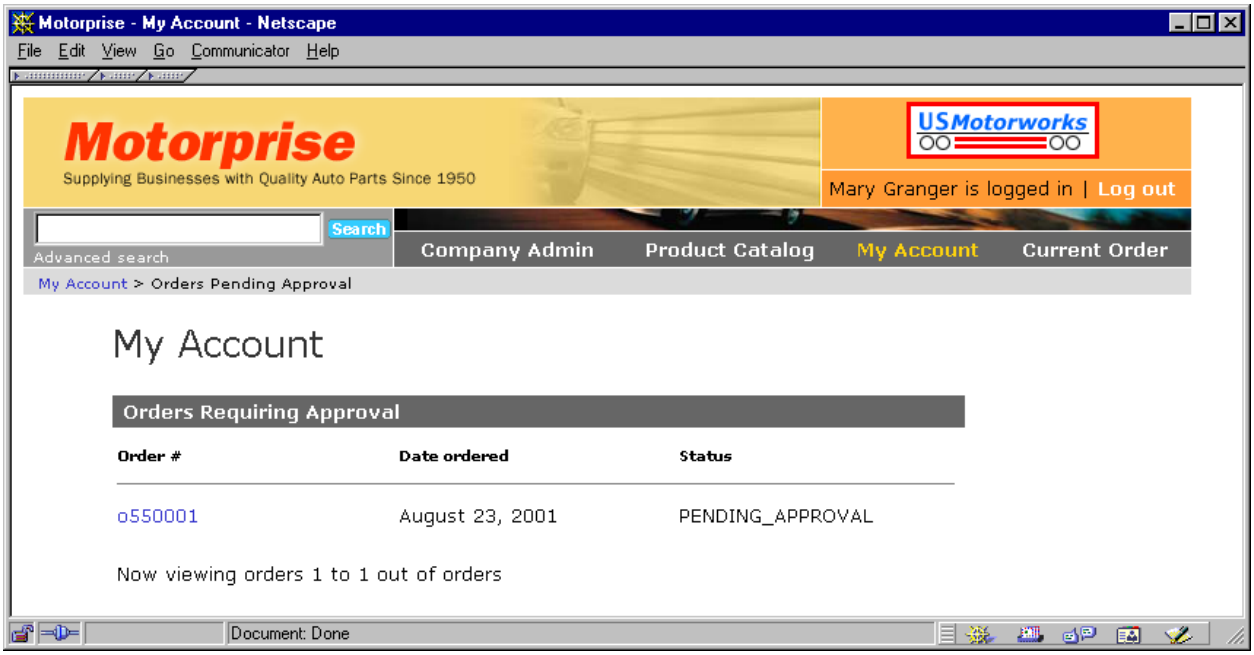
Once an order is placed that requires approval, an e-mail is sent to the person who approves that buyer's orders. Also, the order is displayed on the approver's approvals page, `user/approvals.jsp`. Clicking any order in that list takes him to a screen where he can reject or approve the order.

Approvers can see the orders that are to be approved on the Motorprise home page and also from the My Account section by clicking the Approval Requests link (`user/approvals.jsp`). The following is the relevant JSP code in `approvals.jsp` that displays the pending approval requests.

```
<%/* Pass profile id and state values to ApprovalRequiredDroplet to get all
   the orders*/%>

<dsp:droplet name="ApprovalRequiredDroplet">
   <dsp:param bean="Profile.id" name="approverid"/>
   <dsp:param name="state" value="open"/>
   <dsp:param name="startIndex" param="startIndex"/>
   <dsp:oparam name="output">
```

```
   <tr valign=top>
      <td width=33%><span class="small"><b>Order #</b></span></td>
      <td width=33%><span class="small"><b>Date ordered</b></span></td>
      <td width=33%><span class="small"><b>Status</b></span></td>
   </tr>
   <tr>
      <td colspan=3><hr size=1 color="#666666"></td>
   </tr>
   <%/* Iterate through each order and display order info */%>

   <dsp:droplet name="ForEach">
      <dsp:param name="array" param="result"/>
      <dsp:oparam name="output">
   <tr>
      <td><dsp:a href="order_pending_approval.jsp">
         <dsp:param name="orderId" param="element.id"/>
         <dsp:valueof param="element.id"/></dsp:a>
      </td>
      <td><dsp:valueof date="MMMMM d, yyyy"
         param="element.submittedDate"/></td>
      <td><dsp:valueof param="element.state"/></td>
   </tr>
</dsp:oparam>
<dsp:oparam name="outputEnd">
```

We used the `ApprovalRequiredDroplet` to get the orders with pending approvals for the given `approverid`. Because we want the orders that are pending, we pass the value `open` to the `state` parameter to fetch the orders. The `ApprovalRequiredDroplet` returns an array of orders in the `result` parameter, on which we iterate using a `ForEach` droplet getting the individual orders and providing a link to access each order.



Orders that require approval are displayed in the My Account section of the approver.

On the approvals page, the approver can click on the order number that takes him or her to `order_pending_approval.jsp`, which displays the order information and options to approve or reject the order.

If the approver chooses the Approve Order link, he or she is taken to `approve_order.jsp` to optionally enter a message for the user that will appear on the order and to approve the order and submit it to be fulfilled. The following is the relevant JSP code in `approve_order.jsp`:

```
<%/* Set the order id of of approvalformahandler to the order to be approved and
   provide submit buttons for approve/cancel. */%>
<dsp:form action="approvals.jsp" method="post">
   <dsp:input bean="ApprovalFormHandler.orderId" paramvalue="orderId"
      type="hidden"/>
   <span class=help>Enter a message.</span><br>
   <dsp:textarea bean="ApprovalFormHandler.approverMessage" rows="7"
      cols="50"></dsp:textarea><p>
   <%/* page to be redirected if the order approval is successful */%>
   <dsp:input bean="ApprovalFormHandler.approveOrderSuccessURL" type="hidden"
      value="approve_confirm.jsp"/>
   <%/* page to be redirected if the order approval fails */%>
   <dsp:input bean="ApprovalFormHandler.approveOrderErrorURL" type="hidden"
      value="approve_order.jsp"/>
   <dsp:input bean="ApprovalFormHandler.approveOrder" type="submit" value="Approve
      order"/>  
```

```
   <input type="submit" value="Cancel">  
</dsp:form>
```

If the approver clicks the Reject Order link, he or she is taken to `reject_order.jsp` to optionally enter a message for the user and to reject the order. The following is the relevant JSP code in `reject_order.jsp` to display this:

```
<dsp:form action="order_pending_approval.jsp" method="post">
   <dsp:input bean="ApprovalFormHandler.orderId" paramvalue="orderId"
      type="hidden"/>
   <span class=help>Enter a message.</span><br>
   <dsp:textarea bean="ApprovalFormHandler.approverMessage" rows="7"
      cols="50"></dsp:textarea><p>
   <input name="orderId" type="hidden" value="<dsp:valueof param="orderId"/>">
   <dsp:input bean="ApprovalFormHandler.rejectOrderSuccessURL" type="hidden"
      value="reject_confirm.jsp"/>
   <dsp:input bean="ApprovalFormHandler.rejectOrderErrorURL" type="hidden"
      value="reject_order.jsp"/>
   <dsp:input bean="ApprovalFormHandler.rejectOrder" type="submit" value=
      "Reject order"/>  
   <input type="submit" value="Cancel">  
</dsp:form>
```

When the approver approves or rejects the order, an e-mail is sent to the buyer updating him of that order's status. The buyer also can see the status of his orders in the My Account section on the Open Orders page, `en/user/orders_open.jsp`.

Whenever order is approved or rejected, the Approval Update event is fired with the `approvalStatus` property of the message set to `approved` or `rejected`. In order to send an e-mail when the order is approved or rejected, we created a scenario, ApprovalNotification, that listens to this event and, depending on the `approvalStatus`, sends an approved or rejected e-mail to the customer. (For more information on the ApprovalNotification, see the Merchandising chapter in this guide.)

When the order is approved, it is automatically submitted and the status of the order displays as "submitted" in the buyer's order history. In the approver's My Account section, the order is moved to the "approved orders" view. If the order is rejected, the order will not be submitted and appears in the buyer's order history with the status "rejected."

You can create similar approval conditions and plug them into the `checkRequiresApproval` pipeline chain in the same way.

For more information on the approval process, see the *ATG Commerce Programming Guide*.

7 Displaying and Accessing the Product Catalog

This chapter describes how the catalog pages are displayed to customers and how customers navigate the catalog.

Product Template Pages
Describes how to create product template pages for displaying products and SKUs in the catalog. Detailed code examples are shown to help you understand how we implemented this in Motorprise.

Category Template Pages
Describes how to create category template pages for displaying categories in the product catalog.

Searching
Describes how to add search features to your pages.

Comparing
Describes how to add side-by-side comparison to your pages.

Product Template Pages

In order to allow users to browse through your catalog of products, you need to create pages that display information about the various categories and products you sell. You could create a separate JSP for every single product, but this would be inefficient. Each time you create a new product or remove one from your catalog, you would have to make modifications.

The catalog is organized by product type. Users can navigate the catalog by drilling down through the various categories. For Motorprise, we created generic template pages for product categories and for products. These templates are simply JSPs that display general information about one category or one product.

The templates are general and data-driven, rather than being hard-coded for a specific product or category. All of the information they display comes from the repository. These general, dynamic pages are convenient because, once they are written, you can use them to display any category or product in a product catalog. For example, a very simple site could have one category template page file and one product template page file. Because these pages are dynamic, they are easy to maintain. As you add more products to your repository, you don't necessarily have to change the catalog template pages.

We created a general product template page in Motorprise, `MotorpriseJSP/j2ee-apps/Motorprise/web-app/en/catalog/product.jSP`, to display any product. It takes the ID of the product and displays all of its relevant information using different components and page templates. The following information is available for each product:

- Part name
- Part number
- Manufacturer
- Description
- Unit of Measure (UOM)
- Regular Price
- Dynamic price (for this user if different from regular price)
- Picture of item (this is only included for some products)
- Discounts for quantity purchased
- Inventory status (and inventory level for enterprise customers)

Using Product Lookup

We used `/atg/commerce/catalog/ProductLookup`, a component that is an instance of `atg.commerce.catalog.custom.CatalogItemLookupDroplet` class, to fetch product attributes from the repository. This is the beginning of the code from `product.jsp` where we used the `ProductLookup` component:

```
<dsp:droplet name="/atg/commerce/catalog/ProductLookup">
  <dsp:oparam name="output">
…
<table border=0 cellpadding=4>
<tr>
   <td>
      <span class=categoryhead>
      <dsp:valueof param="element.displayName">No name</dsp:valueof></span>
      <br>
      <b><dsp:valueof param="element.description"/></b></td>
   </td>
</tr>
<tr valign=top>
   <td>
      <dsp:include page="../common/FormError.jsp" flush="false"></dsp:include>
      <dsp:droplet name="IsEmpty">
         <dsp:param name="value" param="element.largeImage.url"/>
         <dsp:oparam name="false">
      <dsp:getvalueof id="imageURL" param="element.largeImage.url"
         idtype="java.lang.String">
         <dsp:img hspace="70" alt="Product image" src="<%=imageURL%>"/>
```

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

      <dsp:getvalueof id="pval0" param="element"><dsp:include
         page="SKUProperties.jsp" flush="false"><dsp:param name="product"
         value="<%=pval0%>"/></dsp:include></dsp:getvalueof>
      <br>
      <span class=smallb>Product Description</span><br>
      <span class=small><dsp:valueof param="element.longDescription">
         No description</dsp:valueof>
      </span>
   </td>
</tr>
</table>
…
</dsp:oparam>
</dsp:droplet>
```

`ProductLookup` takes an `id` parameter as input; in this case, it is the repository ID of the product, from the product link the user clicked on the category page. It then binds the `element` parameter to the product with the `id` that was passed in. In this code example, we didn't explicitly pass the `id` parameter to the `ProductLookup` component. The page-level `id` parameter was already defined at the top of the page as an input parameter and thus implicitly passed to `ProductLookup`.

The following code shows how we displayed the product's `longDescription` property, or, a default value if the product does not have a `longDescription` value:

```
<dsp:valueof param="element.longDescription">No description</dsp:valueof>
```

The following code example shows how we displayed the product's image if it is not null.

```
<dsp:droplet name="IsEmpty">
   <dsp:param name="value" param="element.largeImage.url"/>
   <dsp:oparam name="false">
      <dsp:getvalueof id="imageURL" param="element.largeImage.url"
         idtype="java.lang.String">
      <dsp:img hspace="70" alt="Product image" src="<%=imageURL%>"/>
      </dsp:getvalueof>
   </dsp:oparam>
</dsp:droplet>
```

SKU Information

The Motorprise catalog has a single SKU per product. We display the SKU for each product using `SKUProperties.jsp`, which is invoked from `product.jsp`, passing the `product` as parameter. It displays attributes of each SKU such as part number, manufacturer, availability status, and price. The following JSP code is used to display this information.

```
<dsp:droplet name="ForEach">
  <dsp:param name="array" param="product.childSKUs"/>
  <dsp:oparam name="output">
    <table cellspacing="0" cellpadding="0" border="0">
      <tr>
        <td> </td>
        <td><span class=smallb>Part Number:</span></td>
        <td> </td>
        <td><span class=small><dsp:valueof param=
           "element.manufacturer_part_number">No id</dsp:valueof></span>
        <td> </td>
      </tr>
      <tr>
        <td> </td>
        <td><span class=smallb>Manufacturer:</span></td>
        <td> </td>
        <td><span class=small><dsp:valueof param=
           "product.manufacturer.displayName">Unknown</dsp:valueof>
  <td> </td>
      </tr>

      <dsp:droplet name="/atg/dynamo/droplet/Switch">
        <dsp:param bean="Profile.transient" name="value"/>
        <dsp:oparam name="false">
          <tr>
            <td> </td>
            <td><span class=smallb>Availability:</span></td>
            <td> </td>
            <dsp:droplet name="/atg/commerce/inventory/InventoryLookup">
              <dsp:param name="itemId" param="element.repositoryId"/>
              <dsp:oparam name="output">
                <td><span class=small><dsp:valueof param=
                  "inventoryInfo.availabilityStatusMsg">
                  Unknown</dsp:valueof></span>
              </dsp:oparam>
            </dsp:droplet>
            <td> </td>
          </tr>

          <dsp:droplet name="/atg/dynamo/droplet/Switch">
            <dsp:param bean="Profile.parentOrganization.customerType"
               name="value"/>
             <dsp:oparam name="Enterprise">
               <tr>
                 <td> </td>
                 <td><span class=smallb>Stock Level:</span></td>
                 <td> </td>
                 <dsp:droplet name="/atg/commerce/inventory/InventoryLookup">
                   <dsp:param name="itemId" param="element.repositoryId"/>
```

```
                    <dsp:oparam name="output">
                      <td><span class=small><dsp:valueof
                         param="inventoryInfo.stockLevel">
                         Unknown</dsp:valueof></span>
                    </dsp:oparam>
                  </dsp:droplet>
                  <td> </td>
                </tr>
              </dsp:oparam>
            </dsp:droplet>
            <dsp:getvalueof id="pval0" param="product"><dsp:getvalueof id="pval1"
               param="element"><dsp:include page="DisplayPrice.jsp"
               flush="true"><dsp:param name="Product"
               value="<%=pval0%>"/><dsp:param name="Sku"
               value="<%=pval1%>"/></dsp:include></dsp:getvalueof></dsp:getvalueof>
        </dsp:oparam>
      </dsp:droplet>
    </table>
  </dsp:oparam>
</dsp:droplet>
```

Inventory Information

We used `InventoryLookup` to show the availability status (such as "in stock" or "out of stock") of the product to all registered users. In addition, if the buyer belongs to an enterprise customer such as USMW, we also display the actual amount in stock. Anonymous or guest users do not see the inventory information or price. The following code from `catalog/SKUProperties.jsp` is used to display this information.

```
<dsp:droplet name="/atg/dynamo/droplet/Switch">
        <dsp:param bean="Profile.transient" name="value"/>
        <dsp:oparam name="false">
          <tr>
            <td> </td>
            <td><span class=smallb>Availability:</span></td>
            <td> </td>
            <dsp:droplet name="/atg/commerce/inventory/InventoryLookup">
              <dsp:param name="itemId" param="element.repositoryId"/>
              <dsp:oparam name="output">
                <td><span class=small><dsp:valueof param=
                  "inventoryInfo.availabilityStatusMsg">
                  Unknown</dsp:valueof></span>
              </dsp:oparam>
            </dsp:droplet>
            <td> </td>
          </tr>

          <dsp:droplet name="/atg/dynamo/droplet/Switch">
```

```
        <dsp:param bean="Profile.parentOrganization.customerType"
          name="value"/>
         <dsp:oparam name="Enterprise">
           <tr>
             <td> </td>
             <td><span class=smallb>Stock Level:</span></td>
             <td> </td>
             <dsp:droplet name="/atg/commerce/inventory/InventoryLookup">
               <dsp:param name="itemId" param="element.repositoryId"/>
               <dsp:oparam name="output">
                 <td><span class=small><dsp:valueof param=
                   "inventoryInfo.stockLevel">Unknown</dsp:valueof></span>
               </dsp:oparam>
             </dsp:droplet>
             <td> </td>
           </tr>
         </dsp:oparam>
       </dsp:droplet>
       <dsp:getvalueof id="pval0" param="product"><dsp:getvalueof id="pval1"
         param="element"><dsp:include page="DisplayPrice.jsp"
         flush="true"><dsp:param name="Product" value="<%=pval0%>"/><dsp:param
         name="Sku" value="<%=pval1%>"/>
         </dsp:include></dsp:getvalueof></dsp:getvalueof>
    </dsp:oparam>
  </dsp:droplet>
```

Displaying Product Prices

Motorprise uses price lists to maintain the prices of different products and SKUs. Price lists allow you to target a specific set of prices to a specific group of customers. We used two types of pricing in Motorprise: list and volume. A list price is the full, undiscounted price of a product. Volume pricing determines the price based on the number of items ordered. We used two types of complex volume pricing in Motorprise:

- **Bulk pricing:** the price of a product is calculated based on the minimum quantity that is ordered.
- **Tiered pricing:** the price of a product is calculated using a fixed quantity or weight at different pricing levels.

We used two page fragments to display prices on the product template pages in Motorprise: `MotorpriseJSP/j2ee-apps/Motorprise/web-app/en/catalog/DisplayPrice.jsp` and `MotorpriseJSP/j2ee-apps/Motorprise/web-app/en/common/DisplayComplexPrice.jsp`. `DisplayPrice.jsp` formats and displays list prices. `DisplayComplexPrice.jsp` displays the detailed information of complex prices such as the number of price levels and the minimum and maximum prices at each level.

`DisplayPrice.jsp` is invoked to display the price of the product. We used the `PriceDroplet` to determine the pricing scheme used for a given product and SKU. It takes SKU and product repository items as input and gives as an output the price repository items. The `price` parameter contains all pricing

attributes such as pricing scheme used, list price, and complex price. We used the following JSP fragment to display the price of the product:

```
<dsp:droplet name="PriceDroplet">
  <dsp:param name="product" param="Product"/>
  <dsp:param name="sku" param="Sku"/>
  <dsp:oparam name="output">
    <dsp:droplet name="Switch">
      <dsp:param name="value" param="price.pricingScheme"/>
      <dsp:oparam name="listPrice">
        <tr>
          <td> </td>
          <td><span class=smallb>Price:</span></td>
          <td>  </td>
          <td>
```

We used the `CurrencyConversionFormatter` droplet to format the list price correctly for its locale. Notice that `locale` (the locale that `price.listPrice` is defined in) and `targetLocale` (the locale that we will be displaying the price for) are the same. This means that no currency conversion actually takes place, only formatting. The result is formatted correctly for `Profile.pricelist.locale`. If the default currency is the euro, then we use the correct currency symbol by setting the `euroSymbol` property.

```
            <%/*Display price for locale of user: */%>
            <dsp:droplet name="CurrencyConversionFormatter">
                 <dsp:param name="currency" param="price.listPrice"/>
                 <dsp:param bean="Profile.priceList.locale" name="locale"/>
                 <dsp:param bean="Profile.priceList.locale" name="targetLocale"/>
                 <dsp:param name="euroSymbol" value="&euro;"/>
                 <dsp:oparam name="output">
                   <span class=small><dsp:valueof valueishtml="<%=true%>"
                      param="formattedCurrency">no price</dsp:valueof></span>
                 </dsp:oparam>
               </dsp:droplet>

            <%/*Switch to see whether user is of German locale */%>
            <dsp:droplet name="Switch">
                     <dsp:param bean="Profile.priceList.locale" name="value"/>
                     <dsp:oparam name="de_DE_EURO">

                     <%/*Display price in DM for German users: */%>
```

At this point, we display the same price as before, we need to display it correctly for the German user. In this case `locale` and `targetLocale` are not the same. This means that before the result is formatted for the target locale, the currency (`price.listPrice`) must be converted from the default currency of `de_DE_EURO` to the default currency of `de_DE`. We mathematically calculate the value based on the exchange rates stored in `/atg/droplet/ExchangeRates.properties`. The result is then formatted for the `de_DE` locale. (In this way, the price, which is stored in euros, is displayed in Deutsch Marks.)

```
                    <dsp:droplet name="CurrencyConversionFormatter">
                      <dsp:param name="currency" param="price.listPrice"/>
                      <dsp:param bean="Profile.priceList.locale" name="locale"/>
                      <dsp:param name="targetLocale" value="de_DE"/>
                      <dsp:oparam name="output">
                        <span class=small>(<dsp:valueof
                           param="formattedCurrency">no price
                           </dsp:valueof>)</span>
                      </dsp:oparam>
                    </dsp:droplet>
                    </dsp:oparam>
              </dsp:droplet>
            </td>
            <td> </td>
          </tr>
        </dsp:oparam>

        <dsp:oparam name="bulkPrice">
          <dsp:getvalueof id="pval0" param="price.complexPrice"><dsp:getvalueof
            id="pval1" param="price.pricingScheme"><dsp:include page=
            "DisplayComplexPrice.jsp" flush="true"><dsp:param name="complexPrice"
            value="<%=pval0%>"/><dsp:param name="pricingScheme"
            value="<%=pval1%>"/></dsp:include></dsp:getvalueof></dsp:getvalueof>
        </dsp:oparam>

        <dsp:oparam name="tieredPrice">
          <dsp:getvalueof id="pval0" param="price.complexPrice"><dsp:getvalueof
             id="pval1" param="price.pricingScheme"><dsp:include
             page="DisplayComplexPrice.jsp" flush="true"><dsp:param name=
             "complexPrice" value="<%=pval0%>"/><dsp:param name="pricingScheme"
             value="<%=pval1%>"/></dsp:include></dsp:getvalueof></dsp:getvalueof>
        </dsp:oparam>
      </dsp:droplet>

    </dsp:oparam>
    <dsp:oparam name="error">
      There was a pricing error.
    </dsp:oparam>

</dsp:droplet>
```

In the code sample above, if the pricing scheme is a list price, it is formatted and displayed; otherwise, `DisplayComplexPrice.jsp` is invoked to display the complex price.

We used `ComplexPriceDroplet` to display the detailed information of complex prices such as the number of price levels and the minimum and maximum prices at each level. The JSP code fragment below demonstrates how we iterate through each level of the complex price and display its pricing information:

```
<tr>
  <td> </td>
  <td valign=top><span class=smallb>Bulk Price:</span></td>
  <td> </td>
  <td>
    <table border=0 cellspacing=0 cellpadding=0 width=100%>

<dsp:droplet name="ComplexPriceDroplet">
  <dsp:param name="complexPrice" param="complexPrice"/>
  <dsp:oparam name="output">
    <dsp:droplet name="For">
      <dsp:param name="howMany" param="numLevels"/>
      <dsp:param name="indexName" value="index"/>
        <dsp:oparam name="output">
          <tr>
            <td><span class=small>
                <dsp:valueof param="levelMinimums[param:index]"/>
            </span></td>
            <td>
                 - 
            </td>
            <td><span class=small>
                <dsp:valueof param="levelMaximums[param:index]">or
                   over</dsp:valueof>
            </span></td>
            <td>

              <%/*Display price for locale of user: */%>
              <dsp:droplet name="CurrencyConversionFormatter">
                  <dsp:param name="currency" param="prices[param:index]"/>
                  <dsp:param bean="/atg/userprofiling/
                     Profile.priceList.locale" name="locale"/>
                  <dsp:param bean="/atg/userprofiling/
                     Profile.priceList.locale" name="targetLocale"/>
                  <dsp:param name="euroSymbol" value="&euro;"/>
                  <dsp:oparam name="output">
                      <span class=small><dsp:valueof
                       valueishtml="<%=true%>" param="formattedCurrency">no
                          price</dsp:valueof></span>
                  </dsp:oparam>
                </dsp:droplet>
                <dsp:droplet name="Switch">
                  <dsp:param bean="/atg/userprofiling/
                     Profile.priceList.locale" name="locale"/>
                  <dsp:oparam name="default">
                  </dsp:oparam>
                  <dsp:oparam name="de_DE">

                  <%/*Don't Display in en directory */%>
```

```
                    <%/*
                    <dsp:droplet name="CurrencyConversionFormatter">
                      <dsp:param name="currency" param="prices[param:index]"/>
                      <dsp:param bean="/atg/userprofiling/
                         Profile.priceList.locale" name="locale"/>
                      <dsp:param name="targetLocale" value="de_DE"/>
                      <dsp:oparam name="output">
                        <span class=small>(<dsp:valueof param=
                           "formattedCurrency">no price</dsp:valueof>)</span>
                      </dsp:oparam>
                    </dsp:droplet> */%>
                    </dsp:oparam>
                    </dsp:droplet>
                  </td>
                </tr>
              </dsp:oparam>
            </dsp:droplet>
          </dsp:oparam>
        </dsp:droplet>
      </table>
      </td>
    </tr>
 </dsp:oparam><!--  ********************************** End of Bulk Pricing -->
```

We used the same method to display tiered prices.

Currency Formatting

In the above code example, we used the `CurrencyConversionFormatter` droplet to format the currency according to the `targetLocale` parameter passed. It takes as inputs the currency to format, the locale of the user, `targetLocale`, and outputs the `formattedCurrency` parameter.

Add To Order

Once the product is displayed to the customer, she has an option on the product page to add a quantity of that item to her current order if she is a registered customer.

We used `CartModifierFormHandler` to add the product to the current order, as shown in the code below. The `product` parameter is passed to `AddToCart.jsp` to be used for adding the product.

```
<input name="id" type="hidden" value="<dsp:valueof param=
   "product.repositoryId"/>">
   <dsp:input bean="CartModifierFormHandler.addItemToOrderSuccessURL"
      type="hidden" value="../checkout/cart.jsp?noCrumbs=false"/>
   <dsp:input bean="CartModifierFormHandler.SessionExpirationURL" type="hidden"
      value="../common/session_expired.jsp"/>
   <dsp:input bean="CartModifierFormHandler.productId"
```

```
      paramvalue="product.repositoryId" type="hidden"/>
   <dsp:droplet name="/atg/dynamo/droplet/ForEach">
      <dsp:param name="array" param="product.childSKUs"/>
      <dsp:oparam name="output">
      <table border=0 cellpadding=3 width=100%>
         <tr>
         <td>
         <dsp:input bean="CartModifierFormHandler.catalogRefIds"
            paramvalue="element.repositoryId" type="hidden"/>
         <span class=smallb>Qty</span> 
         <dsp:input bean="CartModifierFormHandler.quantity" size="4" type="text"
            value="1"/>  
         <dsp:input bean="CartModifierFormHandler.addItemToOrder" type="submit"
            value="Add to order"/>
         <br>
         </td>
         </tr>
      </table>

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

If the item is successfully added to the order, the **Add to Order** button redirects the user to `/checkout/cart.jsp`. Otherwise, the user is redirected to the same page showing the errors. We iterate through all the SKUs providing the input option for the user for each SKU and adding it to `CartModifierFormHandler.catalogRefIds`.

Note: because Motorprise is a demo site, each product only has one SKU. However, we included the code to iterate through all the SKUs of a product so that you could easily apply this code if your catalog contains products with multiple SKUs apiece.

Add To List

If the customer doesn't want to purchase the product at the time, but wants to save the item for future purchase, he can save it in any of his existing purchase lists as shown in the following code fragment from `AddToList.jsp`:

```
<dsp:form action="product.jsp" method="post">
   <input name="id" type="hidden" value="<dsp:valueof param=
      "product.repositoryId"/>">
   <dsp:input bean="PurchaselistFormHandler.addItemToPurchaselistErrorURL"
      type="hidden" value="product.jsp"/>
   <dsp:input bean="PurchaselistFormHandler.productId"
      paramvalue="product.repositoryId" type="hidden"/>
   <dsp:droplet name="/atg/dynamo/droplet/ForEach">
      <dsp:param name="array" param="product.childSKUs"/>
      <dsp:oparam name="output">
         <table border=0 cellpadding=3 width=100%>
```

```
            <tr>
               <td><dsp:input bean="PurchaselistFormHandler.catalogRefIds"
                  paramvalue="element.repositoryId" type="hidden"/>
               <span class=smallb>Qty</span> 
               <dsp:input bean="PurchaselistFormHandler.quantity" size="2"
                  type="text" value="1"/> 
         </dsp:oparam>
      </dsp:droplet>

      <dsp:select bean="PurchaselistFormHandler.purchaseListId">
         <dsp:droplet name="ForEach">
            <dsp:param bean="Profile.purchaselists" name="array"/>
            <dsp:oparam name="output">
               <dsp:getvalueof id="elem" idtype="atg.repository.RepositoryItem"
                  param="element">
               <dsp:option value="<%=elem.getRepositoryId()%>"/>
               <dsp:valueof param="element.eventName">Unnamed Purchase List
               </dsp:valueof>
               </dsp:getvalueof>
            </dsp:oparam>
         </dsp:droplet>
      </dsp:select></td>
      </tr>
      <tr>
         <td><dsp:input bean="PurchaselistFormHandler.addItemToPurchaselist"
            type="submit" value="Add to list"/></td>
      </tr>

      <tr>
         <td>
            <table border=0 cellpadding=3 width=100%>
            <tr>
               <td><span class=smallb><dsp:a href=
                  "../user/purchase_lists.jsp?noCrumbs=false"><dsp:param
                  name="product" param="product.repositoryId"/><dsp:param
                  name="noCrumbs" value="false"/>Create new purchase
                  list</dsp:a></span></td>
            </tr>
            </table>
         </td>
      </tr>

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

We used `PurchaseListFormHandler` to add the given item to any of the purchase lists. We iterate through all the SKUs of the product providing an input option of quantity of item, and provide all the existing purchase lists of the customer in the dropdown box to choose one of them.

Create New Purchase List

In addition to being able to add an item to a purchase list, the user can also create a new purchase list. The **Create new purchase list** link directs the user to the Purchase Lists page (`user/purchase_lists.jsp`) in My Account, where the user can type in a new list name and click the **Create list** button. For more information on creating purchase lists, see the *Creating Purchase Lists* section in the My Account chapter of this guide.

Category Template Pages

Just as template pages are created to display products, template pages are also created to display categories. Motorprise has three different types of categories: root categories that contain other child categories, child categories that contain other child categories, and child categories that contain the products. We developed three generic template pages to display these types of categories.

`/en/common/CatalogNav.jsp`

`/en/catalog/category.jsp`

`/en/catalog/sub_category.jsp`

`CatalogNav.jsp` is used to display the root categories of the user. In Motorprise, we used assigned a catalog to each user. If a user doesn't have a catalog defined in his or her profile, we retrieve one from the contract of the `parentOrganization`. Each such catalog contains the three different types of categories mentioned above. We used `CatalogNav.jsp` throughout the catalog to provide the left navigation pane.

CatalogNav.jsp is used to display the categories on the left navigation pane.

`CatalogNav.jsp` displays all the root categories of the user by iterating through all of them as shown below.

```
<dsp:droplet name="/atg/dynamo/droplet/ForEach">
   <dsp:param  name="array" bean=
      "/atg/userprofiling/Profile.catalog.allRootCategories"/>
   <dsp:oparam name="output">
      <tr bgcolor="#FFFFFF">
         <td colspan=2><dsp:img src="../images/d.gif"/></td>
      </tr>
      <tr bgcolor="F7D774">
         <td><dsp:img src="../images/d.gif" hspace="5"/></td>
         <td><dsp:img src="../images/d.gif" vspace="1"/><br>

            <dsp:getvalueof id="urlStr" idtype="java.lang.String"
               param="element.template.url">
            <dsp:a page="<%=urlStr%>">
               <dsp:param name="id" param="element.repositoryId"/>
               <dsp:param name="navAction" value="pop"/>
               <dsp:param name="Item" param="element"/>
```

```
                <b><font size=-1 color="#555555"><dsp:valueof param=
                   "element.displayName"/></font></b>
             </dsp:a>
             </dsp:getvalueof>

      <br>
             <dsp:img src="../images/d.gif" vspace="1"/><br></td>
      </tr>
   </dsp:oparam>

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

The `template.url` property of the category contains the URL of the template to use for displaying the category.

`Category.jsp` is used to display the child categories that contain other child categories. For example, within the Exhaust category, there are Reducers, Gaskets, Headers, Mufflers, and Pipes subcategories.

Exhaust category, there are Reducers, Gaskets, Headers, Mufflers, and Pipes subcategories.

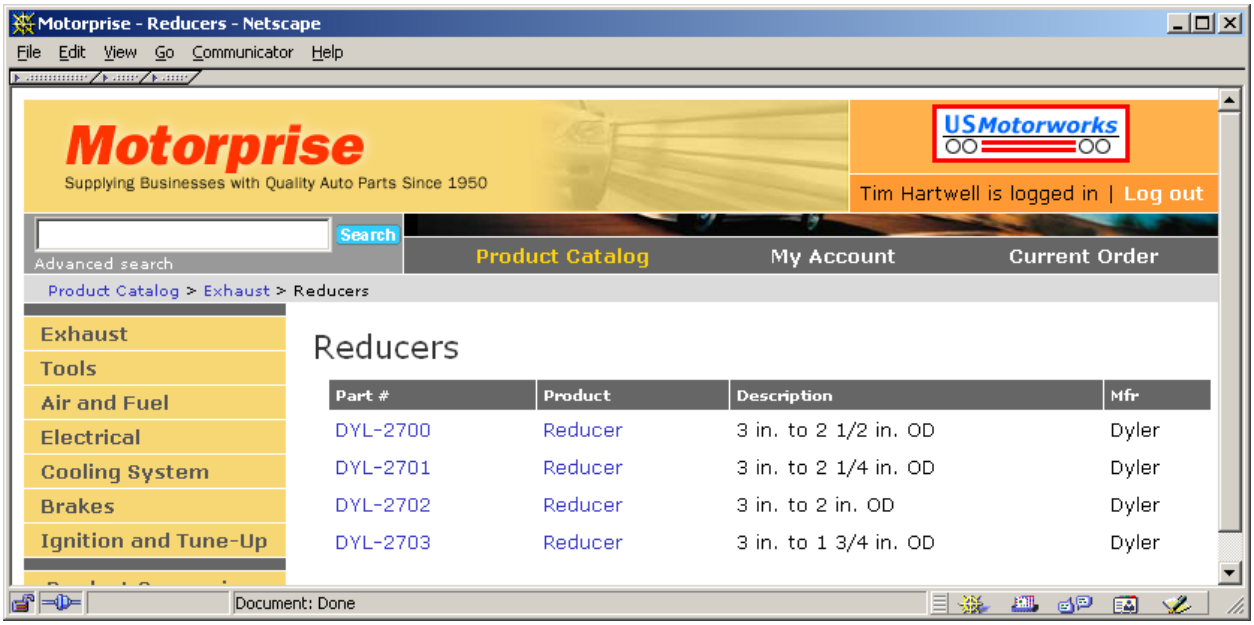
`Category.jsp` takes the `id` of the category as the page parameter and uses `CategoryLookup` to get the category and to display the child categories in table form as show below.

```
<dsp:droplet name="/atg/commerce/catalog/CategoryLookup">
   <dsp:oparam name="output">

   <dsp:getvalueof id="page_title" param="element.displayName">
   <dsp:include page="../common/HeadBody.jsp" flush="true">
      <dsp:param name="pagetitle" value="<%=page_title%>"/>
   </dsp:include>
   </dsp:getvalueof>

   <table border=0 cellpadding=0 cellspacing=0 width=800>
      <tr>
         <td colspan=2><dsp:include page="../common/BrandNav.jsp"
            flush="true"></dsp:include></td>
      </tr>

      <tr bgcolor="#DBDBDB">
         <td colspan=2 height=18><span class="small">  
            <dsp:droplet name="ArrayIncludesValue">
               <dsp:param name="array" bean="Profile.catalog.allrootcategories"/>
               <dsp:param name="value" param="element"/>
               <dsp:oparam name="false">
               <dsp:include page="../common/breadcrumbs.jsp" flush="true">
                  <dsp:param name="displaybreadcrumbs"
                     value="true"/></dsp:include>
          </dsp:oparam>
          <dsp:oparam name="true">
            <dsp:include page="../common/breadcrumbs.jsp" flush="true"><dsp:param
               name="displaybreadcrumbs" value="true"/><dsp:param name="navAction"
               value="jump"/><dsp:param name="navCount" value="0"/></dsp:include>
          </dsp:oparam>
        </dsp:droplet>
       </span>
      </td>
    </tr>


    <tr valign=top>
      <td width=175>
        <!-- category navigation -->
        <dsp:include page="../common/CatalogNav.jsp" flush="true"></dsp:include>
        <!-- incentives slot -->
        <dsp:include page="../common/Incentive.jsp" flush="true"></dsp:include>

      </td>
      <td width=625>
      <!-- promotion slot -->
      <table border=0 cellpadding=4 width=100%>
        <!--this row used to ensure proper spacing of table cell-->
        <tr><td colspan=2><dsp:img src="../images/d.gif" hspace="304"/></td></tr>
```

```
        <tr>
          <td colspan=2> <span class="categoryhead">
            <dsp:valueof param="element.itemDisplayName">No name
            </dsp:valueof></span></td>
        </tr>
        <tr>
      <td>

        <dsp:droplet name="/atg/dynamo/droplet/ForEach">
          <dsp:param name="array" param="element.childCategories"/>
          <dsp:oparam name="outputStart">
            <table border=0 cellpadding=3 width=100%>
          </dsp:oparam>
          <dsp:oparam name="output">
            <tr>
              <td width=3><dsp:img src="../images/d.gif" hspace="1"/></td>
              <td>
                <dsp:getvalueof id="urlStr" idtype="java.lang.String"
                   param="element.template.url">
            <dsp:a page="<%=urlStr%>">
              <dsp:param name="id" param="element.repositoryId"/>
              <dsp:valueof param="element.displayName">No name</dsp:valueof>
            </dsp:a>
                  </dsp:getvalueof>
                  <br> <dsp:valueof param="element.description"/>
              </td>
            </tr>
          </dsp:oparam>
          <dsp:oparam name="empty">
            <tr>
              <td width=3></td>
              <td>No child categories found.</td>
            </tr>
          </dsp:oparam>
        </dsp:droplet>
            </table>
            </td>
          </tr>
    </table>
    </td>
  </tr>
 </table>
  </dsp:oparam>
</dsp:droplet>
```

`CategoryLookup` takes the `id` of the category as input. However, we don't explicitly pass the `id` parameter of category to `CategoryLookup`. It is already defined as a page-level parameter, and outputs category object. We iterate through the `childCategories` of the category displaying the category and its attributes, also providing an URL to display that category.

sub_category.jsp is used to display the categories that contain products by taking the id of the category as the page parameter.



CategoryLookup is used to display the products contained in the category. The JSP fragment is shown below.

```
<dsp:droplet name="/atg/commerce/catalog/CategoryLookup">
   <dsp:oparam name="output">

   <dsp:getvalueof id="page_title" param="element.displayName">
   <dsp:include page="../common/HeadBody.jsp" flush="true">
      <dsp:param name="pagetitle" value="<%=page_title%>"/>
    </dsp:include>
    </dsp:getvalueof>

    <table border=0 cellpadding=0 cellspacing=0 width=800>
    <tr>
      <td colspan=2><dsp:include page="../common/BrandNav.jsp" flush=
        "true"></dsp:include></td>
    </tr>

    <tr bgcolor="#DBDBDB">
      <td colspan=2 height=18><span class="small">  
        <dsp:droplet name="ArrayIncludesValue">
          <dsp:param name="array" bean="Profile.catalog.rootcategories"/>
          <dsp:param name="value" param="element"/>
          <dsp:oparam name="false">
```

```
        <dsp:include page="../common/breadcrumbs.jsp" flush="true"><dsp:param
          name="displaybreadcrumbs" value="true"/></dsp:include>
      </dsp:oparam>
      <dsp:oparam name="true">
        <dsp:include page="../common/breadcrumbs.jsp" flush="true"><dsp:param
          name="displaybreadcrumbs" value="true"/><dsp:param name="navAction"
          value="jump"/><dsp:param name="navCount" value="0"/></dsp:include>
      </dsp:oparam>
    </dsp:droplet>
     </span>
   </td>
 </tr>

<tr valign=top>
  <td width=175>
  <!-- left panel -->
  <dsp:include page="../common/CatalogNav.jsp" flush="true"></dsp:include>
  <!-- incentives slot -->
  <dsp:include page="../common/Incentive.jsp" flush="true"></dsp:include>
  </td>
  <td width=625><!-- main content -->

  <table border=0 cellpadding=4 width=100%>
    <!--this row used to ensure proper spacing of table cell-->
    <tr><td colspan=2><dsp:img src="../images/d.gif" hspace="304"/></td></tr>
<tr>
      <td colspan=2>
       <span class="categoryhead">
      <dsp:valueof param="element.itemDisplayName">No name
        </dsp:valueof></span>
      <br><dsp:img src="../images/d.gif" vspace="4"/><br>
        <table border=0 cellpadding=4 cellspacing=1 width=100%>
          <dsp:droplet name="/atg/dynamo/droplet/ForEach">
            <dsp:param name="array" param="element.childProducts"/>
            <dsp:oparam name="outputStart">
              <tr valign="bottom">
                <td><dsp:img src="../images/d.gif" hspace="1"/></td>
                <td bgcolor="#666666" colspan=2><span class="smallbw">Part #
                </span></td>
                <td bgcolor="#666666" colspan=2><span class="smallbw">
                  Product</span></td>
                <td bgcolor="#666666" colspan=2><span class=
                 "smallbw">Description</span></td>
                <td bgcolor="#666666" colspan=2><span class=
                  "smallbw">Mfr</span></td>
              </tr>
            </dsp:oparam>
            <dsp:oparam name="output">
              <tr valign="top">
        <td><dsp:img src="../images/d.gif" hspace="1"/></td>
```

```
      <dsp:droplet name="/atg/dynamo/droplet/ForEach">
        <dsp:param name="array" param="element.childSKUs"/>
        <dsp:param name="elementName" value="sku"/>
        <dsp:oparam name="output">
         <td><nobr>
         <dsp:getvalueof id="skuURL" idtype="java.lang.String"
           param="element.template.url">
   <dsp:a page="<%=skuURL%>">
     <dsp:param name="id" param="element.repositoryId"/>
           <dsp:valueof param="sku.manufacturer_part_number">No part
             number</dsp:valueof>
   </dsp:a>
         </nobr></td>
   </dsp:getvalueof>
        </dsp:oparam>
      </dsp:droplet>

      <td> </td>
      <td>
<dsp:getvalueof id="urlStr" idtype="java.lang.String"
  param="element.template.url">
<dsp:a page="<%=urlStr%>">
  <dsp:param name="id" param="element.repositoryId"/>
  <dsp:valueof param="element.displayName">No name</dsp:valueof>
</dsp:a>
</dsp:getvalueof>
      </td>

      <td> </td>
      <td><dsp:valueof param="element.description">No description
      </dsp:valueof></td>

      <td> </td>
      <td><dsp:valueof param=
        "element.manufacturer.displayName">Unknown
      </dsp:valueof></td>



    </tr>
  </dsp:oparam>
  <dsp:oparam name="outputEnd">
    <tr>
      <td> </td>
    </tr>
  </dsp:oparam>
  <dsp:oparam name="empty">
    <tr><td>  No child products.</td></tr>
  </dsp:oparam>
</dsp:droplet>
```

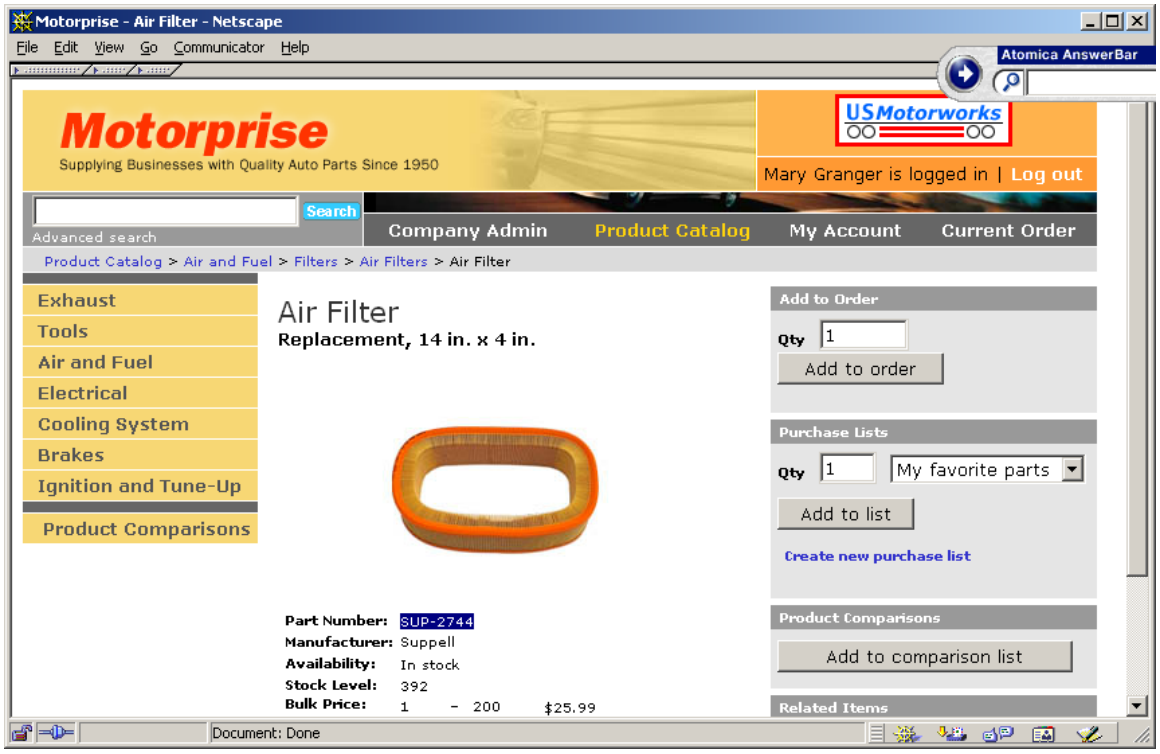
```
            </table>
          </td>
        </tr>
      </table>
      </td>
    </tr>
  </table>
  </dsp:oparam>
</dsp:droplet>
```

In the above code sample, we used `CategoryLookup` to get the category corresponding to the `id`, and we iterated through the products and SKUs of each product displaying the SKU and its attributes.

Historical Navigation

At the top of each template page, we display historical navigation information by listing the hierarchy of categories that a customer has already visited on his or her way to the current page.

For example, if a customer navigates from the top of the Motorprise catalog to the Air and Fuel category, to the Filters and then Air Filters subcategory pages, and then to the product page for the SUP-2744 air filter, the navigation history (at the top of the page) looks like this:



Each of the items in the hierarchical navigation is a link. For example, if the user clicks on Air and Fuel, she is brought to the category template page for Air and Fuel.

If a customer goes directly from the home page to the product page of the SUP-2744 air filter, (for example, from the featured product on the home page) the historical navigation still reflects the catalog's hierarchical structure. We wanted the navigation path to be intuitive to the customer and therefore, the historical navigation list reflects the category pages so that the user can have a sense of how this item fits into the catalog. To do this, we invoke the `CatalogNavHistoryCollector` component with the `navAction` parameter set to "jump".

There are two parts to using historical navigation: collecting the information and displaying it.

Collecting the Navigation History

We used the component `/atg/commerce/catalog/CatalogNavHistoryCollector` to collect the locations visited and to add them to the array of visited locations. In Motorprise, the visited locations are the repository items that represent the products and categories of the product catalog. This snippet, taken from `/Motorprise/en/common/breadcrumbs.jsp`, invokes the `CatalogNavHistoryCollector`:

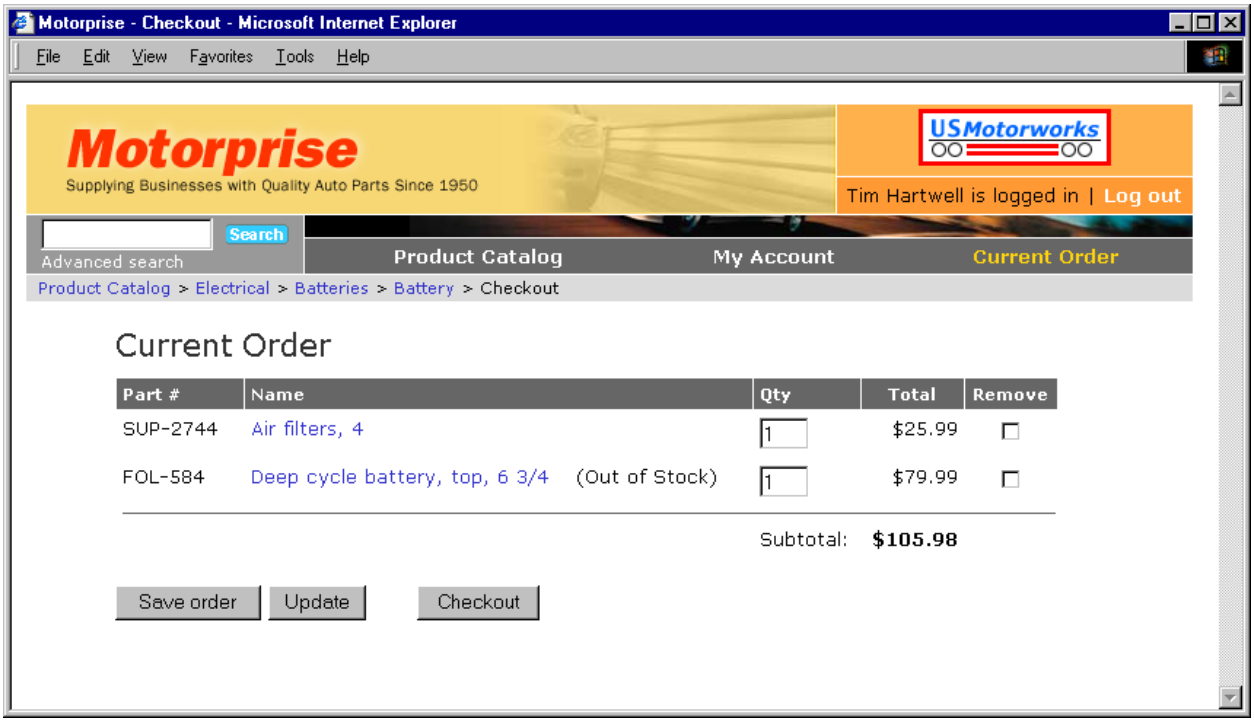
```
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param name="value" param="no_new_crumb"/>
  <dsp:oparam name="true">
  </dsp:oparam>
```

```
  <dsp:oparam name="default">
     <dsp:droplet name="CatalogNavHistoryCollector">
        <dsp:param name="navAction" param="navAction"/>
        <dsp:param name="item" param="element"/>
     </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>
```

There are two things to note in this code snippet. Although both these techniques are specific to the Motorprise implementation, they can be applied to any site. First, notice that the required parameter `navCount` is not passed to `CatalogNavHistoryCollector`. The `navCount` parameter is set in the page that invokes this snippet. Since JSPs invoked as servlets from other JSPs are in a sub-scope of their callers, they have access to the same parameters as the caller.

Second, we used the `no_new_crumb` parameter to decide whether to invoke the snippet. This is just a switch on a parameter passed to the page to determine whether to add the current location to the `NavHistory` or not. However, it demonstrates how we decided to address navigation for pages that do not represent catalog items. For example, the search page, the shopping cart, and the user profile page are not added to the `NavHistory` like regular catalog pages.

Another use of this parameter includes certain instances in the product catalog where we want users to temporarily leave the catalog, but then return quickly. In such situations we use special implementations of breadcrumbs using the `no_new_crumb` parameter. For example, when a user adds an item to their current order, the "Checkout" breadcrumb is added to the list allowing a user to quickly return to any part of the catalog. When the user returns to catalog, the "Checkout" value is not part of the breadcrumbs trail.



"Checkout" is added to the catalog breadcrumbs when a user adds an item to the order.

Displaying the Navigation History

The property `/atg/commerce/catalog/CatalogNavHistory.navHistory` is a `LinkedList` of locations. The `CatalogNavHistoryCollector` populates this list as described in the preceding section. The following snippet from `breadcrumbs.jsp` demonstrates how the navigation history is displayed in Motorprise. A `ForEach` component iterates over the `NavHistory` list and a link is created for each item in the list. Comments in the JSP describe the variations from that behavior.

```
<%
/* ------------------------------------------------
 * use the ForEach droplet to render the navHistory array.
 * ------------------------------------------------ */
%>
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <dsp:param name="value" param="displaybreadcrumbs"/>
  <dsp:oparam name="true">

    <dsp:droplet name="/atg/dynamo/droplet/ForEach">
      <dsp:param bean="CatalogNavHistory.navHistory" name="array"/>
      <dsp:param name="elementName" value="crumb"/>
      <dsp:oparam name="output">
```

```
<dsp:droplet name="/atg/dynamo/droplet/Switch">
  <%
  /* ------------------------------------------------
   * We want to put a separator between the items in the navHistory. In
   * this example we put | sign between them.  We use a switch droplet to
   * identify the first item in the array because we don't want to render
   * a separator, but a link to Store Home before the first item.
   * ------------------------------------------------ */
  %>
  <dsp:param name="value" param="count"/>

  <dsp:oparam name="1">
    &amp;nbsp; <dsp:a href="../home.jsp">Product Catalog</dsp:a> &amp;gt;
  </dsp:oparam>

  <dsp:oparam name="default">
    &amp;gt;
  </dsp:oparam>

</dsp:droplet>

<dsp:droplet name="/atg/dynamo/droplet/IsNull">
  <dsp:param name="value" param="crumb"/>

  <dsp:oparam name="true">
     element is null
  </dsp:oparam>

  <dsp:oparam name="false">
    <dsp:droplet name="/atg/dynamo/droplet/Switch">
      <%
      /* ------------------------------------------------
       * Use a switch droplet to compare size to count. When
       * they are the same, then we are on the last item in
       * array iterated by the ForEach.
       * ------------------------------------------------ */
      %>
      <dsp:param name="value" param="size"/>

      <dsp:getvalueof id="countParam" idtype="Integer" param="count">
      <dsp:oparam name="<%=countParam.toString()%>">
        <dsp:droplet name="/atg/dynamo/droplet/Switch">
          <%
          /* ------------------------------------------------
           * The last item in the list is generally the item we are
           * currently visiting and should therefore not be a link.
           * In some cases, when we do not want to add a new breadcrumb,
           * we want the last item to be a link.  We do this on the
           * shopping cart page, search page, and others.  This is
           * indicated by the "no_new_crumb" parameter.
```

```
                 * ------------------------------------------------ */
                %>
                <dsp:param name="value" param="no_new_crumb"/>

                <dsp:oparam name="true">
              <dsp:getvalueof id="urlStr" idtype="java.lang.String" param=
                "crumb.template.url">
                    <dsp:a page="<%=urlStr%>">
                      <dsp:param name="id" param="crumb.repositoryId"/>
                      <dsp:param name="navAction" value="pop"/>
                      <dsp:param name="Item" param="crumb"/>
                      <dsp:valueof param="crumb.displayName">No name</dsp:valueof>
              </dsp:a>
              </dsp:getvalueof>
                  </dsp:oparam>

                  <dsp:oparam name="default">
                    <dsp:valueof param="crumb.displayName"/>
                  </dsp:oparam>
                </dsp:droplet>
              </dsp:oparam>
              </dsp:getvalueof>

              <dsp:oparam name="default">
             <dsp:getvalueof id="urlStr" idtype="java.lang.String" param=
               "crumb.template.url">
                <dsp:a page="<%=urlStr%>">
            <dsp:param name="id" param="crumb.repositoryId"/>
            <dsp:param name="navAction" value="pop"/>
            <dsp:param name="Item" param="crumb"/>
            <dsp:valueof param="crumb.displayName">No name</dsp:valueof>
               </dsp:a>
               </dsp:getvalueof>
              </dsp:oparam>
            </dsp:droplet>

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

      </dsp:oparam>
    </dsp:droplet> <%/* end ForEach */%>
  </dsp:oparam>
</dsp:droplet>
```

Displaying different sets of breadcrumbs

There are certain instances where a user can arrive at a particular page from different navigational contexts. In the code snippet below, taken from `cart.jsp`, you will notice that the breadcrumbs droplet is enclosed within a switch droplet that examines a parameter called `noCrumbs`. This switch allows us to display two different sets of breadcrumbs depending on the context of the situation. When a user selects

"Add to Order" from the product page, the parameter `noCrumbs` is passed to the current order page. If the value of `noCrumbs` is false, then the page is arrived at from the product page and the navigational history from the product catalog is displayed. Otherwise a new breadcrumb is displayed, indicating that "Current Order" was selected.

```
<dsp:droplet name="Switch">
   <dsp:param name="value" param="noCrumbs"/>
   <dsp:oparam name="false">
      <dsp:include page="../common/breadcrumbs.jsp" flush="true"><dsp:param
         name="displaybreadcrumbs" value="true"/><dsp:param name="no_new_crumb"
         value="true"/></dsp:include> &gt; Checkout
   </dsp:oparam>
   <dsp:oparam name="default">
      <dsp:param name="noCrumbs" value="true"/>
        Current Order
   </dsp:oparam>
</dsp:droplet>
```

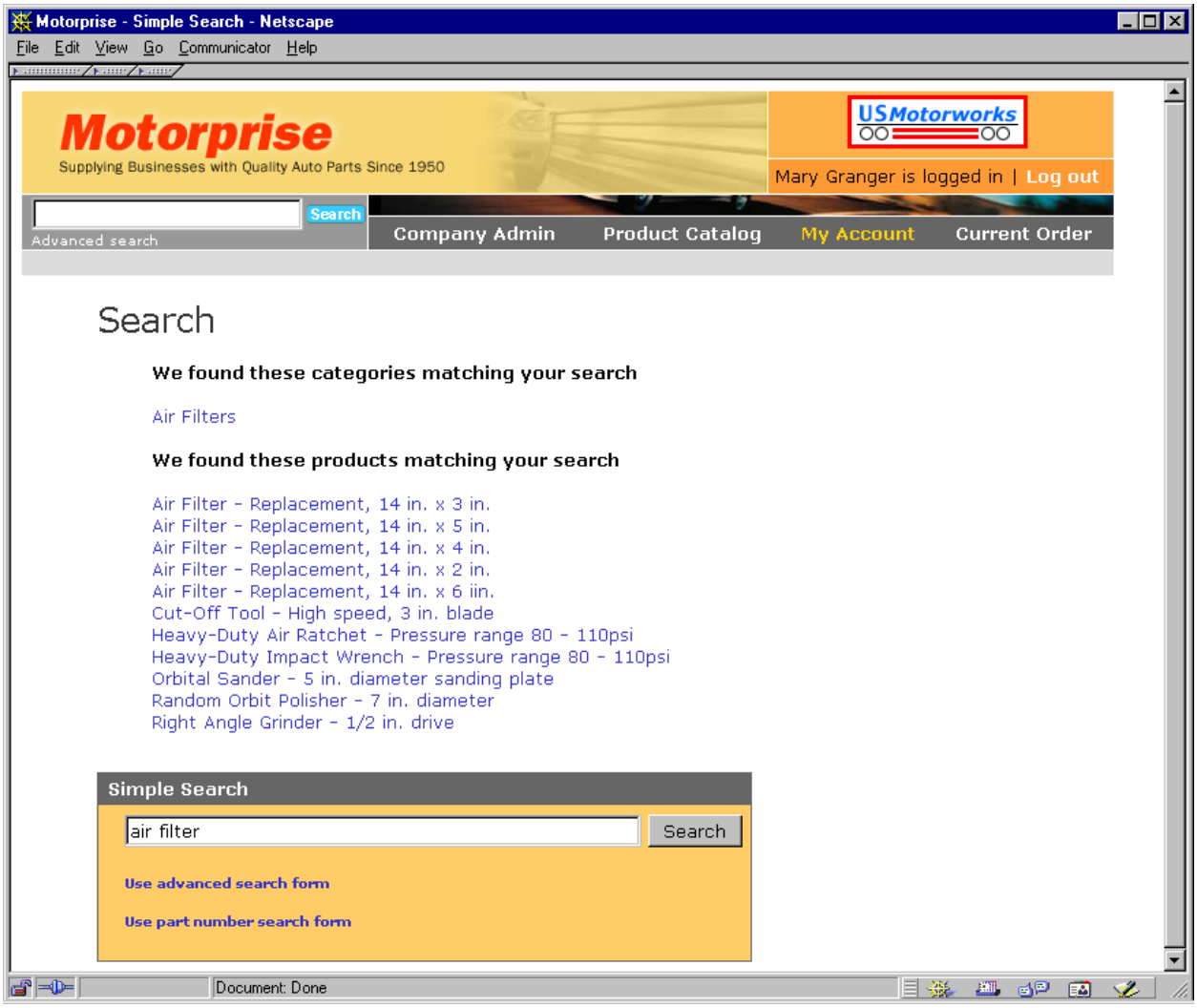
We created a similar implementation for scheduled orders where certain pages are viewed in a number of different contexts.

Searching

Searching allows customers to find products that satisfy a certain set of criteria. In the Motorprise store, we implemented two types of searching: simple and advanced. Simple searches allow a customer to search the catalog for keywords and simple text search against the `displayName` and `description` properties of product and catalog items. In other words, your search will return both products and categories. In an advanced search, a customer can also search using additional product attributes such as manufacturer, category, and part number. The advanced search only returns products. Neither type of searching is case sensitive.

Simple Search

In a simple search, customers search the catalog by entering text. For example, a customer could search for "air filter." The customer input is searched as a complete string (for example, the above query would be based on "air filter" and not "air" or "filter"). We used the `SearchFormHandler` to create the query to the SQL Repository (Product Catalog) and to return a result set. It returns an empty set if no products are found. The result set is a collection of repository items that can be displayed in any format. Simple searches return both products and categories.



Searching for "air filter" in Motorprise.

We used properties or configuration files to define the action a page performs and Boolean arguments to specify the type of searching to perform. Property names specify which catalog properties to search. The single configurable form handler simplifies catalog searching and should support all searching requirements. If your store requires custom searching beyond configuration changes, you can easily extend this form handler or write another one.

We implemented the search feature in Motorprise with a combination of JSPs, JSP fragments, and a Nucleus component configuration file (properties file). The properties file creates and configures a Nucleus component of class `atg.commerce.catalog.SearchFormHandler` to perform the search. The simple search functionality allows the user to enter a string and get products or categories that match the keywords as results.

Products have multiple properties such as `keywords`, `description`, `longDescription`, and `displayName`. The `keywords` property is a multi-valued property and the others are single-valued properties. Thus, for every product, there can be multiple keywords associated with it. A keyword search looks through all keyword properties for a string input. A text search looks through `description` and `displayName` for a string input, but does not search keyword properties. By default, a keyword and text search is done on each query. This means that both keywords as well as description and display name will be queried.

SimpleSearchFragment.jsp

We wanted customers to be able to search from other store pages besides the search page, `simple_search.jsp`. Therefore, we created this small search form fragment, `/en/search/SimpleSearchFragment.jsp`, that defines the search text entry field and the Search button to embed in any page that requires searching. We embedded this fragment in the Motorprise header so that users can search from every page.

The search fragment is embedded in the Motorprise header so users can search from any page.

This fragment imports the `CatalogSearch` component in order to configure its `searchInput` property and invoke its search handler.

Finally, the form redirects to the page name passed via the `FormAction` parameter. This page fragment declares a page parameter that is used as the action for the search form. In other words, the `FormAction` parameter is the file name of the page to which the user is redirected when the search form's `submit` method is invoked.

The following example shows `SimpleSearchFragment.jsp`:

```
<%/* This page fragment is intended to be embedded on any page that
     requires a search capability.  */%>
```

```
<dsp:importbean bean="/atg/commerce/catalog/CatalogSearch"/>

<dsp:form action="simple_search.jsp" method="POST">
<table cellspacing="0" cellpadding="0" border="0" bgcolor="#999999" width=100%>

  <tr><td colspan=3><dsp:img src="../images/d.gif" vspace="0"/></td></td></tr>
  <tr>
    <td><dsp:img src="../images/d.gif" hspace="3"/></td>
    <td width=100>
    <!-- form elements -->
    <input name="repositoryKey" type="hidden" value="<dsp:valueof
      bean='/OriginatingRequest.requestLocale.locale'/>">
    <dsp:input bean="CatalogSearch.searchInput" size="15" type="text" value=""/>
    </td>
     <!--  use this hidden form tag to make sure the search handler is invoked if
       someone does not hit the submit button -->
     <td align=center>
    <dsp:input bean="CatalogSearch.search" type="hidden" value="Search"/>
    <dsp:input type="image" src="../images/motorprise-search.gif"
      bean="CatalogSearch.search" name="search" value="Search" border="0"/> </td>
    <!-- end form elements -->

  </tr>
  <tr>
    <td><dsp:img src="../images/d.gif" hspace="3"/></td>
    <td colspan=2><dsp:a href="advanced_search.jsp"><font color="#FFFFFF"><span
      class=smallw>Advanced search</span></font></dsp:a></td>
  </tr>
  <tr><td colspan=3><dsp:img src="../images/d.gif" vspace="0"/></td></tr>


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

The hidden input tag is an HTML workaround for forms with a single input field. The hidden input allows the user to invoke the `submit` method by typing the return key within the text input field.

Without this hidden tag, the user must click the **Search** button.

simple_search.jsp

The `simple_search.jsp` is a page built of several JSP code fragments that provide their own specific functionality. `simple_search.jsp` uses `SearchResults.jsp` to display the results of the search.

First, we display the search results at the top of the page by including `SearchResults.jsp`. The `CatalogSearch` component is used to perform the actual searching. When the user submits the search, the handler search method is invoked. This basically fills up the search results by `ItemType` property of `catalogSearch`.

The following example shows `simple_search.jsp`:

```
<%@ taglib uri="dsp" prefix="dsp" %>
<dsp:page>

<dsp:importbean bean="/atg/dynamo/droplet/ForEach"/>
<dsp:importbean bean="/atg/commerce/catalog/CatalogSearch"/>
<dsp:importbean bean="/atg/dynamo/droplet/Switch"/>

<DECLAREPARAM NAME="noCrumbs"
   CLASS="java.lang.String"
   DESCRIPTION="This is for deciding what kind of breadcrumbs to display.
   If this is true, then breadcrumbs will show: Store Home|Search,
   instead of nav history. Default is false."
   OPTIONAL>


<dsp:include page="../common/HeadBody.jsp" flush="true"><dsp:param
  name="pagetitle" value=" Simple Search"/></dsp:include>
<table border=0 cellpadding=0 cellspacing=0 width=800>
<tr>
  <td colspan=2><dsp:include page="../common/BrandNav.jsp"
    flush="true"></dsp:include></td>
</tr>
<tr bgcolor="#DBDBDB" >
  <td colspan=2 height=18> 
  <!-- breadcrumbs -->
  <span class="small">
  <dsp:droplet name="Switch">
    <dsp:param name="value" param="noCrumbs"/>
      <dsp:oparam name="true"><dsp:a href="../home.jsp">Product Catalog</dsp:a>
          Simple Search</dsp:oparam>
      <dsp:oparam name="default"><dsp:param name="noCrumbs" value="false"/>
        <dsp:include page="../common/breadcrumbs.jsp" flush="true"><dsp:param
          name="displaybreadcrumbs" value="true"/><dsp:param name="no_new_crumb"
            value="true"/></dsp:include>
      </dsp:oparam>
  </dsp:droplet></span>
  </td>
</tr>


<tr>
  <td width=55><dsp:img src="../images/d.gif" hspace="27"/></td>
  <td valign="top" width=745><br><span class="big">Search</span></td>
</tr>

<tr>
  <td width="40"><dsp:img src="../images/d.gif"/></td>
  <td><br>
    <dsp:getvalueof id="pval0" bean="CatalogSearch.searchResultsByItemType">
```

```
      <dsp:include page="SearchResults.jsp" flush="true"><dsp:param
      name="ResultArray" value="<%=pval0%>"/></dsp:include></dsp:getvalueof>
  </td>
</tr>

<tr><td colspan=2><dsp:img src="../images/d.gif" vspace="6"/></td></tr>

<tr>
  <td width="40"><dsp:img src="../images/d.gif"/></td>
  <td><!-- simple search box -->
    <table bgcolor="#FFCC66" border=0 cellpadding=0 cellspacing=0>
      <tr>
        <td colspan=3>
          <table width=100% cellpadding=4 cellspacing=0 border=0>
          <tr><td class=box-top> Simple Search</td></tr>
          </table>
        </td>
      </tr>

      <tr><td bgcolor="#666666"><dsp:img src="../images/d.gif" width="1"/></td>
          <td>
            <dsp:form action="simple_search.jsp" method="POST">
            <table width=100% cellpadding=6 cellspacing=0 border=0>
            <tr>
              <td></td>
              <td bgcolor="#ffcc66">
              <input name="repositoryKey" type="hidden" value="<dsp:valueof
                bean="/OriginatingRequest.requestLocale.locale"/>">
              <dsp:input bean="CatalogSearch.searchInput" size="30" type="text"/>
              <input name="noCrumbs" type="hidden" value="<dsp:valueof
                param="noCrumbs"/>">
              <!--  use this hidden form tag to make sure the search handler is
                invoked if someone does not hit the submit button -->
              <dsp:input bean="CatalogSearch.search" type="hidden"
                value="Search"/>
              <dsp:input bean="CatalogSearch.search" type="submit"
                value="Search"/><br><!--<span class="help">Separate words or
                  phrases by <b>AND</b> or <b>OR</b></span>-->
              <p>
              <span class=smallb><dsp:a href="advanced_search.jsp">
                <dsp:param name="noCrumbs" param="noCrumbs"/>
                Use advanced search form</span></dsp:a>
              </td>
            </tr>
            <tr>
              <td></td>
              <td>
              <span class=smallb><dsp:a href="part_number_search.jsp">
               <dsp:param name="noCrumbs" param="noCrumbs"/>
                 Use part number search form</dsp:a></span>
```

```
            </td>
          </tr>
          </table>
          </dsp:form>
        </td>
        <td bgcolor="#666666"><dsp:img src="../images/d.gif" width="1"/></td>
    </tr>
    <tr><td bgcolor="#666666" colspan=3><dsp:img src=
      "../images/d.gif"/></td></tr>
    </table>
  </td>
</tr>
</table>


</body>
</html>
</dsp:page>
```

SearchResults.jsp

We used the `SearchResults` page fragment to iterate and display search results from `simple_search.jsp`. The `ForEach` component is sent the parameter array, which can take many kinds of multi-valued objects.

The type in this case is a `java.util.Map` that is the product of the `SearchFormHandler` component.

The map has one entry for each value specified in the `itemTypes` value of the component's configuration file. In `SimpleSearch`, the `itemTypes` are `category` and `product`.

The map is a collection of search result arrays and their associated `itemType` identifiers. The value associated with category and product in the map is an array of category or product items, or null if no results are found for that key.

The category and product map elements are extracted by iterating through the map with the standard `ForEach` component in the outermost `ForEach` droplet tag. For each element of the map, the `Switch` component is then invoked to distinguish the category and product elements, and to affect the order in which they're rendered on the page.

Once the category and product search result array is obtained, it is necessary to iterate through each element of that array to display the results. In order to detect an empty result array, we used another `Switch` component, this time for the unset `dsp:oparam` feature of `Switch` that identifies an empty input array and a message is rendered indicating that no items of the given type were found. When the array is non-null, the default `dsp:oparam` tag of `Switch` is rendered. In this case, another call to the `ForEach` component iterates through the array and displays each of its items.

```
<%@ taglib uri="dsp" prefix="dsp" %>
<dsp:page>
```

```
<%
/* ----------------------------------------------------------------
This JSP droplet displays the contents of 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.
----------------------------------------------------------------*/
%>


<DECLAREPARAM NAME="ResultArray"
              CLASS="java.util.Map"
              DESCRIPTION="Array of Search Results">


<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 name="array" param="ResultArray"/>

  <%/*Each item in this array is a Collection of Categories or Products...*/%>
  <dsp:param name="elementName" value="ResultCollection"/>

  <dsp:oparam name="output">
    <dsp:droplet name="Switch">

      <%--The key tells us if this is a Collection of Products or Categories: --%>
      <dsp:param name="value" param="key"/>

      <%--For the list of CATEGORIES: --%>
      <dsp:oparam name="category">

        <blockquote>

        <dsp:droplet name="Switch">
          <dsp:param name="value" param="ResultCollection"/>
          <dsp:oparam name="default">
            <p>

            <%-- For each Category in the Collection: --%>
            <dsp:droplet name="ForEach">
              <dsp:param name="array" param="ResultCollection"/>
```

```
          <dsp:param name="sortProperties" value="+displayName"/>
          <dsp:param name="elementName" value="Category"/>
          <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="urlStr" idtype="java.lang.String"
        param="Category.template.url">
            <dsp:a page="<%=urlStr%>">
              <dsp:param name="id" param="Category.repositoryId"/>
              <dsp:param name="navAction" value="jump"/>
              <dsp:param name="Item" param="Category"/>
              <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 name="value" param="ResultCollection"/>

      <dsp:oparam name="default">

        <%/*For each Product in the Collection: */%>
        <dsp:droplet name="ForEach">
          <dsp:param name="array" param="ResultCollection"/>
          <dsp:param name="sortProperties" value="+displayName"/>
          <dsp:param name="elementName" value="Product"/>
```

```
            <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="urlStr" idtype="java.lang.String" param=
            "Product.template.url">
              <dsp:a page="<%=urlStr%>">
                <dsp:param name="id" param="Product.repositoryId"/>
                <dsp:param name="navAction" value="jump"/>
                <dsp:param name="Item" param="Product"/>
                <dsp:valueof param="Product.displayName">No name
                </dsp:valueof> - <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:page>
```

Advanced Search

In addition to the keyword and text search of the simple search functionality, the advanced search functionality allows the customer to search for products by part number and manufacturer, which are

defined as product properties. Advanced search is divided into two types of search, part number search and searching by using manufacturer and category.

Part Number Searching

Since part number is not a property of `Product` items but instead a property of `sku` items, we use a different component, `atg/projects/b2bstore/catalog/PartNumberSearchFormHandler`, to search for part numbers. This component uses a custom form handler, `CatalogSearchFormHandler`, and uses different properties than `AdvProductSearch`.

This is the properties file for `catalogSearchFormHandler`:

```
# Allow users to search by part number
# Part number exists on a sku
$class=atg.commerce.catalog.custom.CatalogSearchFormHandler
$scope=request
catalogTools=/atg/commerce/catalog/CatalogTools
doTextSearch=true
textSearchPropertyNames=manufacturer_part_number
itemTypes=sku
repositories=/atg/commerce/catalog/ProductCatalog
```

The following is the code for Part Number searching in `part_number_search.jsp`.

```
<dsp:form action="part_number_search.jsp" method="POST">
<table width="100%" border="0" cellspacing="0" cellpadding="4" bgcolor="#FFCC66">
   <tr><td colspan=2><dsp:img src= "../images/d.gif" height="10"/></td></tr>
   <tr valign="top">
      <td width="15%" align="right">
      <span class="smallb">Part#</span>
      </td>
      <td>
         <dsp:input bean="PartNumberSearchFormHandler.searchInput" size="25"
            type="text"/> 
            <br> <span class="small">Use <b>*</b> for partial part number
               search.</span>
      </td>
   </tr>
   <tr>
      <td> </td>
      <td>
      <input name="repositoryKey" type="hidden" value="<dsp:valueof
         bean="/OriginatingRequest.requestLocale.locale"/>">
      <input name="noCrumbs" type="hidden" value="<dsp:valueof
         param="noCrumbs"/>"><br>
      <dsp:input bean="PartNumberSearchFormHandler.search" type="hidden"
        value="Search"/>
      <dsp:input bean="PartNumberSearchFormHandler.search" type="submit"
```

```
        value="Search"/>
</dsp:form></td>
```

The SearchInput property of PartNumberSearchFormHandler component is set using the part number input from user. Based on this manufacturer_part_number property of SKU is searched.

part_number_search.jsp invokes the PartNumberSearchResults.jsp fragment passing the result of the search that contains the array of SKUs of the products found. The following is the code fragment in part_number_search.jsp to include this:

```
<dsp:getvalueof id="pval0" idtype="java.util.List"
   bean="PartNumberSearchFormHandler.searchResults">

   <dsp:include page="PartNumberSearchResults.jsp" flush="true">
   <dsp:param name="Skus" value="<%=pval0%>"/>
   </dsp:include>

   </dsp:getvalueof>
```

Once we get SKUs with the given part number, we fetch products corresponding to those SKUs in PartNumberSearchResults.jsp. We iterate through each SKU and get all the products corresponding to SKU using RQLQueryForEach and display them.

The following is a snippet from PartNumberSearchResults.jsp that display the products found:

```
<dsp:droplet name="ForEach">
  <dsp:param name="array" param="Skus"/>
  <dsp:param name="elementName" value="element"/>
  <dsp:oparam name="outputStart">
   <b>We found these products matching your search</b>
   <br><br>
  </dsp:oparam>
  <dsp:oparam name="output">

  <dsp:droplet name="RQLQueryForEach">
    <dsp:param name="skuId" param="element"/>
    <dsp:param name="repository" bean="/atg/commerce/catalog/ProductCatalog" />
    <dsp:param name="itemDescriptor" value="product"/>
    <dsp:param name="queryRQL" value="childSKUs INCLUDES :skuId  "/>
    <dsp:param bean="/atg/dynamo/transaction/TransactionManager"
      name="transactionManager"/>
    <dsp:oparam name="output">
      <%-- Display a link to the element: --%>
      <dsp:getvalueof id="urlStr" idtype="java.lang.String"
        param="element.template.url">
      <dsp:a page="<%=urlStr%>">
        <dsp:param name="id" param="element.repositoryId"/>
```

```
      <dsp:param name="navAction" value="jump"/>
      <dsp:param name="Item" param="element"/>
      <dsp:valueof param="element.displayName">No name</dsp:valueof> -
         <dsp:valueof param="element.description"/>
       </dsp:a>
       </dsp:getvalueof>
       <br>
      </dsp:oparam>
  </dsp:droplet>
  </dsp:oparam>
  <dsp:oparam name="empty">
    <b>There are no products matching your search</b>
    <p>
  </dsp:oparam>
</dsp:droplet>
```

Manufacturer and Text Input Search

The `AdvProductSearch` component is used for this search.

The advanced search functionality uses the `SearchFormHandler` but requires a slightly more complex properties file than the simple search Nucleus component. Advanced searches return only products because the additional search criteria are based on products only.

`/atg/commerce/catalog/AdvProductSearch.properties`

```
$class=atg.commerce.catalog.SearchFormHandler
$scope=session
doKeywordSearch=true
keywordsPropertyNames=keywords
doTextSearch=true
textSearchPropertyNames=description,displayName
doAdvancedSearch=true
advancedSearchPropertyNames=childSKUs
doHierarchicalSearch=true
ancestorCategoriesPropertyName=ancestorCategories
minScore=1
catalogTools=CatalogTools
itemTypes^=CatalogTools.productItemTypes
maxResultsPerPage=20
enableCountQuery=true
```

At the Motorprise configuration layer, we added a manufacturer property to this component:

```
# /atg/commerce/catalog/AdvProductSearch
advancedSearchPropertyNames+=manufacturer
```

As with the simple search, the SearchFormHandler is session scoped. Keyword and text searching are configured identically for both simple and advanced searches and they both use the same catalog. For the enumerated types used in the search (manufacturer), the possible values are inserted into <select> input tags on the form. These values are not coded into the form, but instead are retrieved from the catalog via the propertyValuesByType property of the SearchFormHandler.

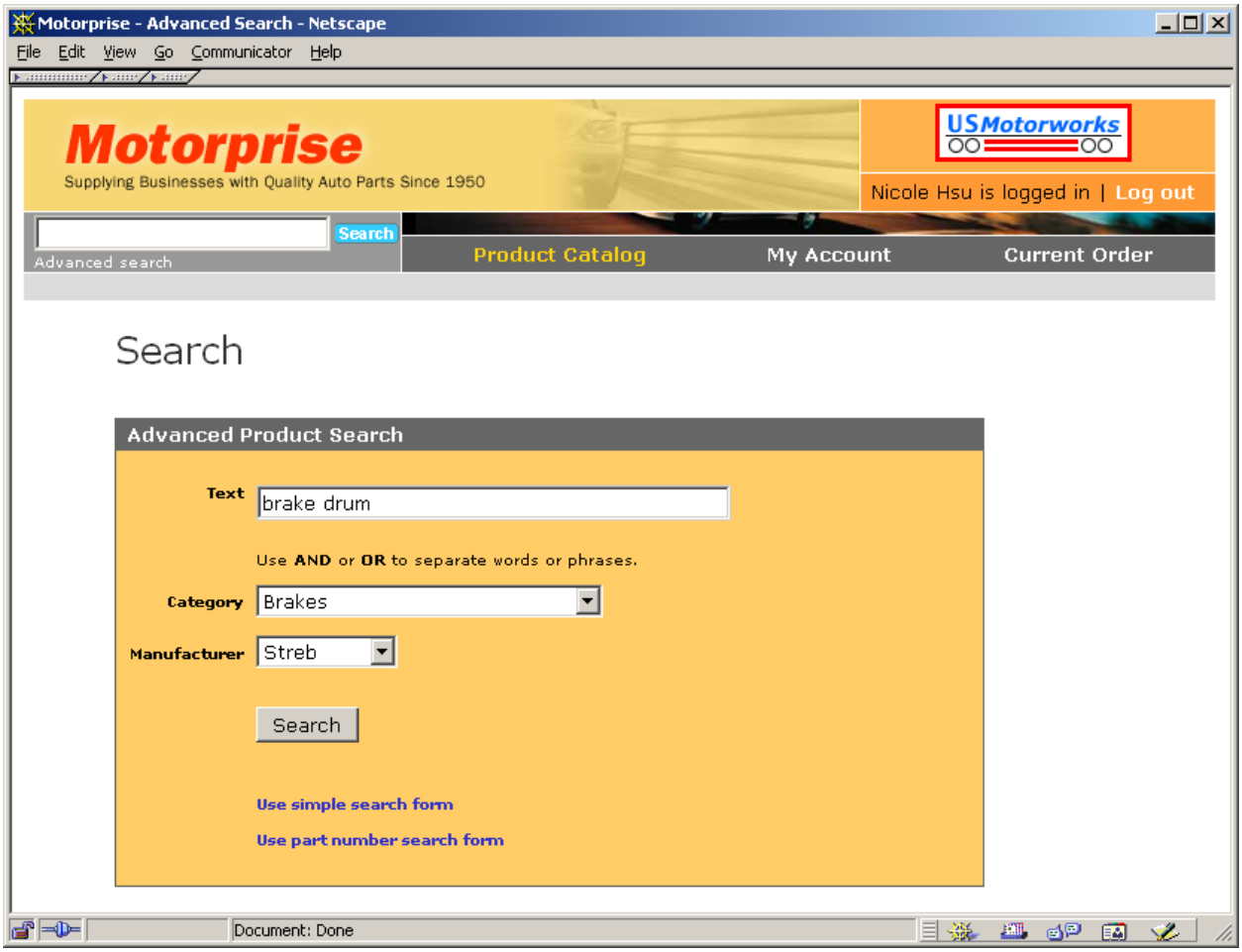
The propertyValuesByType property is a map with property names for keys and arrays of all possible values for the given property. By default, no enumerated property information is available from the propertyValuesByType property of the SearchFormHandler. Instead, you must explicitly set the names of required enumerated properties in the search component's advancedSearchPropertyNames property. In this case, the value is manufacturer.

Thus, when a search is performed in Motorprise the propertyValuesByType map returns two entries. One entry corresponds to the category item-type and the other entry corresponds to the product item-type. This is because these are the two item-descriptor types that are searched on in the ProductCatalog repository. Associated with the category entry will be a list of category repository items that were found in the search; similarly the product key will contain a list of product repository items.

By having the list of repository items split up by type, we are able to control how a repository item is displayed. For example, when a matching product repository item gets displayed, we want to link to product.jsp; for categories we want to link to category.jsp. That is why in the simple_search.jsp example shown above, there is a switch statement on the key property of the propertyValuesByType array.

The search form's categories in the advanced search, manufacturer, is populated with valid choices for each item. These choices (<select> components) are loaded from the repository rather than coded into the page so that the page need not be changed when changes are made to the repository.

To display the list of all categories, we used the RepositoryValues servlet bean. It takes the input parameter itemDescriptorName that in this case is set to category because that is the type of repository item that we want to list. The servlet bean outputs an array of repository items that can be rendered using the ForEach component.



Using advanced search.

The following is part of the code for category search in `advanced_search.jsp`:

```
<!--search box starts here-->
  <table width=100% bgcolor="#FFCC66" border=0 cellpadding=0 cellspacing=0>
    <tr bgcolor="#666666">
      <td bgcolor="#666666" width=1><dsp:img src="../images/d.gif"
        width="1"/></td>
      <td colspan=2>
      <table cellpadding=3 cellspacing=0 border=0>
        <tr><td class=box-top> Advanced Product Search</td></tr>
      </table>
      </td>
    </tr>

    <tr>
```

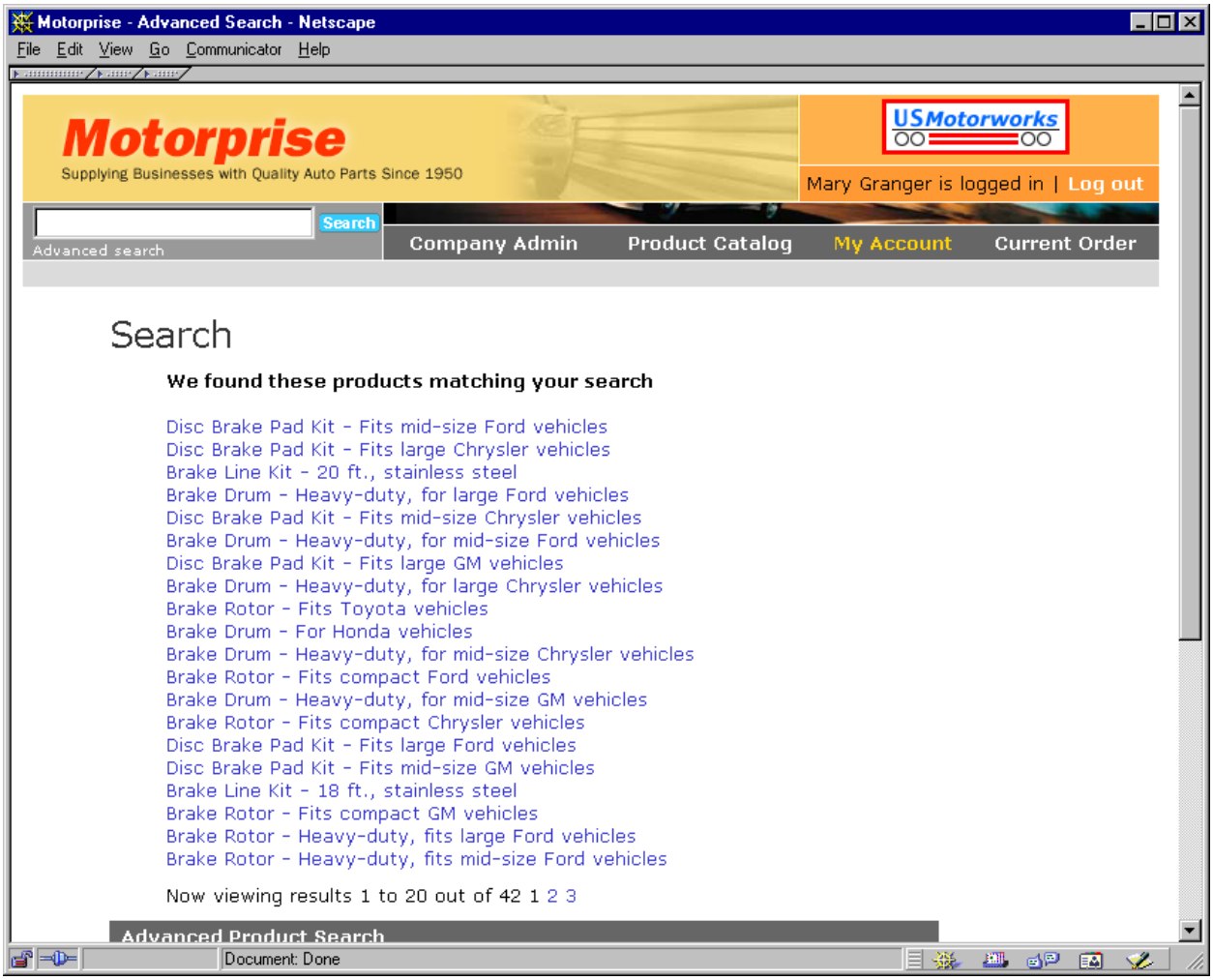
```
<td bgcolor="#666666" width=1><dsp:img src="../images/d.gif"
  width="1"/></td>
<td>
<table width=100% bgcolor="#FFCC66" border=0 cellpadding=0 cellspacing=0>
  <tr valign=top>
    <td width=50%>
      <dsp:form action="advanced_search.jsp" method="POST">
      <table width="100%" border="0" cellspacing="0" cellpadding="4"
        bgcolor="#FFCC66">
        <tr><td colspan=2><dsp:img src= "../images/d.gif"
          height="10"/></td></tr>

        <tr valign=top>
          <td width="15%" align="right"><span class=
            "smallb">Text</span></td>
          <td><dsp:input bean="AdvProductSearch.searchInput" size="25"
            type="text"/>  <br> <span class="small">Use <b>AND</b> or
            <b>OR</b> to separate words or phrases.</span>
          </td>
        </tr>
        <tr>
          <td align="right"><span class=smallb>Category</span></td>
          <td>
          <dsp:select bean="AdvProductSearch.hierarchicalCategoryId">
            <dsp:option value=""/>-- All categories --
            <dsp:droplet name="RepositoryValues">
              <dsp:param name="itemDescriptorName" value="category"/>
              <dsp:oparam name="output">
                <dsp:droplet name="ForEach">
                  <dsp:param name="array" param="values"/>
                  <dsp:param name="sortProperties" value="+displayName"/>
                  <dsp:oparam name="output">
                    <dsp:getvalueof id="elem"
                     idtype="atg.repository.RepositoryItem" param="element">
                      <dsp:option value="<%=elem.getRepositoryId()%>"/>
                      <dsp:valueof param="element.displayName"/>
                    </dsp:getvalueof>
                  </dsp:oparam>
                </dsp:droplet>
              </dsp:oparam>
            </dsp:droplet>
          </dsp:select>
          </td>
        </tr>
        <tr>
          <td align="right"> <span class=
            smallb>Manufacturer</span></td>
          <td>
          <dsp:select bean="AdvProductSearch.propertyValues.manufacturer">
            <dsp:option value=""/>-- Any --
```

```
      <dsp:droplet name="RepositoryValues">
        <dsp:param name="itemDescriptorName" value="product"/>
        <dsp:param name="propertyName" value="manufacturer"/>
        <dsp:oparam name="output">
          <dsp:droplet name="ForEach">
            <dsp:param name="array" param="values"/>
            <dsp:param name="sortProperties" value="+displayName"/>
            <dsp:oparam name="output">

              <dsp:getvalueof id="elemName" idtype="java.lang.String"
                param="element.displayName">
                <dsp:option value="<%=elemName%>"/>
                <dsp:valueof param="element.displayName"/>
              </dsp:getvalueof>

            </dsp:oparam>
          </dsp:droplet>
        </dsp:oparam>
      </dsp:droplet>
    </dsp:select>
    </td>
  </tr>
  <tr>
    <td> </td>
    <td><br>
    <dsp:input bean="AdvProductSearch.currentResultPageNum"
      type="hidden" value="1"/>
    <input name="repositoryKey" type="hidden" value="<dsp:valueof
      bean="/OriginatingRequest.requestLocale.locale"/>">
    <input name="noCrumbs" type="hidden" value="<dsp:valueof
param="noCrumbs"/>">
    <dsp:input bean="AdvProductSearch.search" type="hidden"
      value="Search"/>
    <dsp:input bean="AdvProductSearch.search" type="submit"
      value="Search"/>
    </td>
  </tr>
  <tr>
    <td colspan=2> </td>
  </tr>
  <tr>
    <td> </td>
    <td><span class=smallb><dsp:a href="simple_search.jsp">
           <dsp:param name="noCrumbs" param="noCrumbs"/>
           Use simple search form</dsp:a></span>
    </td>
  </tr>
```

Displaying Search Results

The `SearchFormHandler` that comes with ATG only allows you to navigate through your search results if you searched on one item type. Advanced search only searches products. Simple search searches both products and categories and thus does not use the navigation features.

We configured `SearchFormHandler` to return only twenty search results at a time for advanced searches and included text that informs the user which results he is viewing and how many there are total. (Simple search lists all results on one page.)



Advanced search displays twenty results at a time.

We used the properties `enableCountQuery` and `maxResultsPerPage` to enable page navigation and set the number of results to be returned. The advanced search and the search using manufacturer part number are configured to return only twenty results per page. The following code is used to display page navigation URLs in `advanced_search.jsp`.

```
<dsp:droplet name="Compare">
  <dsp:param bean="AdvProductSearch.resultSetSize" name="obj1"/>
  <dsp:param bean="AdvProductSearch.maxResultsPerPage" name="obj2"/>
  <dsp:oparam name="greaterthan">
    Now viewing results
    <b><dsp:valueof bean="AdvProductSearch.startCount"/> -
    <dsp:valueof bean="AdvProductSearch.endIndex"/></b>
    out of
    <b><dsp:valueof bean="AdvProductSearch.resultSetSize"/></b>
  </dsp:oparam>
</dsp:droplet>

<dsp:droplet name="Switch">
 <dsp:param bean="AdvProductSearch.resultPageCount" name="value"/>
 <dsp:oparam name="1">
 </dsp:oparam>

 <dsp:oparam name="default">
  <br>Results pages:
  <dsp:droplet name="Switch">
     <dsp:param name="value" bean="AdvProductSearch.currentResultPageNum"/>
     <dsp:oparam name="1">

     </dsp:oparam>
     <dsp:oparam name="default">
       <dsp:droplet name="For">
         <dsp:param name="howMany" bean="AdvProductSearch.resultPageCount"/>
         <dsp:oparam name="output">
           <dsp:droplet name="Switch">
             <dsp:param name="value" bean=
               "AdvProductSearch.currentResultPageNum"/>
             <dsp:getvalueof id="countParam" idtype="Integer" param="count">
               <dsp:oparam name="<%=countParam.toString()%>">
                 <dsp:a href="advanced_search.jsp" bean=
                   "AdvProductSearch.currentResultPageNum"
                   paramvalue="index">&lt;&lt; Previous</dsp:a>  
               </dsp:oparam>
             </dsp:getvalueof>
           </dsp:droplet>
         </dsp:oparam>
       </dsp:droplet>
     </dsp:oparam>
   </dsp:droplet>
```

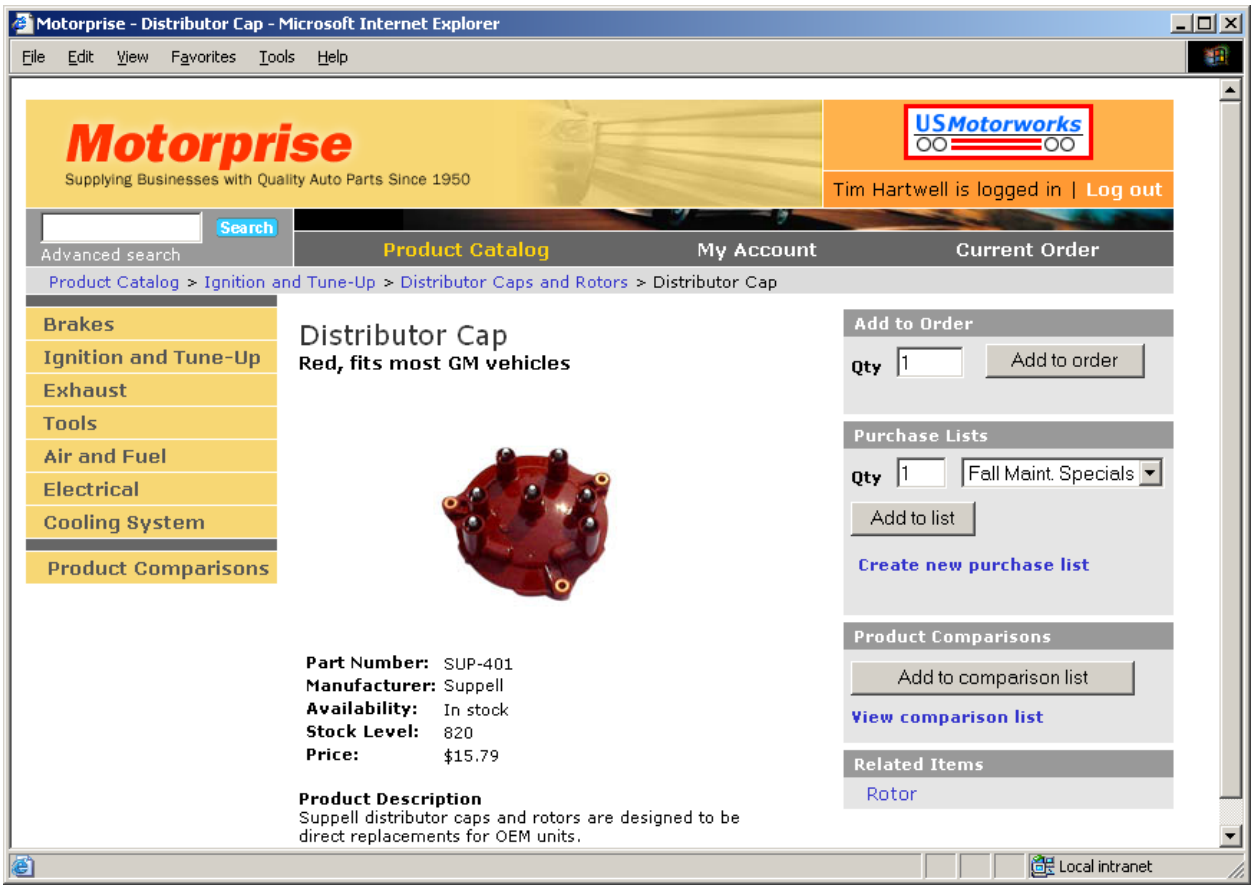
The `currentResultPageNum` property of `SearchFormHandler` determines which set of results to display. We iterate through `resultPageCount`, displaying a URL for each page, and when the user clicks on any URL, the corresponding number is set to `SearchFormHandler.currentResultPageNum` and it returns the appropriate set.

Comparing

The Motorprise site includes a product comparison feature that allows users to see the attributes of several products at once. Users can add products to a comparison list from a product page. They can also search for products on the Product Comparisons page and add to the comparison list from the search results.

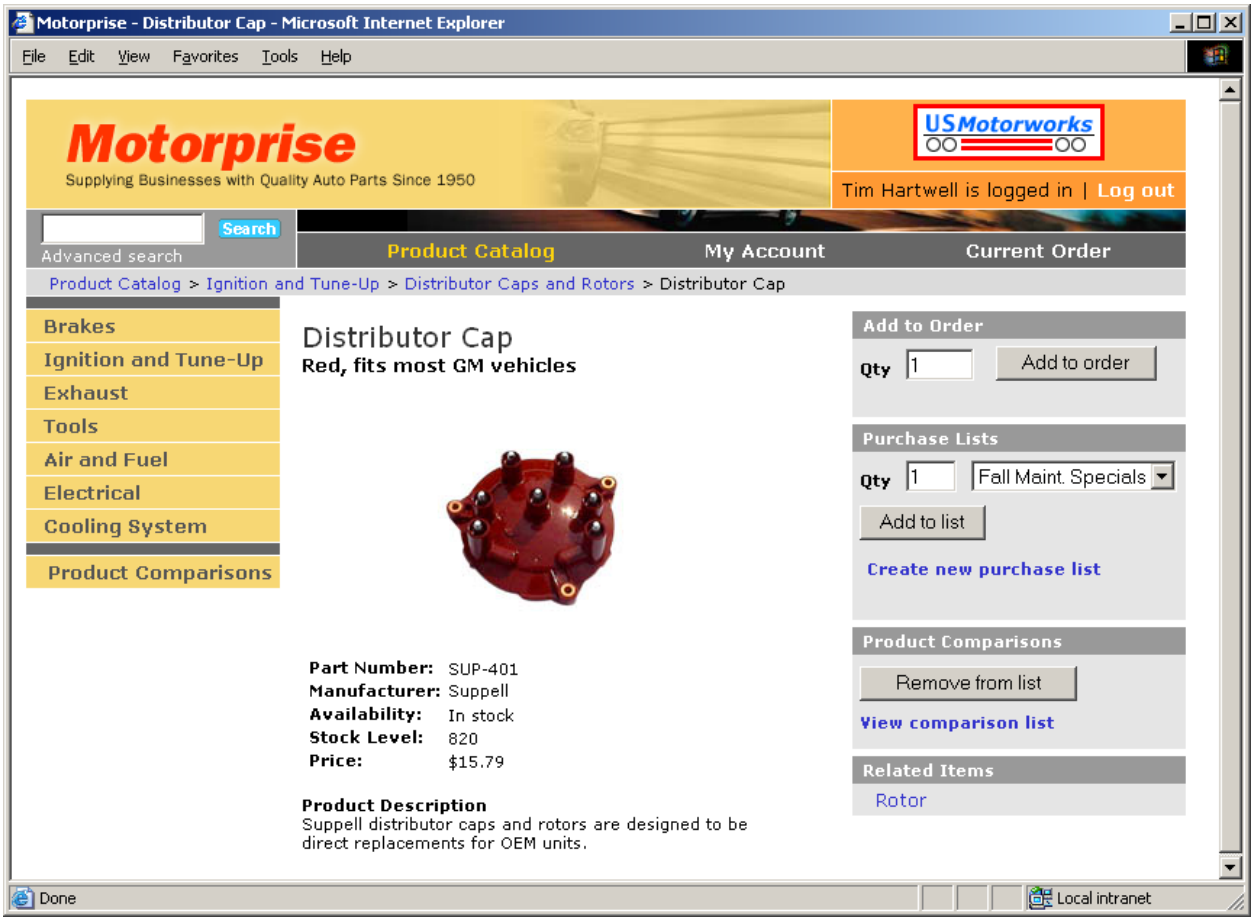
Adding Products to the Comparison List from the Product Page

On the product page, we used a simple form to display an **Add to comparison list** button. We used the `ProductListHandler.addProduct` method to add the product the user is viewing to a list of products to compare.



The Product page includes a section for adding items to and viewing the comparison list.

A **View comparison list** link is displayed on the page if there are products in the comparison list. Once a user adds a product to the list, the button is changed to display **Remove from list** and to use the `ProductListHandler.removeProduct` method so that the product can be removed from the list.



When a product is on the comparison list, the button changes to display "Remove from list."

Below is the code we used to implement the Product Comparisons section on the product page.

```
<dsp:form action="product.jsp" method="post">
  <tr>
    <td bgcolor="#E5E5E5">
    <input name="id" type="hidden" value="<dsp:valueof
      param="element.repositoryId"/>">
    <dsp:input bean="CartModifierFormHandler.SessionExpirationURL" type="hidden"
      value="../common/session_expired.jsp"/>
    <dsp:input bean="CartModifierFormHandler.productId"
      paramvalue="product.repositoryId" type="hidden"/>

    <dsp:droplet name="ProductListContains">
      <dsp:param bean="ProductList" name="productList"/>
      <dsp:param name="productID" param="element.repositoryId"/>
      <dsp:oparam name="true">
```

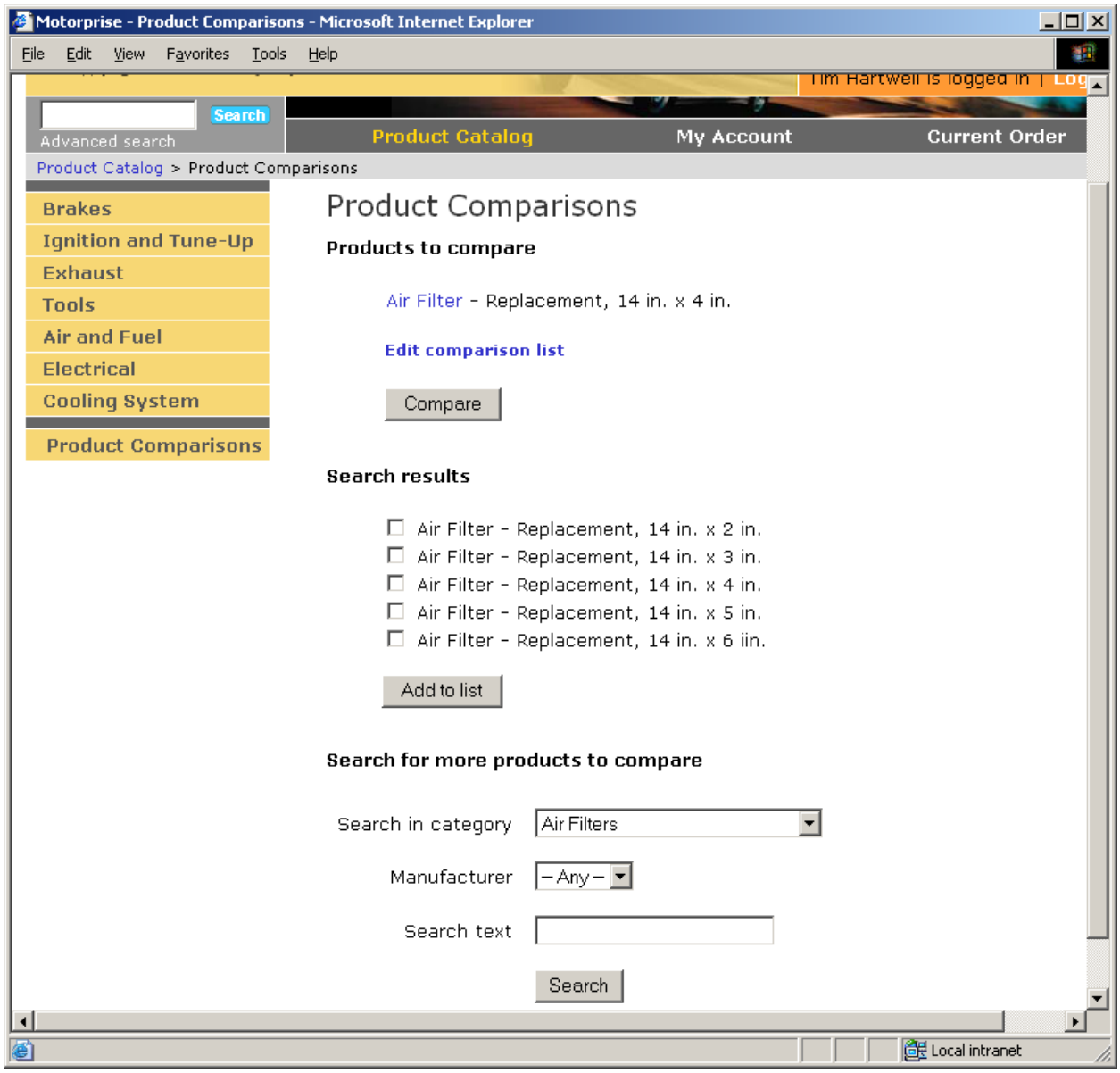
```
        <dsp:input bean="ProductListHandler.productID" paramvalue="productID"
          type="hidden"/>
        <dsp:input bean="ProductListHandler.removeProduct" type="submit"
          value="Remove from list"/>
      </dsp:oparam>
      <dsp:oparam name="false">
        <dsp:input bean="ProductListHandler.productID" paramvalue="productID"
          type="hidden"/>
        <dsp:input bean="ProductListHandler.addProduct" type="submit" value="Add
          to comparison list"/>
       </dsp:oparam>
     </dsp:droplet>
  </td>
</tr>
<!--check to see if there are items in comparison list-->
<dsp:droplet name="IsEmpty">
  <dsp:param bean="ProductList.items" name="value"/>
  <dsp:oparam name="false">
<tr>
  <td><span class=smallb><dsp:a href="compare.jsp">View comparison
    list</dsp:a></td>
</tr>
  </dsp:oparam>
  </dsp:droplet>
  </dsp:form>
```

Using the Product Comparisons Page

The **Product Comparisons** link under the list of categories on the left catalog navigation bar directs users to the Product Comparisons page where they can search for products to compare by category, manufacturer, and/or text.

The Product Comparisons page has three sections:

- the list of products to compare (if the user has added any)
- a search form
- the search results (if a search was performed)



The Product Comparisons page.

Searching for Products to Compare

Customers can use the search form to find products by entering a category, a manufacturer, text, or any combination of those. The search results are displayed above the search box with check boxes so they can be added to the comparison list.

For the search form we used an instance of the `atg.commerce.catalog.custom.SearchFormHandler` from the ATG Business Commerce and configured it for comparisons. We set it to look for products, to

search in names and descriptions, and to do a hierarchical search to allow it to search in particular categories.

From the results list, products are added to the comparison list using the same method as the product page. Unlike the comparison list, which is session-scoped, the search results list is request-scoped because it is an interim step. After the selected products are added to the compare list, the search results are cleared. We used the `ProductListHandler` to allow users to remove selected items or to clear the entire compare list.

For more information about using the `ProductListHandler` see the *Implementing Product Comparison chapter* in the *ATG Commerce Guide to Setting Up a Store*.

Viewing Compare Results

When users click the **Compare** button, the results are displayed as a list: part #, product name, description, manufacturer, price, and availability. By default the list is sorted by the part number. Users can sort the results by a different category by clicking on any of the column headings. They also can reverse the sorting order from ascending to descending.

Motorprise has a policy that guest (or unregistered) users cannot view pricing information for products. In order to enable guest users to compare products without seeing prices, we used the component `ProductComparisonList` (located at `/atg/commerce/catalog/comparison/ProductList`). It has a reference to a `TableInfo` object that controls the product properties that are displayed when you compare products to each other. By default, the product comparison list is configured to use the component at `/atg/commerce/catalog/comparison/TableInfo`.

For Motorprise, we used two `TableInfo` components, `MemberComparisonTable` and `GuestComparisonTable`, located at `/atg/projects/b2bStore/catalog./MemberComparisonTable` includes the `price` column, while `GuestComparisonTable` does not.

Because Motorprise has multiple locales, we used the `TableInfo.columnHeadings` property to map column names to locale-specific column headings. For more information on localizing table headings, see the *Implementing Sortable Tables* chapter of the *ATG Page Developer's Guide*.

`MemberComparisonTable.properties`

```
$class=atg.service.util.TableInfo
$scope=session

columns=\
     part=sku.manufacturer_part_number,\
     product=productLink ; product.displayName,\
     description=product.description,\
     manufacturer=product.manufacturer.displayName,\
     price=priceInfo.amount,\
     availability=inventoryInfo.availabilityStatusMsg

columnHeadings=\
     part=Part \#,\
```

```
     part_de=Produktnr.,\
     product=Product,\
     product_de=Produkt,\
     description=Description,\
     description_de=Beschreibung,\
     manufacturer=Mfr,\
     manufacturer_de=Hrst.,\
     price=Price,\
     price_de=Preis,\
     availability=Availability,\
     availability_de=Verfügbarkeit
```

GuestComparisonTable.properties

```
$class=atg.service.util.TableInfo
$scope=session

columns=\
     part=sku.manufacturer_part_number,\
     product=productLink ; product.displayName,\
     description=product.description,\
     manufacturer=product.manufacturer.displayName,\
     availability=inventoryInfo.availabilityStatusMsg

columnHeadings=\
     part=Part \#,\
     part_de=Produktnr.,\
     product=Product,\
     product_de=Produkt,\
     description=Description,\
     description_de=Beschreibung,\
     manufacturer=Mfr,\
     manufacturer_de=Hrst.,\
     availability=Availability,\
     availability_de=Verfügbarkeit
```

Notice that the entry for `Product=` uses the expression `productLink ; product.displayName` as the property name for the column. This tells the `TableInfo` component that we want to display the property `productLink` in the table, but sort on the product's `displayName`. We did this because `productLink` is an HTML anchor tag that specifies a link to the product's catalog page. Typical values for `productLink` might look like this:

```
<dsp:a href="/Motorprise/en/catalog/product.jsp?id=prod70004"/>Wheel
   Bearing</desp:a>
<dsp:a href="/Motorprise/en/catalog/product.jsp?id=prod110145"/>Disk Brake
   Pad Kit</dsp:a>
```

If we sorted entries based on the literal text of these anchor tags, the Wheel Bearing entry would come before the Disk Brake Pad Kit because its product ID comes first. Using the ability of `TableEntry` to specify a separate sort property allows us to sort these items based on the display name the user sees, regardless of the literal text of the links.

We used the following code in `compare_result.jsp` to select which table is presented:

```
<dsp:importbean bean="/atg/commerce/catalog/comparison/ProductList"/>
<dsp:importbean bean="/atg/userprofiling/Profile"/>
<dsp:importbean bean="/atg/projects/b2bstore/catalog/MemberComparisonTable"/>
<dsp:importbean bean="/atg/projects/b2bstore/catalog/GuestComparisonTable"/>
<%/*
  First decide which table info object to use when building the product
  comparison table.  Members who are logged in get one table, while guests
  get a different table that omits price information.
*/%>

<dsp:droplet name="Switch">
  <dsp:param bean="Profile.transient" name="value"/>
  <dsp:oparam name="true">
    <dsp:setvalue bean="ProductList.tableInfo" beanvalue="GuestComparisonTable"/>
  </dsp:oparam>
  <dsp:oparam name="false">
    <dsp:setvalue bean="ProductList.tableInfo" beanvalue="MemberComparisonTable"/>
  </dsp:oparam>
</dsp:droplet>
```

This code checks to see if the user is logged in or not, and based on that fact, sets the `tableInfo` property of the product comparison list to either the guest table definition (which does not include a price column) or the member table definition (which does). The rest of the page can then refer to `ProductList.tableInfo` and get a `TableInfo` component that applies to the current user, whether the user is logged in or not.

In general, Motorprise displays the product comparison table using the patterns and techniques described in the *Implementing Sortable Tables* section of the *ATG Page Developer's Guide*. We applied special logic to the table's `Price` column, however, to ensure that prices are formatted correctly for the user's locale and display an appropriate currency symbol.

You can change the number of table columns and their order by editing the configuration of the corresponding `TableInfo` component. Since we cannot know which column will display the price, we used the following code to check each column heading as we iterate over the values in the table, adding the `currency` tag converter if the column heading is `Price` and the `valueishtml` tag converter for all other values:

```
<dsp:droplet name="ForEach">
<%/*
  Here we are going to iterate over all of the property names
```

```
  we want to display in this row using BeanProperty to display
  each one.  We handle the "Price" column specially, using a
  currency converter on the value to get properly localized
  currency symbols and formatting.
  */%>
  <dsp:param bean="ProductList.tableColumns" name="array"/>
  <dsp:param name="sortProperties" value=""/>
  <dsp:oparam name="output">
    <td><dsp:droplet name="BeanProperty">
    <dsp:param name="bean" param="currentProduct"/>
    <dsp:param name="propertyName" param="element.property"/>
    <dsp:oparam name="output">
      <dsp:droplet name="Switch">
        <dsp:param name="value" param="element.name"/>
        <dsp:oparam name="price"><dsp:valueof converter="currency"
          param="propertyValue"/></dsp:oparam>
         <dsp:oparam name="default"><dsp:valueof valueishtml="<%=true%>"
           param="propertyValue"/></dsp:oparam>
      </dsp:droplet>
    </dsp:oparam>
    </dsp:droplet></td>
  </dsp:oparam>
</dsp:droplet>
```

Note that the name used in the JSP must match the name used in the `TableInfo` component for this logic to work correctly. For example, for Motorprise we edited the `TableInfo` component and changed the column names for its English and German catalog pages. Thus, we had to change the column name used in `compare_result.jsp` to match.

8 B2B Personalization

Motorprise is designed to give each of its customer organizations a customized experience. We created a contract for each customer organization that specifies its particular catalog and price lists, including volume pricing. This chapter contains the following sections:

Creating Custom Catalogs
Describes how we created custom catalogs for Motorprise customers USMW and NDAP.

Creating Price Lists
Describes how we set up custom price lists for Motorprise customers USMW and NDAP.

Using Volume Pricing
Explains how we set bulk and tiered pricing using the ACC.

Creating Contracts
Describes how we set up contracts for USMW and NDAP.

Creating Custom Catalogs

For Motorprise, we customized the catalogs for users of USMW and NDAP to display only the categories and products that are relevant to their businesses.

We set up a folder structure of Master Catalogs, Subcatalogs, and Customer Catalogs. Then, in the Subcatalog folder, we created all of our subcatalogs - one for each main or root category. For example, we have a subcatalog called "Brakes." In this main category, we placed all the products that are brake-related, such as brake lines and drums.

We set up our catalog this way for several reasons:

- to easily show certain groups of products to certain customers.
- to be able to share categories among customers so that we wouldn't have to create a separate catalog for each customer.
- to use the same categories to build our master catalog.
- to allow Motorprise product specialists to manage categories.

These business needs are specific to Motorprise. You can create and manage a catalog structure that specifically fits your business strategy using ATG Business Commerce's flexible catalog management functionality.

In each subcatalog, we created categories that make up the catalog structure; each of these categories has multiple subcategories. Products and SKUs are created in these subcatalogs and reside there.

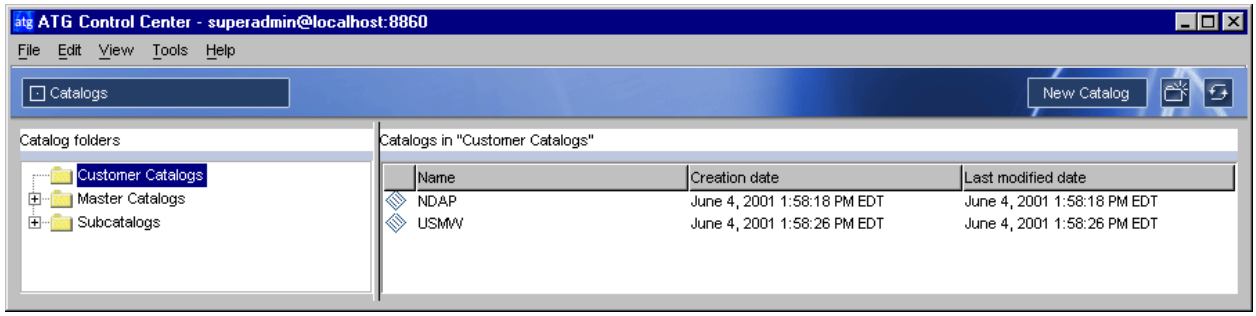
For example, the Electrical subcatalog has a category called Bulbs that contains multiple products, all of which are named "bulb". For each of these products, we can manage its properties and images, define cross-sell relationships, administer product variations or SKUs, and also view the other catalogs that include this particular product.

(Catalogs cannot share categories. However, they can share other catalogs. Thus, both USMW and NDAP can share the categories *within* a subcatalog, such as Electrical.)

Next, we created a Base Catalog in the Master Catalog folder and added all the subcatalogs to it. You can view all the categories and products in the Base Catalog and use the **Click here** link to go to the subcatalogs to edit them.

Users of the Motorprise site who do not have a catalog, such as anonymous users, see the Base Catalog by default. For more information on this feature, see the section on the Catalog scenario in the Merchandising chapter of this guide.

We then created customer catalogs for USMW and NDAP. USMW has the following categories: Electrical, Cooling System, Brakes, Ignition and Tune-Up, Exhaust, and Air and Fuel. NDAP has the following categories: Belts and Hoses, Electrical, Cooling System, Brakes, Ignition and Tune-Up, Tools, and Air and Fuel. The customer catalogs for USMW and NDAP are built of various subcatalogs and thus contain all the categories, products, and SKUs within them. We determined which subcatalogs they would use based on their business requirements. For example, USMW does not purchase Tools from Motorprise so they don't see that category.



Catalogs in Motorprise.

In the ACC, you can view all the products in these customer catalogs. You can also access and edit products through the **Click Here** link in any catalog.

For more information see the *ATG Commerce Programming Guide*.

Localizing Catalogs

Motorprise is designed to support two languages: English and German. The content of the site is available in these two languages. We achieved this localization by adding German attributes to the catalog.

We used derived properties and a derivation method called `FirstWithLocale` to accomplish this.

Let's use `category.displayName` in the Motorprise catalog, `<ATG10dir>/Motorprise/config/atg/commerce/catalog/custom/customCatalog.xml`, as an example. This is the name of the category that displays on catalog pages, such as "Brakes" or "Tools."

First, we removed the existing definition of `displayName` from the default catalog schema:

```
<table name="dcs_category" type="primary" id-column-name="category_id">
  <property name="displayName" xml-combine="remove" />
```

The `remove` command must be in the correct table tag; otherwise the definition of `displayName` is not properly removed. This command will override the DCS table.

Our reason for removing this is that we are going to replace this "generic" property and replace it with an English display name and a German display name.

Then, inside the same table tag, we created a definition for the English display name. We reused the same column in the same database; that way, we didn't have to do any data migration. The English value for `displayName` is already in this column. We call this new property `displayName_en` and then use an `attribute` tag to assign it a `locale` with a value of en.

```
<property name="displayName_en" data-type="string" column-name=
   "display_name" required="true" queryable="true" category-
   resource="categoryPresentation" display-name-resource="displayName">
 <attribute name="propertySortPriority" value="-8" />
 <attribute name="locale" value="en" />
</property>
```

Note that we used the same `category-resource` and `display-name-resource` for `displayName_en` that we used for `displayName`. You will notice below that we also used the same ones for `displayName_de`.) The ACC is able to recognize when a property has a `locale` attribute. It appends this attribute to the end of the resource string. For example, if the resource string is "Display name" then the English version will be "Display name (en)". The most important part of this property definition is the `locale` attribute. You could use a more specific value such as "en_US" if necessary. The derivation method, which is described below, recognizes either.

Now that we have an English display name, for Motorprise we also want a German display name.

First, we created a new table, `dbc_category_de`, in the database to store all German properties for a category (except the multi-value properties like `keywords`.)

Inside the new table tag we create the German display name, `displayName_de`:

```
<table name="dbc_category_de" type="auxiliary" id-column-name=
   "category_id">
   <property name="displayName_de" data-type="string" column-name=
      "display_name" queryable="true" category-resource=
      "categoryPresentation" display-name-resource="displayName">
      <attribute name="propertySortPriority" value="-8" />
      <attribute name="locale" value="de" />
  </property>
</table>
```

Now we have an English display name and a German display name. We want the correct language to be chosen based on the user's locale when they enter the site. To do this, we recreate the `displayName` property and use the `FirstWithLocale` derivation method. (We used the same name so we wouldn't need to make any changes to the JSPs.)

```
<property name="displayName" data-type="string" category-resource=
   "categoryPresentationDerived" writable="false" display-name-resource=
   "displayName" queryable="true">
  <derivation user-method="atg.commerce.dp.FirstWithLocale">
 <expression>displayName_en</expression>
 <expression>displayName_de</expression>
 </derivation>
 <attribute name="resourceBundle"
     value="atg.projects.b2bstore.CustomCatalogTemplateResources" />
 <attribute name="propertySortPriority" value="-8" />
 <attribute name="keyService" value="/atg/userprofiling/LocaleService" />
 <attribute name="defaultKey" value="en_US" />
 </property>
```

This property is not in a table tag since it is not stored in the database. The `keyService` is a Nucleus component that tells us the locale of the current user. The `defaultKey` is used if there is no value for the locale that is provided by the key service. For example, if the user's locale were `sn_ZW`, he or she would see the English display name, since we have not defined a display name for Shona speakers in Zimbabwe. The derivation method will first look for the full locale (such as de_DE) returned by the key service. If it can't find anything, it then looks for just the language part of the locale. This is why we can set the locale of `displayName_de` to de and it will still be used if locale is de_DE.

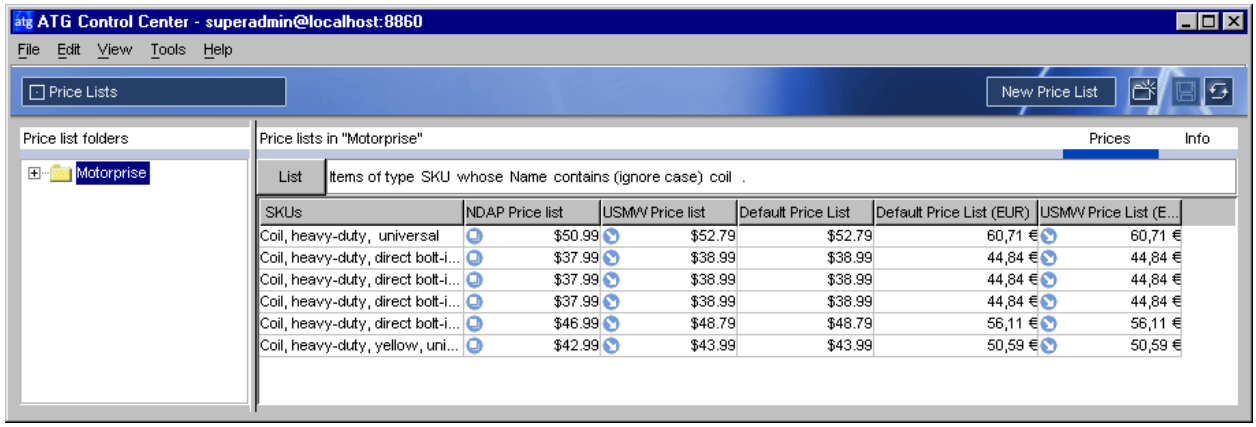
Here are the definitions of the German properties for "category". Notice that everything except "keywords" is in one table.

```
<!-- create german versions in a separate table -->
<table name="dbc_category_de" type="auxiliary" id-column-name=
   "category_id">

  <property name="displayName_de" data-type="string" column-name=
   "display_name" queryable="true" category-resource=
```

```
   "categoryPresentation" display-name-resource="displayName">
    <attribute name="propertySortPriority" value="-8"/>
    <attribute name="locale" value="de"/>
  </property>

  <property name="description_de" data-type="string" column-name=
     "description" queryable="true" category-resource=
     "categoryPresentation" display-name-resource="description">
    <attribute name="propertySortPriority" value="-7"/>
    <attribute name="locale" value="de"/>
  </property>

  <property name="longDescription_de" data-type="string" column-
     name="long_description" queryable="true" category-resource=
     "categoryPresentation" display-name-resource="longDescription">
    <attribute name="propertySortPriority" value="-6"/>
    <attribute name="locale" value="de"/>
  </property>

  <property name="template_de" item-type="media" column-name="template_id"
     queryable="true" category-resource="categoryPresentation"
     display-name-resource="template">
    <attribute name="propertySortPriority" value="-5"/>
    <attribute name="locale" value="de"/>
  </property>

</table>

<!-- create german keywords -->
<table name="dbc_cat_key_de" type="multi" id-column-name="category_id"
   multi-column-name="sequence_num">

  <property name="keywords_de" data-type="list" component-data-type=
     "string" column-name="keyword" category-resource=
     "categoryPresentation" display-name-resource="keywords">
     <attribute name="propertySortPriority" value="-4"/>
     <attribute name="locale" value="de"/>
  </property>
</table>
```

Creating Price Lists

Motorprise often negotiates special pricing for its customers. To manage this special pricing, we created various price lists in Motorprise. The defaultPriceList is used if there is no other price in another customer price list. We also set up custom price lists for USMW and NDAP to manage their prices. For example, if you look in the Pricing > Price Lists area of the ACC, and search for SKUs whose names contain "Coil," you will see the customized pricing.



Motorprise offers customized pricing to USMW and NDAP.

For more information, see the *ATG Commerce Programming Guide* and the *ATG Commerce Guide to Setting Up a Store*.

Localizing Price Lists

We also created a list for use by the USMW German Division called the USMW Price List (Eur). It is identical to the defaultPriceList but converted into Euros. Both of these lists have a `locale` property. For the USMW German prices, we set this to `de_DE_EURO` since we defined the prices in Euros. For the US dollar prices, the locale is `en_US` since the US dollar is the default currency for this locale. The correct price list must be assigned to each user's profile.

A user's price list is defined by the parent organization's contract. Thus, even if a registered user changes his language of choice for viewing the site, he will still see the same prices.

We created a new class, `atg.droplet.CurrencyConversionFormatter`, that combines properties of `CurrencyFormatter`, `CurrencyConversionTagConverter`, and `EuroTagConverter`.

We used this droplet to convert and format the price lists in US dollars and Euros. It takes as inputs the amount to convert, its original locale (that of the price list), and its target locale.

There is one output parameter (in `oparam="output"`) called `formattedCurrency`:

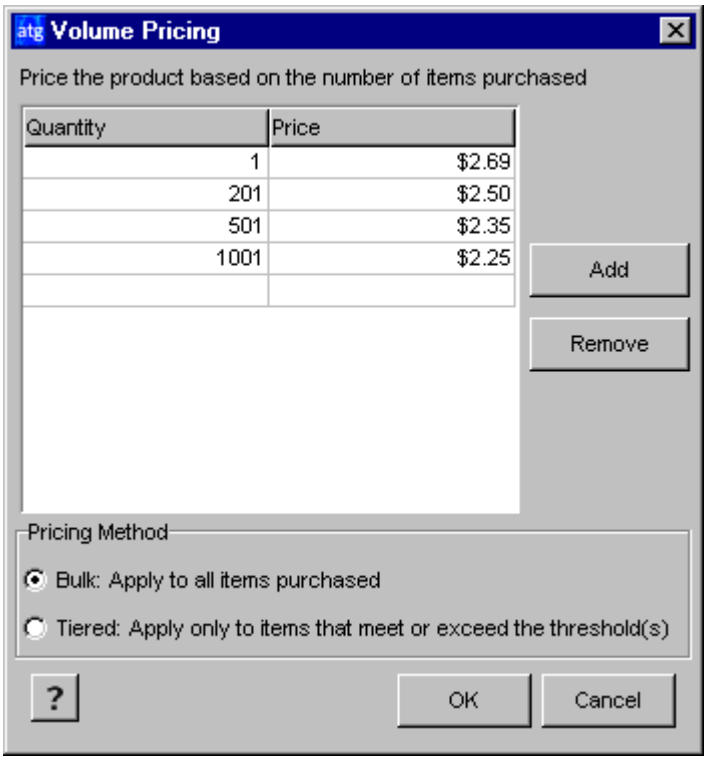
```
<droplet bean="/atg/dynamo/droplet/CurrencyConversionFormatter">
  <param name="currency" value="param:price.listPrice">
  <param name="locale" value="en_US">
  <param name="targetLocale" value="de_DE_EURO">
  <param name="euroSymbol" value="&euro;">
  <oparam name="output">
    <valueof param="formattedCurrency" valueishtml>no price</valueof>
  </oparam>
</droplet>
```

For more information on `CurrencyConversionFormatter`, see the *ATG Commerce Programming Guide*.

Using Volume Pricing

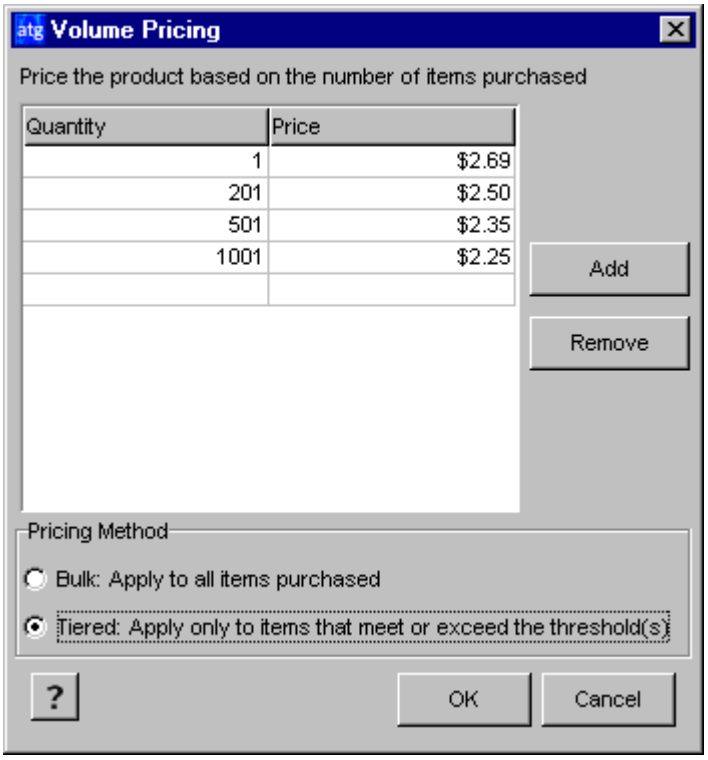
We used volume pricing in Motorprise. ATG Business Commerce provides two kinds of volume pricing:

Bulk: the price applies to all items purchased. For example, in the screenshot below, if you bought 500 items, they would each cost $2.50 apiece.



Setting bulk pricing in the ACC.

Tiered: the price applies only to items that meet or exceed the threshold. For example, in the screenshot below, if you bought 500 items, the first 1- 200 items would cost $2.69 each and the next 300 items would cost $2.50 each.



Setting tiered pricing in the ACC.

We used the ACC to assign volume pricing to certain items. For more information on volume pricing, see the *ATG Commerce Programming Guide*.

Creating Contracts

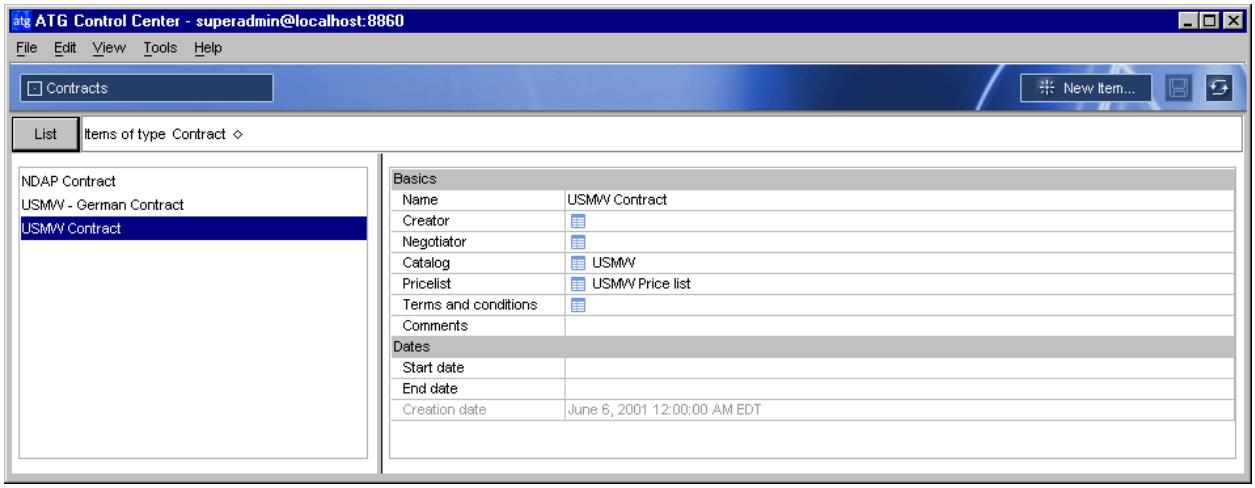
For Motorprise, catalogs are assigned to an organization via a contract. You can create contracts in the Purchases and Payments > Contracts area in the ACC. We used contracts to associate a price list and a catalog with an organization.

In Motorprise, each organization has only one contract. However, you could easily extend your sites to allow multiple contracts per organization and to set the default to the currently active contract when the user logs in.

Each organization has a `Contract` property and it derives its `Catalog` and `Pricelist` properties from the contract.

Users inherit the `Contract` property and thus its `Catalog` and `Pricelist` properties. You could assign catalogs and price lists directly to organizations, rather than using contracts; in that case users would inherit the `Catalog` and `Pricelist` properties directly from the organization.

For more information, see the *ATG Commerce Programming Guide*.



The USMW contract associates USMW with a price list and a catalog.

Localizing Contracts

We created a contract specific to USMW in Germany with the same product catalog, but we assigned it a different price list.

For German Motorprise users, we display pricing on product pages and promotions in both euros and Deutsche marks. However, only euros are display in the checkout, order history, and comparison pages. In these pages, the price is displayed in the actual currency, which is defined by the contract.

9 Processing Orders

This chapter describes the order capture functionality in the Motorprise site and includes the following sections:

Current Order
Describes how orders are edited and saved.

Shipping Information
Explains Motorprise shipping features such as default and alternate addresses, splitting items among addresses, and multiple shipping methods.

Payment Information
Explains Motorprise billing features such as P.O. numbers, requisitions, and cost centers.

Specifying Cost Centers
Explains how users can specify cost centers.

Confirming and Committing Orders
Describes how orders are confirmed and submitted.

Motorprise users have a number of options when placing orders. They can choose among different shipping and billing methods and addresses. They also have the ability to split their orders by line item or dollar amount.

Current Order

In Motorprise, the Current Order page, `checkout/cart.jsp`, displays each line item, its quantity and price, and the current subtotal for the order. Users can change the quantity of any item, remove items, save the order, or proceed to the checkout process.

We used several different form handlers on `checkout/cart.jsp` to provide the utilities for saving and updating orders.

Saving an Order

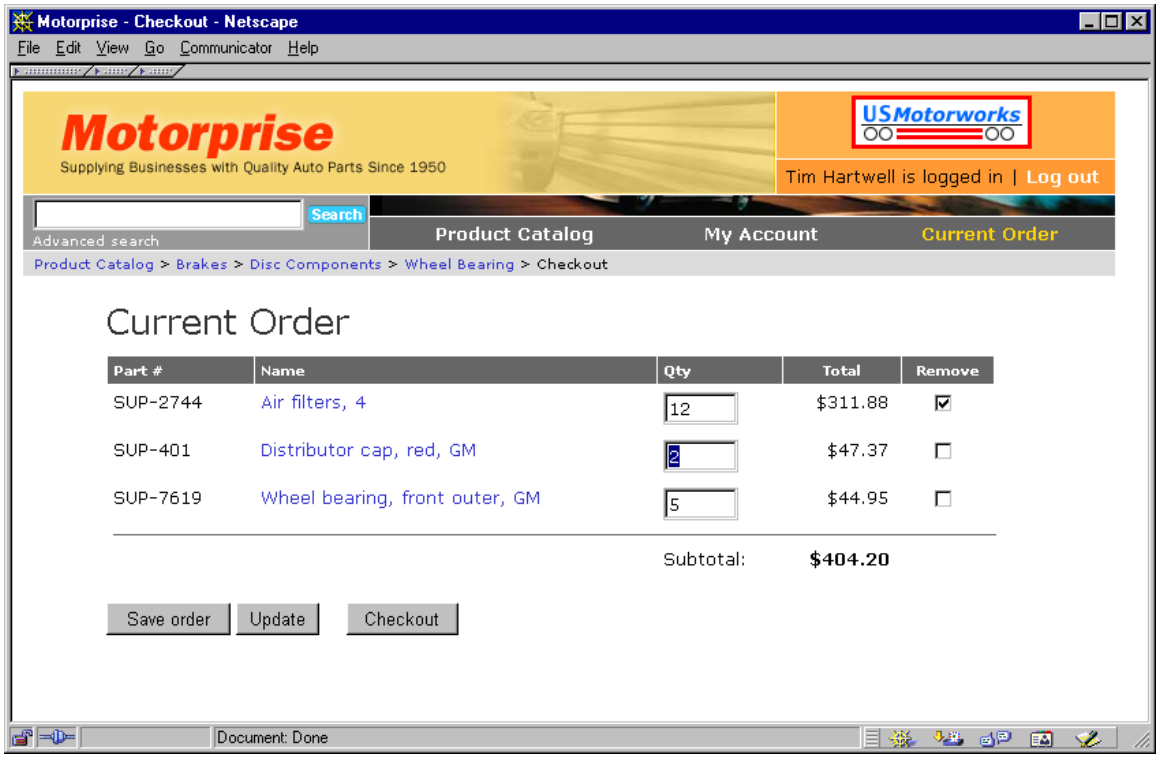
Users can save the items in a current order and then place the order at a future time. The **Save Order** button redirects the user to `user/save_order.jsp`, where we used `/atg/commerce/order/purchase/SaveOrderFormHandler` to save a named order:

```
<dsp:input bean="SaveOrderFormHandler.description" type="text"/>
<dsp:input bean="SaveOrderFormHandler.saveOrder" type="submit" value="Save
   order"/>
<dsp:input bean="SaveOrderFormHandler.saveOrder" type="hidden" value="save"/>
<dsp:input bean="SaveOrderFormHandler.saveOrderSuccessURL" type="hidden"
   value="../user/saved_orders.jsp"/>
<dsp:input bean="SaveOrderFormHandler.saveOrderErrorURL" type="hidden"
   value="../user/save_order.jsp"/>
```

For more information on saving orders, see the My Account chapter of this guide.

Updating an Order

Users can delete items from their orders or change the quantities of items on the Current Order page. They simply enter a new quantity in the Qty field to change the quantity or enter a checkmark in the Remove box to delete a line item. After entering these changes, they click the **Update** button.



*Users can change quantities or remove line items and then click the **Update** button.*

In `checkout/cart.jsp`, we used a `ForEach` droplet to iterate over each line item in the order. The output displayed for each line item includes important information such as the current quantity, price, a link to the referenced product page, and the ability to change its quantity or delete it.

In order to update the order, we used the `/atg/commerce/order/purchase/CartModifierFormHandler.setOrder` handler method in `checkout/cart.jsp`. This handler method determines the submitted quantity of each `CommerceItem`, adjusting those whose quantities have been modified, and removing those whose quantities are not greater than zero.

Changing the Quantity of an Item

To change the quantity of an item in the order, the `CartModifierFormHandler` is provided a request parameter named after the `CommerceItem.catalogRefId`, whose value is the new quantity for that `CommerceItem`:

```
<input type="text" size="3"
  name="<dsp:valueof param='CommerceItem.catalogRefId'/>"
  value="<dsp:valueof param='CommerceItem.quantity'/>">
```

Deleting an Item

To selectively remove items from the order, we set the `CartModifierFormHandler.removalCatalogRefIds` property to an array of the `catalogRefIds`:

```
<%-- Display "remove" checkbox column --%>
<td align=middle>
   <dsp:getvalueof id="skuId" param="CommerceItem.catalogRefId">
      <dsp:input type="checkbox" bean=
         "CartModifierFormHandler.removalCatalogRefIds" value="<%=skuId%>"/>
   </dsp:getvalueof>
</td>
```

Updating the Order

After a user has modified quantities and marked items for removal, he or she may either update the shopping cart page by clicking Update or proceed with the checkout process by clicking Checkout. Both of these submit buttons invoke handlers that modify the order based on the changes in the request. The checkout button also redirects the user to `checkout/shipping.jsp`. The following code snippet illustrates the Update and Checkout handlers:

```
<%-- Update Order button: --%>
<dsp:input bean="CartModifierFormHandler.setOrder" type="submit"
value="Update"/>

<%--
GoTo this URL if user pushes RECALCULATE button and there are no errors:
--%>
<dsp:input bean="CartModifierFormHandler.setOrderSuccessURL" type="hidden"
   value="../checkout/cart.jsp"/> <%/* stay here */%>

<%--
```

```
GoTo this URL if user pushes RECALCULATE button and there are errors:
--%>
<dsp:input bean="CartModifierFormHandler.setOrderErrorURL" type="hidden"
   value="../checkout/cart.jsp"/> <%/* stay here */%>

<%-- CHECKOUT Order button: --%>
      <dsp:input bean="CartModifierFormHandler.moveToPurchaseInfo"
   type="submit" value="Checkout"/>

<dsp:input bean="CartModifierFormHandler.moveToPurchaseInfoSuccessURL"
   type="hidden" value="../checkout/shipping.jsp"/> <%-- move on to shipping --%>
```

For more information, see the *ATG Commerce Programming Guide*.

Shipping Information

Each Motorprise user has a default shipping address and an alternate list of authorized shipping addresses. These addresses are held in the Profile and Organization properties `defaultShippingAddress` and `shippingAddrs`. In Motorprise, they are inherited from the parent organization, and can only be modified by administrators from that organization. Other buyers cannot enter new shipping addresses or modify existing ones. You could configure the ability to edit shipping addresses differently in your own site.

Motorprise allows users to choose among various shipping options to meet their complex business needs. They can:

- ship all items in an order to their default shipping address.
- select a different address from their list of alternates.
- split the line items in an order among various shipping addresses.
- choose among various shipping methods such as two-day or overnight shipping for each address.

All Motorprise products are hard goods that require physical addresses. These addresses are associated with the Order in the form of `HardgoodShippingGroup` objects. We used `/atg/commerce/order/purchase/ShippingGroupDroplet` on `checkout/shipping.jsp` to create `HardgoodShippingGroups` based on the addresses found in the profile. This component also creates a default `HardgoodShippingGroup` with an address based on the user's `defaultShippingAddress` profile property.

These `ShippingGroups` are added to the `ShippingGroupMapContainer`, which is used during the purchase process to store the user's shipping groups. This container is instantiated by Nucleus and is located at `/atg/commerce/order/purchase/ShippingGroupContainerService`.

Additionally, for each line item in the order, the `ShippingGroupDroplet` creates a `CommerceItemShippingInfo` object and adds it to the `CommerceItemShippingInfoContainer`. The `CommerceItemShippingInfo` is a helper object that keeps track of relationships between the order's line

items and the HardgoodShippingGroups that represent the shipping information. We use it to track the quantity of each item that belongs to each ShippingGroup. The container is used during the purchase process to store the helper objects for the user.

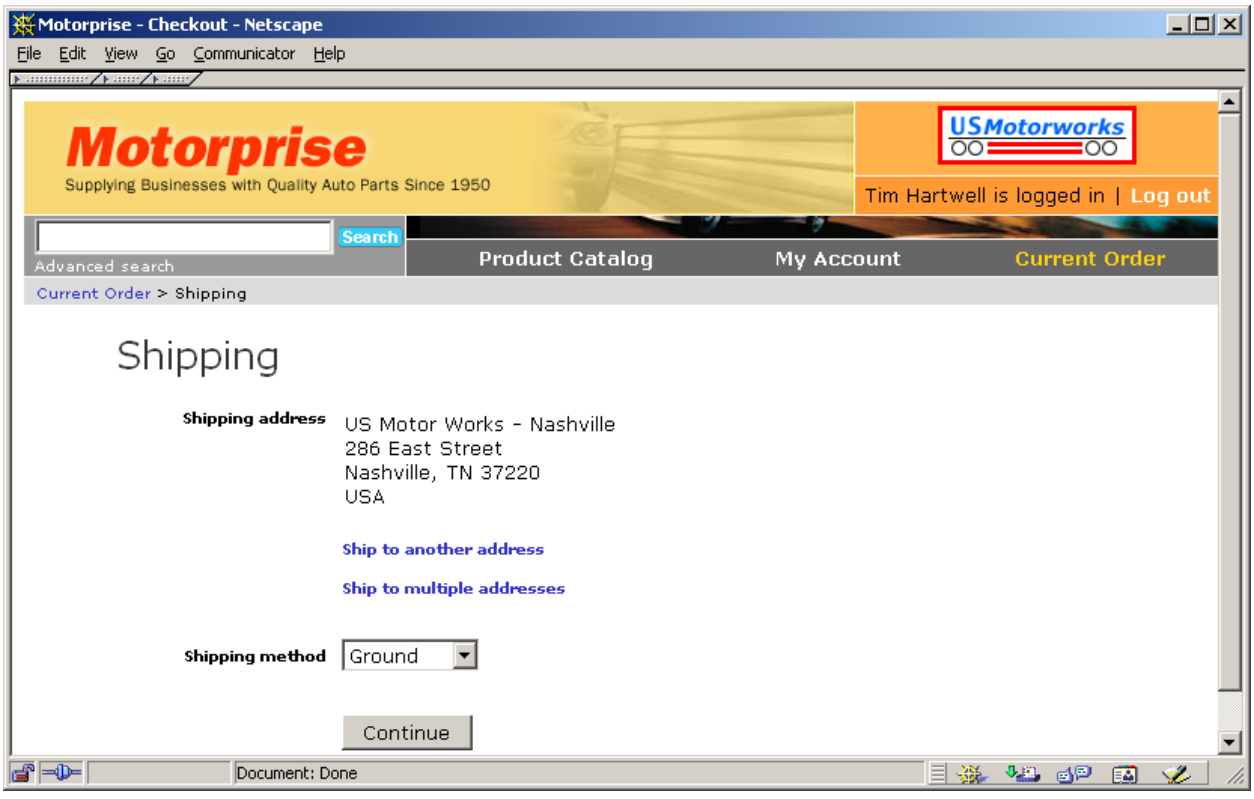
For more information on the ShippingGroupMapContainer, see the API documentation for class atg.commerce.order.purchase.ShippingGroupContainerService.

You can look at checkout/shipping.jsp to see how we used the ShippingGroupDroplet. For more information, see the *ATG Commerce Programming Guide*.

Selecting a Default Address

Users can choose to have all items in their order shipped to the default shipping address. In this case, they only need to select a shipping method and click the **Continue** button. This action invokes the ShippingGroupFormHandler.applyShippingGroups method. The standard ATG Commerce validateShippingInfo pipeline chain is executed and redirects the user to checkout/billing.jsp.

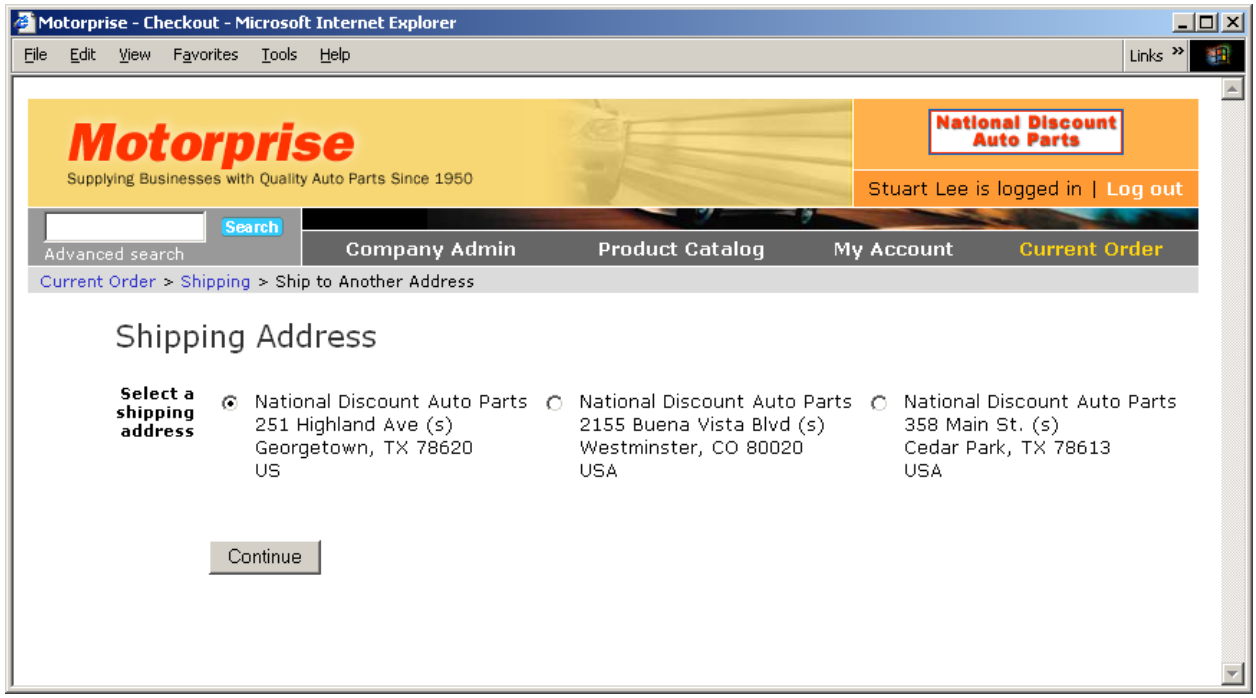
For more information on the validateShippingInfo pipeline chain, see the *ATG Commerce Programming Guide*.



Using a default shipping address.

Shipping to an Alternate Address

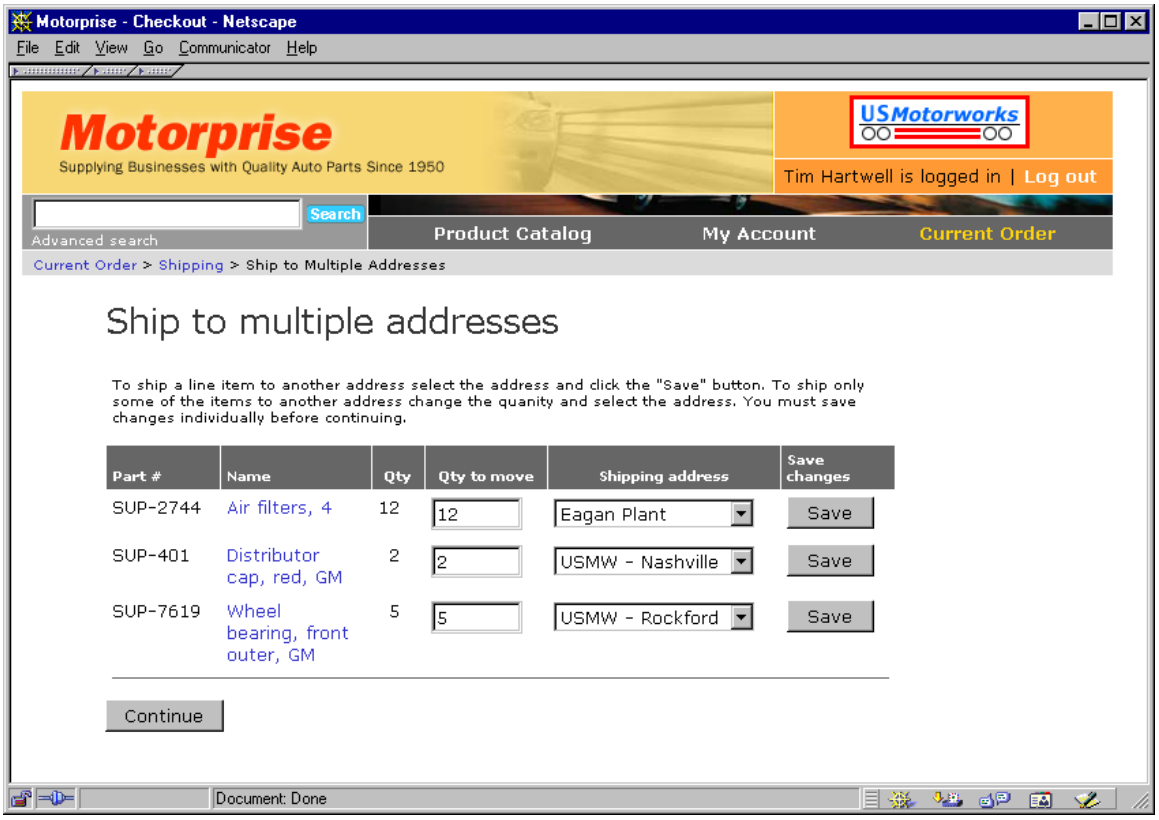
Users can also choose to have all of their items shipped to an alternate shipping address by clicking the **Ship to another address** link. This redirects the user to `checkout/shipping_address.jsp`, where the user is presented with all available addresses. The user can select from any of the available shipping addresses, and continue with selecting a shipping method for this address.



Shipping to Multiple Addresses

Users may decide to ship the items in their order to multiple addresses by clicking the **Ship to multiple addresses** link to `checkout/ship_to_multiple.jsp`.

Users are presented with dropdown lists of their available shipping addresses. They simply select the desired address and the appropriate quantity for each line item, and click the **Save** button.



Assigning multiple shipping addresses.

The `atg/commerce/order/purchase/ShippingGroupFormHandler` component is used to keep track of the user's selections. It maintains a list of associations between the various line items and how they are apportioned among shipping addresses, and allows the user to ship one set of items to one address and another set of items to another address. This component also accommodates shipping different quantities of the same item to different addresses.

First, the `ShippingGroupDroplet` is initialized. The page takes an initialization parameter called `init` that specifies whether the old `ShippingGroupMapContainer` and `CommerceItemShippingInfoContainer` should be cleared and reinitialized. This parameter is set to `true` the first time the page is accessed. As the user modifies the information and the page is refreshed, the parameter is then set to `false` to preserve the user's changes.

Once initialized, the `ShippingGroupMapContainer` contains the user's `ShippingGroups`, and the `CommerceItemShippingInfoContainer` contains one `CommerceItemShippingInfo` object per line item in the order. The following JSP from `ship_to_multiple.jsp` ensures this initialization:

```
<dsp:droplet name="ShippingGroupDroplet">
  <dsp:param name="clearShippingGroups" param="init"/>
  <dsp:param name="initShippingGroups" param="init"/>
  <dsp:param name="shippingGroupTypes" value="hardgoodShippingGroup"/>
```

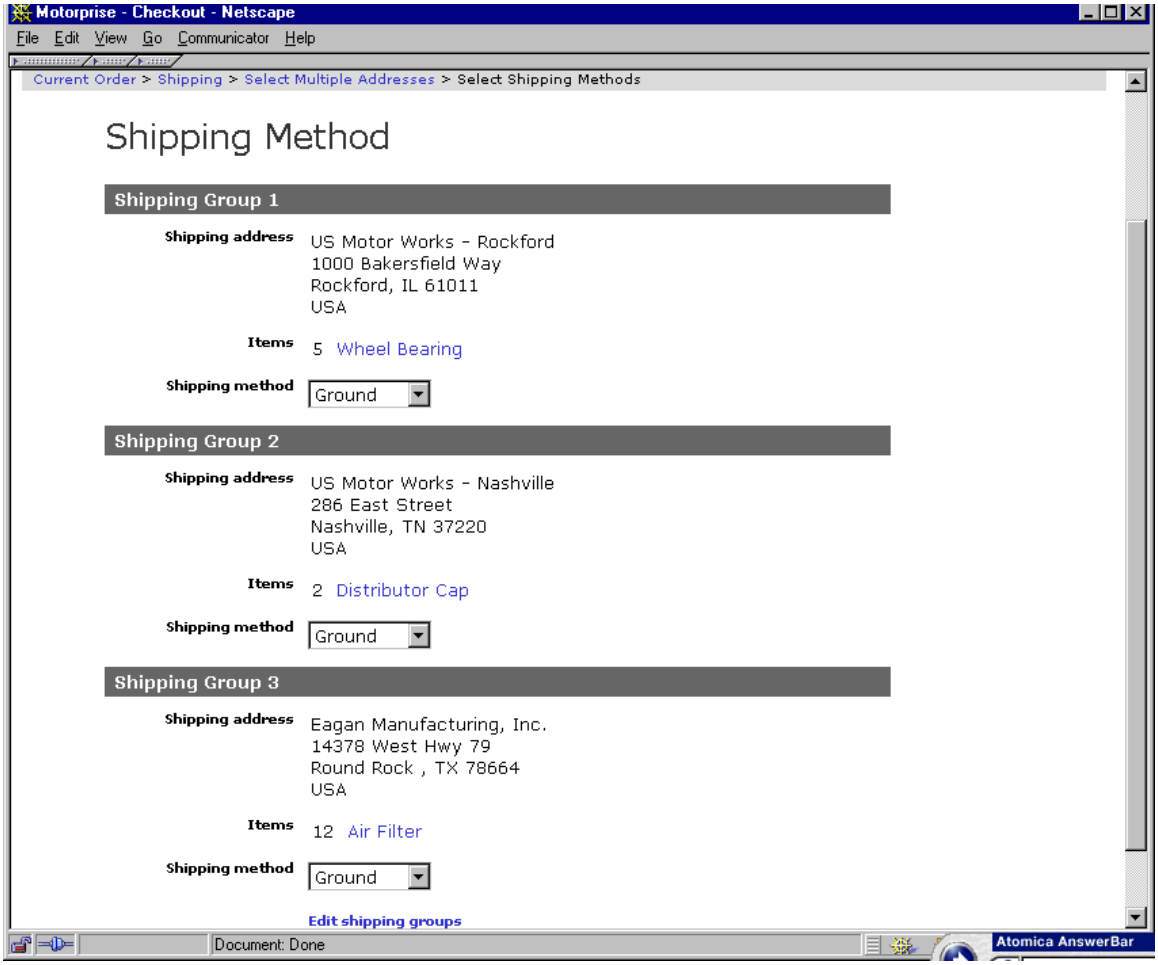
You can have multiple shipping types by including multiple, comma-delimited entries in the `shippingGroupTypes` value, such as `value="hardgoodShippingGroup,electronicShippingGroup"`.

The user is presented with a form to specify a shipping address for each item in the order; the default address is selected automatically. Any changes made by the user essentially modify the `CommerceItemShippingInfo` objects for each `CommerceItem` object.

A `CommerceItemShippingInfo` object holds the information that relates a `ShippingGroup` in the `ShippingGroupMapContainer` to a `CommerceItem` by amount. If a `CommerceItem` is going to be associated with more than one `ShippingGroup`, then a `CommerceItemShippingInfo` object must describe each of these associations. The `CommerceItemShippingInfoContainer` maintains these associations structurally as a list of `CommerceItemShippingInfo` objects keyed by the `CommerceItem`.

We created a form by iterating over the `CommerceItems` in the order. For each `CommerceItem`, we iterate over its list of `CommerceItemShippingInfo` objects. Initially each `CommerceItemShippingInfo` object refers to a `CommerceItem`, its quantity, and the `ShippingGroup` specified by the `DefaultShippingGroupName` of the `ShippingGroupMapContainer`.

The form permits the user to update the `SplitQuantity` and `SplitShippingGroupName` property values of these `CommerceItemShippingInfo` objects, and to submit these by calling the `ShippingGroupFormHandler.splitShippingInfos`. Thus, `CommerceItems` can be associated with additional or different `ShippingGroups` than those provided by the `ShippingGroupDroplet` initialization.



Multiple shipping groups

Here is the JSP from `checkout/ship_to_multiple.jsp` that creates this form:

```
<%--
    For each CommerceItem in the Order, we obtain a List of
    CommerceItemShippingInfo objects.
    Each CommerceItemShippingInfo object associates the CommerceItem to a
    particular ShippingGroup.
--%>

<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">
    <dsp:setvalue paramvalue="element" param="cisiItem"/>
    <dsp:form action="ship_to_multiple.jsp" method="post">
    <tr valign=top>
     <td><nobr><dsp:valueof param=
       "commerceItem.auxiliaryData.catalogRef.manufacturer_part_number"
        /></nobr></td>
     <td></td>
     <td><dsp:a href="../catalog/product.jsp?navAction=jump">
           <dsp:param name="id" param="commerceItem.auxiliaryData.productId"/>
           <dsp:valueof param=
            "commerceItem.auxiliaryData.catalogRef.displayName"/></dsp:a></td>
     <td></td>

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

    <%--
              These form elements permit the user to assign ShippingGroups by
              name and for a specific quantity to a CommerceItem.
    --%>

     <dsp:input bean=
       "ShippingGroupFormHandler.currentList[param:index].splitQuantity"
         paramvalue="element.quantity" size="4" type="text"/></td>
     <td> </td>
     <td>
       <dsp:select bean=
         "ShippingGroupFormHandler.currentList[param:index].
         splitShippingGroupName">
       <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="SGName" idtype="String"
               param="cisiItem.shippingGroupName">
             <dsp:getvalueof id="keyname" idtype="String" param="key">
             <dsp:oparam name="<%=SGName%>">
               <dsp:option selected="<%=true%>" value=
                 "<%=keyname%>"/><dsp:valueof param="key"/>
             </dsp:oparam>
             <dsp:oparam name="default">
               <dsp:option selected="<%=false%>" value=
                 "<%=keyname%>"/><dsp:valueof param="key"/>
             </dsp:oparam>
```

```
                    </dsp:getvalueof>
                    </dsp:getvalueof>

                  </dsp:droplet>

                  <%-- equivalent code to switch droplet using core:case. Both of
                    these work correctly.
                  <dsp:getvalueof id="keyname" idtype="String" param="key">
                  <dsp:getvalueof id="SGName" idtype="String"
                    param="cisiItem.shippingGroupName">
                    <core:switch value="<%=keyname%>">
                      <core:case value="<%=SGName %>">
                        <dsp:option selected="<%=true%>" value=
                          "<%=keyname%>"/><dsp:valueof param="key"/>
                      </core:case>


                      <core:defaultCase>
                        <dsp:option selected="<%=false%>" value=
                          "<%=keyname%>"/><dsp:valueof param="key"/>
                      </core:defaultCase>
                    </core:switch>
                  </dsp:getvalueof>
                  </dsp:getvalueof>
                            --%>

                </dsp:oparam>
              </dsp:droplet>
              </dsp:select>
            </td>
            <td></td>
            <td>
              <%--Split the CommerceItemShippingInfos and redirect right back here
                with init=false.
              --%>

              <dsp:input bean="ShippingGroupFormHandler.splitShippingInfosSuccessURL"
                type="hidden" value="ship_to_multiple.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>
        </dsp:oparam>
      </dsp:droplet>
    </dsp:oparam>
  </dsp:droplet>
```

We use a complex set of form elements here. The `ShippingGroupDroplet.CommerceItemShippingInfoContainer.CommerceItemShippingInfoMap` bean property is a map of `CommerceItemShippingInfo Lists`.

The value of the request parameter `commerceItem.id` is the dynamic key used to obtain the `CommerceItemShippingInfo List` that corresponds to the current `CommerceItem` in the outer `ForEach` iteration. In order to obtain a reference to a particular `CommerceItemShippingInfo` list, we set the `ShippingGroupFormHandler.listId` property with the list's key. The list is then exposed via the `ShippingGroupFormHandler.currentList` property. We then iterate over the list and expose the `splitQuantity` and `splitShippingGroupName` properties of each item, as referenced in a list by the `index` param.

When the user is satisfied with the `ShippingGroup` to `CommerceItem` associations, he or she clicks the **Continue** button to proceed with the purchase process. This invokes the `ShippingGroupFormHandler.applyShippingGroups` handler. The handler collects the information in each `CommerceItemShippingInfo` object and adds the appropriate `ShippingGroups` to the order, along with their necessary relationships to each `CommerceItem` . For more information on the `ShippingGroupFormHandler`, see the *ATG Commerce Programming Guide*. The JSP to accomplish this is as follows:

```
<%--
    Invoke the applyShippingGroups handler and redirect to shipping_method.jsp
    upon success.
--%>

<dsp:form action="ship_to_multiple.jsp" method="post">
<dsp:input bean="ShippingGroupFormHandler.applyShippingGroupsSuccessURL"
   type="hidden" value="shipping_method.jsp"/>
<dsp:input bean="ShippingGroupFormHandler.applyShippingGroups" type="submit"
   value="Continue"/>
</dsp:form>
```

The user is then redirected to `checkout/shipping_method.jsp`, where he or she may select a shipping method for each `ShippingGroup`.

Available Shipping Methods

Motorprise uses the `atg/commerce/pricing/AvailableShippingMethods` servlet bean to provide the user with a list of available shipping methods.

The `AvailableShippingMethods` servlet bean queries the `ShippingPricingEngine` to determine all the available shipping methods for a particular `ShippingGroup`. The `ShippingPricingEngine` iterates over its registered shipping calculators and includes the returned shipping method of each one. Motorprise ships with three registered `ShippingPricingCalculators` that correspond to Ground, TwoDay and NextDay shipping methods. When the user is finished, the form's action redirects the user to the next portion of the purchase process, which is specification of payment information.

Payment Information

Motorprise users can use one or more payment methods to pay for each order. Each payment method is associated with a billing address. Payment methods can be applied to the entire order, or to a portion of the order such as a line item, the shipping cost, or tax. Valid payment methods for Motorprise include:

- Credit card (a P-card or corporate purchasing card)
- Invoice request

Users only see the payment methods that they are authorized to use. If users are authorized to request invoices, they will be able to enter as many PO or requisition numbers as they want. If users are authorized to use credit cards, they will be able to use those credit cards in their lists of authorized credit cards. In Motorprise, users cannot enter new credit cards; they must contact their company administrator to request a new payment method. You could set different restrictions on payment methods depending on your business model.

`Checkout/billing.jsp` first uses the `AuthorizedPaymentTypesDroplet` servlet bean to determine which `PaymentGroup` types the user can use. The `AuthorizedPaymentTypesDroplet` distinguishes between those users who are authorized to use credit cards, invoices, both, or neither. Subsequently, the servlet bean embeds a particular page fragment that includes the proper payment forms for that `PaymentGroup` type:

```
<dsp:droplet name="AuthorizedPaymentTypesDroplet">
      <dsp:param bean="Profile" name="profile"/>
      <dsp:oparam name="output">
        <dsp:droplet name="Switch">
          <dsp:param name="value" param="potentialPaymentTypes.creditCard"/>
          <dsp:oparam name="true">
            <dsp:droplet name="Switch">
              <dsp:param name="value"
                param="potentialPaymentTypes.invoiceRequest"/>
              <dsp:oparam name="true">
                <dsp:droplet name="IsEmpty">
                  <dsp:param bean="Profile.paymentTypes" name="value"/>
                  <dsp:oparam name="true">
                    <dsp:include page="billing_invoice.jsp" flush="true">
                      <dsp:param name="init" value="true"/></dsp:include>
                  </dsp:oparam>
                  <dsp:oparam name="false">
                    <dsp:include page="billing_invoice_cc.jsp" flush="true">
                      <dsp:param name="init" value="true"/></dsp:include>
                  </dsp:oparam>
                </dsp:droplet>
              </dsp:oparam>
              <dsp:oparam name="false">
                <dsp:droplet name="IsEmpty">
                  <dsp:param bean="Profile.paymentTypes" name="value"/>
                  <dsp:oparam name="true">
```

```
                <dsp:include page="no_billing.jsp" flush="true"></dsp:include>
              </dsp:oparam>
              <dsp:oparam name="false">
                <dsp:include page="billing_cc.jsp" flush="true"><dsp:param
                  name="init" value="true"/></dsp:include>
              </dsp:oparam>
            </dsp:droplet>
          </dsp:oparam>
        </dsp:droplet>
      </dsp:oparam>
      <dsp:oparam name="false">
        <dsp:droplet name="Switch">
          <dsp:param name="value" param=
            "potentialPaymentTypes.invoiceRequest"/>
          <dsp:oparam name="true">
            <dsp:include page="billing_invoice.jsp" flush="true"><dsp:param
              name="init" value="true"/></dsp:include>
          </dsp:oparam>
          <dsp:oparam name="false">
            <dsp:include page="no_billing.jsp" flush="true"></dsp:include>
          </dsp:oparam>
        </dsp:droplet>
      </dsp:oparam>
    </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>
```

The billing portion of the purchase process relies on `PaymentGroup` objects to encapsulate payment information. The associations between each `PaymentGroup` and the order are maintained in `CommerceIdentifierPaymentInfo` objects. The various billing page fragments use the `PaymentGroupDroplet` to create both the user's `PaymentGroups` and `CommerceIdentifierPaymentInfos`. These are added to the user's `PaymentGroupMapContainer` and `CommerceIdentifierPaymentInfoContainer`, which are session components that maintain these objects throughout the purchase process.

Credit Cards

If the user is authorized to use credit cards, the `PaymentGroupDroplet` is used on the `checkout/billing_cc.jsp`. A snippet of the code is shown below:

```
<%--
    The PaymentGroupDroplet is used here to initialize the user's CreditCard
    PaymentGroups, and an OrderPaymentInfo object to associate a PaymentGroup with
    the Order.
--%>

<dsp:droplet name="PaymentGroupDroplet">
  <dsp:param name="clear" param="init"/>
```

```
  <dsp:param name="paymentGroupTypes" value="creditCard"/>
  <dsp:param name="initPaymentGroups" param="init"/>
  <dsp:param name="initOrderPayment" param="init"/>
  <dsp:oparam name="output">
```

The PaymentGroupDroplet initialization is context dependent, so this page takes an initialization parameter called init that specifies whether the old PaymentGroupMapContainer and CommerceIdentifierPaymentInfoContainer should be cleared and reinitialized. During initialization the PaymentGroupDroplet creates CreditCard PaymentGroups that correspond to the user's paymentTypes Profile property, and creates an OrderPaymentInfo that represents the payment information for the entire order. Once the user selects a PaymentGroup and hits **Continue**, the PaymentGroupFormHandler.applyPaymentGroups handler is invoked. This executes the moveToConfirmation pipeline chain and redirects the user to the subsequent portion of the purchase process.

Invoices

If the user is authorized to use invoices instead of credit cards, the PaymentGroupDroplet is initialized with different parameters indicating which PaymentGroup types to create. The PaymentGroupDroplet is introduced on checkout/billing_invoice.jsp as follows:

```
<%--
    The PaymentGroupDroplet is used here to initialize an InvoiceRequest
    PaymentGroup, and an OrderPaymentInfo object to associate a PaymentGroup with
    the Order.
--%>
<dsp:droplet name="PaymentGroupDroplet">
  <dsp:param name="clear" param="init"/>
  <dsp:param name="paymentGroupTypes" value="invoiceRequest"/>
  <dsp:param name="initPaymentGroups" param="init"/>
  <dsp:param name="initOrderPayment" param="init"/>
  <dsp:oparam name="output">
```

Invoices and Credit Cards

A user might be authorized to use both invoices and credit cards, as shown on checkout/billing_invoice_cc.jsp:

```
<%--
    The PaymentGroupDroplet is used here to initialize the user's CreditCard
    PaymentGroups and an InvoiceRequest PaymentGroup, as well as an
    OrderPaymentInfo object to associate a PaymentGroup with the Order.
--%>

<dsp:droplet name="PaymentGroupDroplet">
  <dsp:param name="clear" param="init"/>
```

```
  <dsp:param name="paymentGroupTypes" value="creditCard,invoiceRequest"/>
  <dsp:param name="initPaymentGroups" param="init"/>
  <dsp:param name="initOrderPayment" param="init"/>
  <dsp:oparam name="output">
```

Repricing Orders

In a checkout process, a user can make changes, such as to the cart or the shipping information, that can affect the pricing of the order. ATG Commerce provides several methods for repricing an order. For example, the `CartModifierFormHandler` class automatically reprices an order after any change to the shopping cart. However, other purchase process form handlers do not automatically reprice the order, so it must be repriced explicitly.

In the Motorprise checkout process, users submit shipping information before billing information. They can change information, such as address or shipping method, that may affect the shipping price, and therefore the total order price. The `ShippingGroupFormHandler` used on the shipping pages does not automatically reprice the order. Thus, the order must be repriced explicitly on the billing pages to ensure that the price reflects the new shipping information.

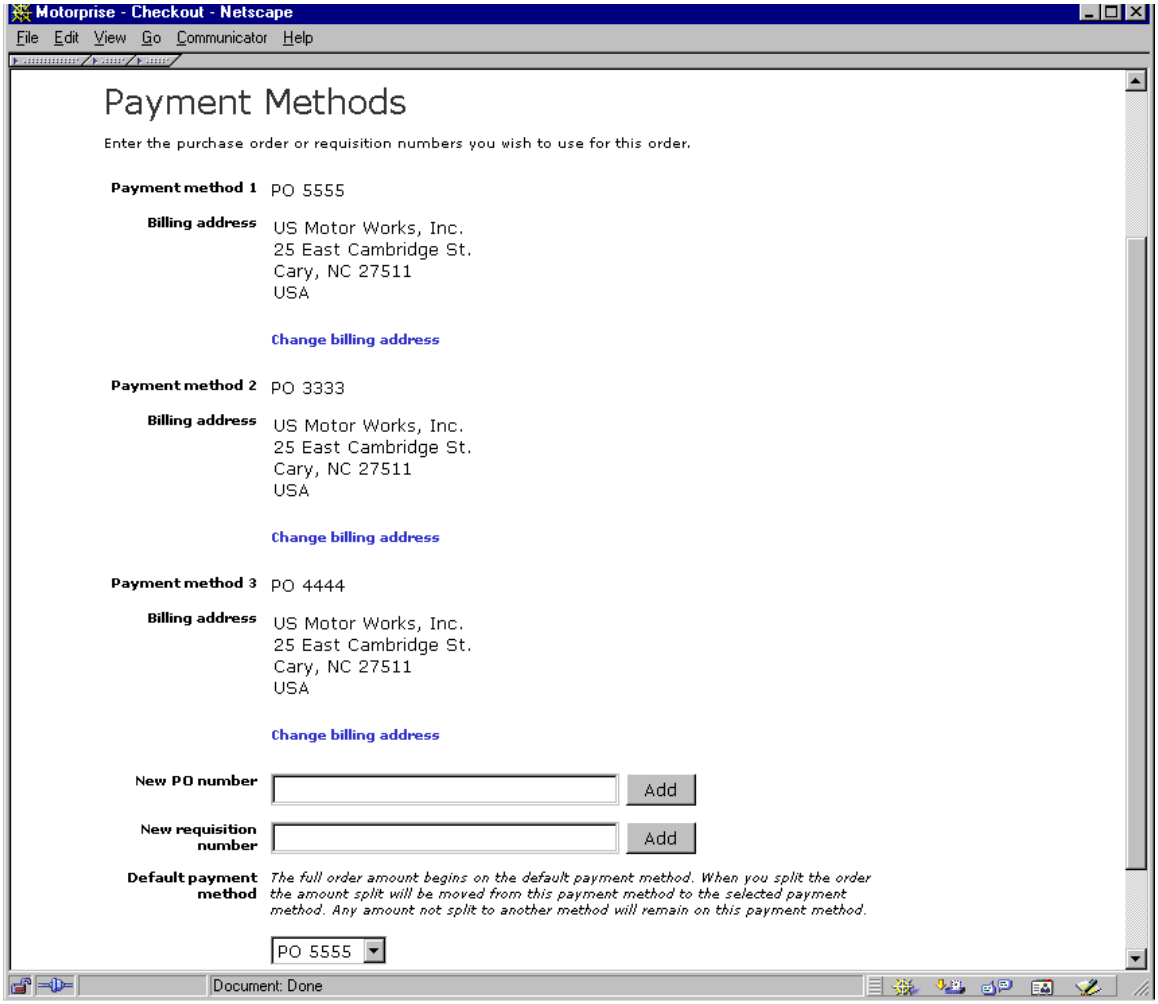
We used the `RepriceOrderDroplet` at the top of the billing pages `billing_invoice.jsp`, `billing_invoice_cc.jsp`, and `billing_cc.jsp`, to reprice the order as follows:

```
<dsp:importbean bean="/atg/commerce/order/purchase/RepriceOrderDroplet"/>

<%--Reprice the Order total so that we can assign PaymentGroups to any
CommerceIdentifier.
--%>
<dsp:droplet name="RepriceOrderDroplet">
  <dsp:param name="pricingOp" value="ORDER_TOTAL"/>
</dsp:droplet>
```

Selecting Multiple Payment Groups

Motorprise provides users the option to use multiple payment methods for processing their order. This feature is implemented in Motorprise beginning with `checkout/payment_methods.jsp`, which is linked to from `billing.jsp`. This page is used to gather a list of the user's possible payment methods, before they are actually applied to specific portions of the order.

This feature again relies on the type of `PaymentGroups` that the user is authorized to use. If the user is authorized to use invoices, then she may enter multiple PO or requisition numbers. If the user is authorized to use credit cards, then she may use all those found in her Profile. Users first select a default `PaymentGroup` before proceeding to the split payment pages. All the order costs are initially applied to this default `PaymentGroup`. Users can then split portions of that cost into different `PaymentGroups`. We used `checkout/SplitPaymentOrderDetails.jsp` to let users split the `PaymentGroups` that will be applied to the entire order.



Adding multiple payment methods and selecting the default payment method.

The `PaymentGroupDroplet` is used to provide the required helper objects and, based on the user's authorization, to include the user's credit cards in the `PaymentGroupMapContainer`. Any `InvoiceRequest` objects created on previous `payment_methods.jsp` are already in the session-scoped container. The following JSP on `SplitPaymentOrderDetails.jsp` ensures this initialization:

```
<%--
    The PaymentGroupDroplet initializes an OrderPaymentInfo object based on
    the value of the request parameter "init". CreditCard PaymentGroups are also
    initialized if the user is authorized to use them.
--%>
<dsp:droplet name="PaymentGroupDroplet">
   <dsp:param name="initOrderPayment" param="init"/>
   <dsp:param name="paymentGroupTypes" value="creditCard"/>
```

```
   <dsp:param name="initPaymentGroups" param="potentialPaymentTypes.creditCard"/>
   <dsp:oparam name="output">
```

Once initialization is complete, we present the user with a form to edit the payment information for the order. The `OrderPaymentInfo` helper objects hold the information that associates `PaymentGroups` in the `PaymentGroupMapContainer` to the `Order` by amount.

A similar page is `checkout/SplitPaymentDetails.jsp`. It works the same way as `SplitPaymentOrderDetails.jsp`, except that it enables the user to split the `PaymentGroups` across line items, shipping costs, and tax. The `PaymentGroupDroplet` creates a `CommerceItemPaymentInfo` object for each `CommerceItem` in the order, a `ShippingGroupPaymentInfo` object for each `ShippingGroup` in the `Order`, and a `TaxPaymentInfo` for the tax. The following JSP ensures this initialization:

```
<%--The PaymentGroupDroplet initializes the CommerceIdentifierPaymentInfo objects
    based on the value of the request parameter "init". CreditCard PaymentGroups
    are also initialized if the user is authorized to use them.
--%>

<dsp:droplet name="PaymentGroupDroplet">
   <dsp:param name="initItemPayment" param="init"/>
   <dsp:param name="initShippingPayment" param="init"/>
   <dsp:param name="initTaxPayment" param="init"/>
   <dsp:param name="paymentGroupTypes" value="creditCard"/>
   <dsp:param name="initPaymentGroups" param="potentialPaymentTypes.creditCard"/>
   <dsp:oparam name="output">
```

We create a form by iterating over the `OrderPaymentInfo` objects for the order. Initially, the `PaymentGroupDroplet` creates one `OrderPaymentInfo` object that associates the entire `Order` cost to the default `PaymentGroup`.

Each time the user splits a portion of the order using a different payment, a new `OrderPaymentInfo` object is added to the `CommerceIdentifierPaymentInfoContainer`.

Splitting the order among multiple payment methods by dollar amount.

The form permits the user to update the `SplitAmount` and `SplitPaymentMethod` property values of these objects, and to submit them by calling `PaymentGroupFormHandler.splitPaymentInfo`. This property associates the order costs with additional or different `PaymentGroups` than those provided by the `PaymentGroupDroplet` initialization.

We present the user with a form to edit all of these `CommerceIdentifierPaymentInfo` objects, which appear as line items that include a payment type and a payment amount. We first retrieve the `CommerceItems`, `ShippingGroups`, and the `Order` itself. For each of these we obtain the corresponding list of `CommerceItemPaymentInfo` objects. We iterate over each list and provide inputs to edit the items' `SplitAmount` and `SplitPaymentMethod` property values.

As is the case with complex `ShippingGroups`, we encounter a complex set of form elements here. The `PaymentGroupDroplet.CommerceIdentifierPaymentInfoContainer.CommerceIdentifierPaymentInfoMap` bean property is a `Map` of `CommerceIdentifierPaymentInfo Lists`. The value of the request parameter `commerceItem.id` is the dynamic key used to obtain the `CommerceIdentifierPaymentInfo List` that corresponds to the current `CommerceItem` in the outer `ForEach` iteration. In order to obtain a reference to a particular `CommerceIdentifierPaymentInfo` list, we set the `PaymentGroupFormHandler.listId` property with the list's key. The list is then exposed via the `PaymentGroupFormHandler.currentList` property. We then iterate over the list and expose the

splitAmount and splitPaymentMethod properties of each item, as referenced in a list by the index param.

Once the user is satisfied with the payment method associations, she may proceed with the purchase process by clicking **Continue**, which invokes the PaymentGroupFormHandler.applyPaymentGroups handler. The handler collects the information in the CommerceIdentifierPaymentInfo objects and adds the appropriate PaymentGroups to the Order, along with the necessary relationships to the CommerceItems in the order. For more information on the PaymentGroupFormHandler, see the *ATG Commerce Programming Guide*. The JSP to accomplish this is as follows:

```
<dsp:form action="split_payment.jsp" method="post">
<dsp:input bean="PaymentGroupFormHandler.applyPaymentGroupsSuccessURL" type=
   "hidden" value="IsEmptyCostCenters.jsp?link=split_payment.jsp"/>
<dsp:input bean="PaymentGroupFormHandler.applyPaymentGroups" type="submit"
   value="Continue"/>
</dsp:form>
```

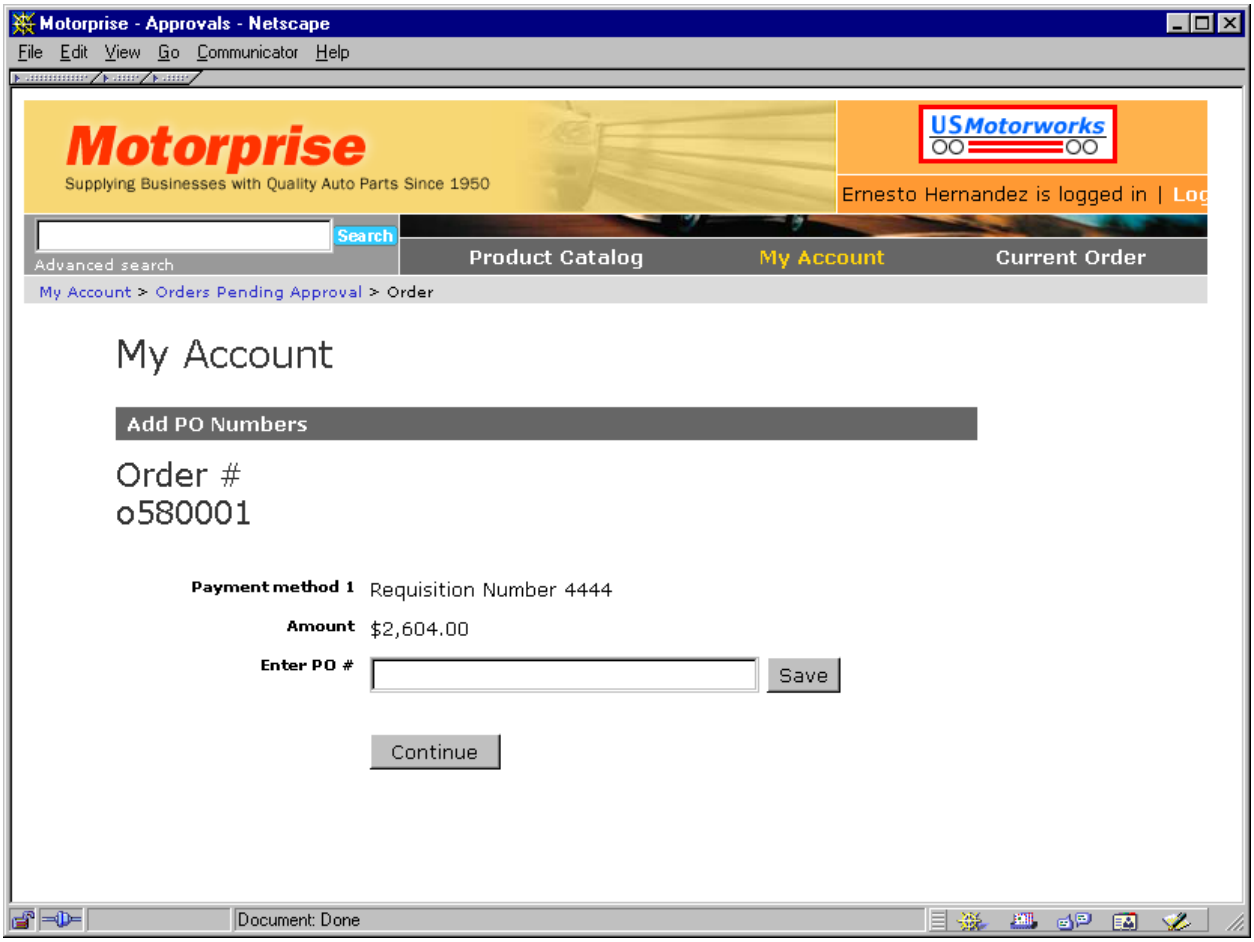
Requisitions

In Motorprise, when a user checks out and requests an invoice, she can specify either a PO number *or* a requisition number. If she chooses the latter, a PO number is specified later when the order is approved. In Motorprise, orders using requisition numbers automatically require approval unless the requireApproval property in the user's profile is set to false. You can set different approval conditions depending on your business model.

In the B2B commerce platform, the buyer must always specify a purchase order number when paying by invoice (even if they are using a requisition number). The Motorprise store overrides this behavior and lets the buyer specify a purchase order number, a requisition number, or both. If the buyer does not specify a purchase order number when creating the order, the approver must specify one when approving the order.

Another pipeline processor was added to the checkRequiresApproval chain in the approval pipeline manager. It iterates over all payment groups in the order, looking for requisition numbers. If it finds one, it adds an error to the pipeline result object to indicate to the approval system that approval is required.

When an approver receives an order with a requisition number but no purchase order number, he is prompted to provide the P.O. number on user/add_po_number.jsp.



An approver must provide a purchase order for orders without them.

Specifying Cost Centers

Users can also specify certain cost centers to be billed internally for items in their orders. Cost centers are defined at the business unit level and are inherited by users.

After the payment portion of the checkout process, the user may be asked to associate cost centers with the order. This determination is made at `checkout/IsEmptyCostCenters.jsp`. If the user does not have cost centers, he or she is redirected to `checkout/confirmation.jsp`. If the user has cost centers, then the `Redirect` droplet proceeds to `checkout/cost_centers.jsp`.

If the user does have cost centers, then she will be able to assign them either to the entire order, or to portions of the order. The list of cost centers that the user may choose from includes a default cost center and an alternate list of authorized cost centers. In the simple case, the user can assign all items in the

order to the default cost center, which is selected by default in `checkout/cost_centers.jsp`, or can choose another cost center from the select box.

`checkout/cost_centers.jsp` uses `CostCenterDroplet` to create appropriate cost centers and `CommerceIdentifierCostCenters` for the given user. It is used as follows in `checkout/cost_centers.jsp` to display all the authorized cost centers in a select box and assign a cost center to the order:

```
<%--
    CostCenterDroplet initializes cost centers available to user, and also
    CostCenters objects corresponding to the order to associate cost centers to
    the order
--%>

<dsp:droplet name="CostCenterDroplet">
<dsp:param name="clear" value="true"/>
<dsp:param name="initCostCenters" value="true"/>
<dsp:param name="initItemCostCenters" value="false"/>
<dsp:param name="initShippingCostCenters" value="false"/>
<dsp:param name="initTaxCostCenters" value="false"/>
<dsp:param name="initOrderCostCenters" value="true"/>
<dsp:param name="useAmount" value="false"/>
<dsp:oparam name="output">

  <tr valign=top>
    <td></td>
        <td align=right><span class=smallb>Cost center</span></td>
    <td>
      <dsp:setvalue bean="CostCenterFormHandler.listId" paramvalue="order.id"/>
      <dsp:input bean="CostCenterFormHandler.listId" paramvalue="order.id"
        priority="<%=(int) 9%>" type="hidden"/>
        <%-- we only expect this to have 1 element at [0], but we put this in a
          ForEach to be safe --%>
      <dsp:droplet name="ForEach">
        <dsp:param bean="CostCenterFormHandler.currentList" name="array"/>
        <dsp:oparam name="output">
          <dsp:select bean=
            "CostCenterFormHandler.currentList[param:index].CostCenterName"><br>

            <%--
                 List all the cost centers available to the user in select
                 box, so he can choose one among them.
            --%>
          <dsp:droplet name="ForEach">
            <dsp:param bean="Profile.costCenters" name="array"/>
            <dsp:param name="elementName" value="CostCenter"/>
            <dsp:oparam name="output">
              <dsp:droplet name="Switch">
                <dsp:param name="value" param="CostCenter.repositoryId"/>
```

```
                <dsp:getvalueof id="costCenterId" idtype="String"
                  param="CostCenter.identifier">
      <dsp:getvalueof id="defaultCostCenter" idtype="String"
        bean="Profile.defaultCostCenter.repositoryId">
                  <dsp:oparam name="<%=defaultCostCenter%>">
                    <dsp:option selected="<%=true%>" value="<%=costCenterId%>"/>
                      <dsp:valueof param="CostCenter.identifier"/> -
                      <dsp:valueof param="CostCenter.description"/>
                  </dsp:oparam>
                  <dsp:oparam name="default">
                    <dsp:option selected="<%=false%>" value="<%=costCenterId%>"/>
                      <dsp:valueof param="CostCenter.identifier"/>
                      <dsp:valueof param="CostCenter.description"/>
                  </dsp:oparam>
                </dsp:getvalueof>
      </dsp:getvalueof>

         </dsp:droplet> <%-- Switch --%>
    </dsp:oparam><%-- End: ForEach.oparam --%>
    </dsp:droplet> <%-- End: ForEach --%>
         </dsp:select>
       </dsp:oparam>
     </dsp:droplet>
  </td></tr>
    </dsp:oparam>
    </dsp:droplet> <%-- End: CostCenterDroplet --%>
```

The user may assign any cost center to the order and can decide to proceed to the next step in the purchase process, confirmation, by submitting the form.

Using Multiple Cost Centers

Splitting cost centers among items in an order is implemented with `cost_centers_line_item.jsp`, which is a link from `cost_centers.jsp`.

This page utilizes a request parameter `init` to enable or disable initialization of the `CostCenterDroplet`. The `CostCenterDroplet` is used on `costCentersLineItemDetails.jsp` to create cost centers for all the items, shipping groups, and tax in the order. Like `PaymentGroupDroplet`, `CostCenterDroplet` is context dependent. This droplet takes an `init` parameter, which specifies whether the old `CostCenterMapContainer` and `CommerceIdentiferCostCenterMap` should be reinitialized. The `CostCenterDroplet` is introduced on `costCentersLineItemDetails.jsp` as follows:

```
<%--
     CostCenterDroplet initializes the CommerceIdentifierCostCenter objects based
     on the requested init parameter and also creates costcenter objects.
--%>

<dsp:droplet name="CostCenterDroplet">
```

```
  <dsp:param name="clear" param="init"/>
  <dsp:param name="initCostCenters" value="true"/>
  <dsp:param name="initItemCostCenters" param="init"/>
  <dsp:param name="initShippingCostCenters" param="init"/>
  <dsp:param name="initTaxCostCenters" param="init"/>
  <dsp:param name="useAmount" value="false"/>
  <dsp:oparam name="output">
```

Once the initialization is complete, we present the user with a form to edit the cost centers, which modifies the `CommerceIdentifierCostCenter` objects. We first gather the `CommerceItems`, `ShippingGroups`, and the tax groups from the order. For each of these we obtain the corresponding list of `CommerceIdentiferCostCenter` objects and iterate over each list.

We provide dropdown lists so that the user can edit the `splitCostCenterName` and `splitQuantity` properties of `CommerceIdentiferCostCenter` objects that correspond to commerce items in the order and the items' `splitCostCenterName` for shipping groups and tax.

The following code illustrates the form to modify `CommerceIdentifierCostCenter` objects for the `CommerceItems` in the order:

```
<%--
    Iterate through the commerceitems of the order and assign that particular
    commerceIdentifierCostCenter to the CostCenterFormHandler to edit the
    costcenter information. Since we provide <dsp:form></dsp:form> elements across
    each item we will be editing only one item at any given time.
    If user splits any item across cost centers, we create extra
    CommerceIdentifierCostCenter to accomodate the split qty.
--%>

<dsp:droplet name="ForEach">
  <dsp:param name="array" param="order.commerceItems"/>
  <dsp:oparam name="output">
    <%--
      Set the current item id to CostCenterFormHandler
      --%>
    <dsp:setvalue paramvalue="element" param="commerceItem"/>
    <dsp:setvalue bean="CostCenterFormHandler.listId"
      paramvalue="commerceItem.id"/>
        <dsp:droplet name="ForEach">
          <dsp:param bean="CostCenterFormHandler.currentList" name="array"/>
          <dsp:oparam name="output">
            <%-- begin line item --%>

            <tr valign=top>
              <dsp:form action="cost_centers_line_item.jsp" method="post">
              <td><dsp:valueof param="commerceItem.catalogRefId"/></td>
              <td></td>
              <td><dsp:a href="../catalog/product.jsp?navAction=jump">
```

```
        <dsp:param name="id" param=
          "commerceItem.auxiliaryData.productId"/>
        <dsp:valueof param=
          "commerceItem.auxiliaryData.productRef.displayName"/>
          </dsp:a></td>
<td></td>


<td align=middle><dsp:valueof param="element.quantity"/></td>
<td> </td>


<td>
<dsp:input bean=
  "CostCenterFormHandler.currentList[param:index].splitQuantity"
  paramvalue="element.quantity" size="4" type="text"/></td>
<td> </td>


<td align=center>
    <%--
      Set the cost center to be used for the current item
      --%>
  <dsp:getvalueof id="itemCenterName" idtype="String" param=
    "element.costCenterName">
  <dsp:select bean="CostCenterFormHandler.
    currentList[param:index].splitCostCenterName">
    <%--
      Iterate through the available cost centers so that user can
      choose one among them to assign for the current item
      --%>
  <dsp:droplet name="ForEach">
    <dsp:param name="array" param="costCenters"/>
    <dsp:oparam name="output">
      <dsp:droplet name="Switch">
        <dsp:param name="value" param="key"/>
        <dsp:getvalueof id="keyname" idtype="String" param="key">
       <%-- <dsp:oparam name="param:...element.costCenterName"> --%>
          <dsp:oparam name="<%=itemCenterName%>">
          <dsp:option selected="<%=true%>" value="<%=keyname%>"/>
            <dsp:valueof param="key"/>
          <dsp:droplet name="ForEach">
           <dsp:param bean="Profile.costCenters" name="array"/>
           <dsp:param name="elementName" value="costCenter"/>
           <dsp:oparam name="output">
            <dsp:droplet name="Switch">
             <dsp:param name="value" param="costCenter.identifier"/>
             <dsp:oparam name="<%=keyname%>">
              <dsp:valueof param="costCenter.description"/>
             </dsp:oparam>
```

```
                        </dsp:droplet>
                       </dsp:oparam>
                      </dsp:droplet>
                    </dsp:oparam>
                    <dsp:oparam name="default">
                      <dsp:option selected="<%=false%>" value="<%=keyname%>"/>
                        <dsp:valueof param="key"/>
                      <dsp:droplet name="ForEach">
                       <dsp:param bean="Profile.costCenters" name="array"/>
                       <dsp:param name="elementName" value="costCenter"/>
                       <dsp:oparam name="output">
                        <dsp:droplet name="Switch">
                         <dsp:param name="value" param="costCenter.identifier"/>
                         <dsp:oparam name="<%=keyname%>">
                          <dsp:valueof param="costCenter.description"/>
                         </dsp:oparam>
                        </dsp:droplet>
                       </dsp:oparam>
                      </dsp:droplet>
                    </dsp:oparam>
                    </dsp:getvalueof>
                  </dsp:droplet>
                </dsp:oparam>
              </dsp:droplet>
              </dsp:select>

              </dsp:getvalueof>
            </td>
            <td align=center>
              <dsp:input bean="CostCenterFormHandler.listId" paramvalue=
                "commerceItem.id" priority="<%=(int) 9%>" type="hidden"/>
              <dsp:input bean="CostCenterFormHandler.splitCostCentersSuccessURL"
                type="hidden" value="cost_centers_line_item.jsp?init=false"/>
              <dsp:input bean="CostCenterFormHandler.splitCostCentersErrorURL"
                type="hidden" value="cost_centers_line_item.jsp?init=false"/>
              <dsp:input bean="CostCenterFormHandler.splitCostCenters"
                type="submit" value="Save"/>
            </td>
          </tr>
          </dsp:form>
          <%-- end line item --%>
        </dsp:oparam>
      </dsp:droplet>
  </dsp:oparam>
</dsp:droplet>
```

Upon submission, the form's action redirects the user to the next portion of purchase process, which begins at `checkout/confirmation.jsp`.

Confirming and Committing Orders

The order confirmation page, `checkout/confirmation.jsp` displays the entire order, including the shipping and payment information and a final total. The user can:

- place the order at that time
- cancel the order
- schedule the order to be placed at some time in the future

When a user submits the order on confirmation screen, we check to see if there are any approval conditions for the order. If the order requires approval, we place the order in an approval pending state. For more information on approvals, please refer to My Account chapter.

If the order does not require approval, it is submitted for processing.

The confirmation page, `checkout/confirmation.jsp`, relies on the `CommitOrderFormHandler` and `CancelOrderFormHandler` respectively to place or cancel the order:

```
<dsp:input bean="CommitOrderFormHandler.commitOrder" type="submit" value="Place
  order now"/>
<dsp:input bean="CommitOrderFormHandler.commitOrderSuccessURL" type="hidden"
  value="../checkout/thank_you.jsp"/>
<dsp:input bean="CommitOrderFormHandler.commitOrderErrorURL" type="hidden"
  value="../checkout/confirmation.jsp"/>
<dsp:input bean="CancelOrderFormHandler.cancelOrder" type="submit" value="Cancel
  order"/>
<dsp:input bean="CancelOrderFormHandler.orderIdToCancel" beanvalue=
  "ShoppingCart.current.id" type="hidden"/>
```

For information on scheduled orders, see the Scheduled Orders section in the My Account chapter.

The order commit page summarizes the order, assigns it an ID and thanks the user.

Backing up Components for Checkout

Because Motorprise is a demo, for performance reasons we did not back up certain components. Shipping and payment information is temporarily stored in memory until the user commits to the order. Thus, if Dynamo stops running for some reason, this information is lost and the user must re-enter it.

You can make a simple configuration change to have this information restored by a backup server. Add the following lines to `/atg/dynamo/Configuration.properties`:

```
sessionBackupServerPropertyList+=\
   /atg/commerce/order/purchase/ShippingGroupContainerService.shippingGroupMap,\
```

```
/atg/commerce/order/purchase/ShippingGroupContainerService.
    commerceItemShippingInfoMap,\
/atg/commerce/order/purchase/ShippingGroupContainerService.
    defaultShippingGroupName,\
/atg/commerce/order/purchase/PaymentGroupContainerService.paymentGroupMap,\
/atg/commerce/order/purchase/PaymentGroupContainerService.
    commerceIdentifierPaymentInfoMap,\
/atg/commerce/order/purchase/PaymentGroupContainerService.
    defaultPaymentGroupName
```

10 SOAP Support

Like most business-to-business Web sites, Motorprise needs to integrate with many other systems, from fulfillment houses to the business systems of its partners. We used the Motorprise example in the ATG Business Commerce Reference Application to demonstrate one possible way of achieving this integration. Motorprise includes a simple example of the exchange of XML documents via Simple Object Access Protocol (SOAP). We elected to use SOAP as the protocol for the communication layer because it is XML-based and thus supports integration with disparate systems. This chapter contains the following sections:

What is SOAP?

Simple Object Access Protocol (SOAP) is a protocol that allows communication between various systems. It describes a standard way to create messages that can be sent from one system to another. SOAP defines the format of a message so that there is a standard place for various parts of the message such as the envelope and the body. Establishing standards such as the location and characteristics of these message parts allows quicker integration since all the systems involved in the communication use the same protocol. In a SOAP communication, in addition to a standard message format, the message body itself is an XML document. This allows for the integration of disparate systems, since any system that can process text (XML) can understand the protocol.

SOAP was born out of a need for XML RPC (Remote Procedure Call). There are two different parts to any SOAP interaction: the client and the server.

The SOAP client:

- generates a SOAP envelope.
- places the payload, or content, into it.
- designates a particular service on the destination server to which the message should be delivered.

- sends the message over HTTP to the destination server.

The server:

- maintains a registry that maps service names to actual locations of services.
- receives messages and examines them to determine to which service they should be delivered.
- looks in its mapping table for the service name.
- delivers the message's content to the associated method in the mapped service.

Using SOAP in Motorprise

The integration needs of most B2B sites center on communicating order information from one system to another. For example, a business commerce site may need to submit orders to a fulfillment house, to a financial system for accounting purposes, or to a buy-side application such as Ariba PunchOut. For the Motorprise store, we created a scenario, SendOrderViaSOAP, to demonstrate this integration.

SendOrderViaSOAP listens for orders to be submitted. When they are, the order is serialized to an XML document and is then sent out over a SOAP request to a SOAP server. The SOAP server then writes the contents of the SOAP message to the Dynamo console.

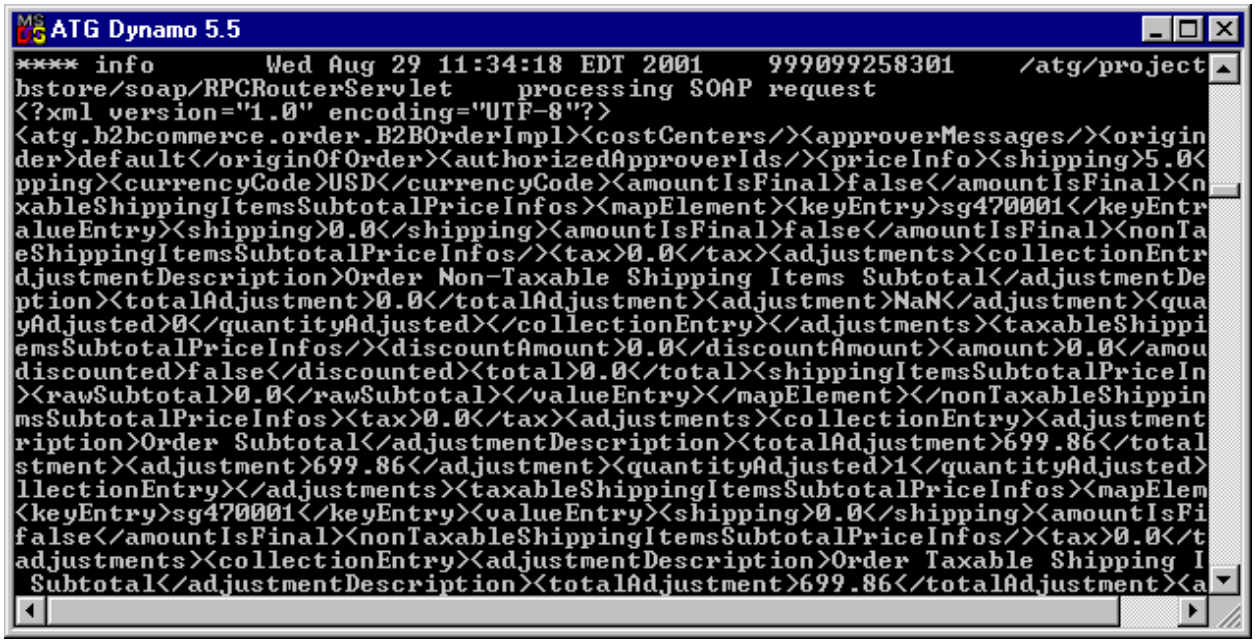
Demonstrating SOAP Functionality in Motorprise

To see an example of how we used SOAP in Motorprise, you must enable the SOAP scenario, SendOrderViaSOAP, and place an order.

By default, SendOrderViaSOAP is disabled to avoid printing XML to the console every time an order is placed in the store. To enable the scenario, do the following:

1. In the ATG Control Center, go to Scenarios > Scenarios.
2. Select the B2B folder.
3. Right-click on the SendOrderViaSOAP scenario and choose "Enable Scenario."

The scenario is now enabled. To observe orders being sent out over SOAP, place an order in the store. (Be sure to place an order that does not require approval. For example, any order placed by Stuart will not require approval.) The scenario listens for any order being submitted, automatically serializes it to XML, and sends it out. You will see the SOAP message printed to the console.



SOAP requests are written to the ATG console.

Client Pieces of SOAP in Motorprise

For Motorprise, we wrote and configured a simple SOAP client that sends a series of parameters as a SOAP message. The class that we created is located at `atg.projects.b2bstore.soap.SimpleSOAPClient`. This class allows the easy configuration of a SOAP client and its commonly changed values.

`<ATG10dir>/MotorpriseJSP/j2ee-apps/motorprise/config/atg/projects/b2bstore/soap/SimpleSOAPClient.properties`:

```
$class=atg.projects.b2bstore.soap.SimpleSOAPClient

SOAPServerURL=http\://hostname\:8080/Dynamo/solutions/B2BStore/soap
targetObjectURI=urn\:motorprise-display-xml
methodName=receiveDocument
```

In the properties file, the URL that the SOAP message is sent to is defined as `http://hostname:8080/Dynamo/solutions/B2BStore/soap`. The server listens for SOAP messages at this URL.

The `targetObjectURI=urn\:motorprise-display-xml` property indicates the service to which to route the message.

Finally, the `methodName=receiveDocument` property indicates which method on the configured service should be invoked. In this case, it is `receiveDocument` method, which simply displays an XML document to the Dynamo console.

The SendOrdersViaSOAP scenario has a custom action associated with it: `atg.projects.b2bstore.soap.SendObjectAsXML`. This is the part of the scenario that sends the order object as an XML document.

To implement this custom action in Motorprise, we modified the configuration file that controls the scenario server, `/atg/scenario/scenarioManager.xml`. We added the following lines to the file:

```
<!-- Action to send an object over a SOAP request -->
<!-- Used in Motorprise to demonstrate sending of -->
<!-- orders as an XML document.                   -->
<action>
  <action-name>
    Send Object As XML via SOAP
  </action-name>
  <action-class>
    atg.projects.b2bstore.soap.SendObjectAsXML
  </action-class>
  <resource-bundle>
    atg.projects.b2bstore.scenario.UserResources
  </resource-bundle>
  <display-name-resource>
    sendObjectAsXML.displayName
  </display-name-resource>
  <description-resource>
    sendObjectAsXML.description
  </description-resource>
  <action-execution-policy>
    collective
  </action-execution-policy>
  <action-error-response>
    delete
  </action-error-response>

  <!-- Parameter that indicates the object that will -->
  <!-- be marshalled to XML.                         -->
  <action-parameter>
    <action-parameter-name>
      marshalObject
    </action-parameter-name>
    <display-name-resource>
      sendObjectAsXML.marshalObject.displayName
    </display-name-resource>
    <action-parameter-class>
      java.lang.Object
    </action-parameter-class>
```

```
    <required>
      true
    </required>
    <description-resource>
      sendObjectAsXML.marshalObject.description
    </description-resource>
  </action-parameter>
  <!-- This is the key that will be passed to the -->
  <!-- to the ObjectMarshallerDispatcher service  -->
  <!-- The key is used to determine which         -->
  <!-- marshaller to use.                         -->
  <action-parameter>
    <action-parameter-name>
      marshalKey
    </action-parameter-name>
    <display-name-resource>
      sendObjectAsXML.marshalKey.displayName
    </display-name-resource>
    <action-parameter-class>
      java.lang.String
    </action-parameter-class>
    <required>
      false
    </required>
    <description-resource>
      sendObjectAsXML.marshalKey.description
    </description-resource>
  </action-parameter>
</action>
```

This code registers an action with the Scenario Server that will be displayed in the ACC as "Send Object As XML via SOAP." This action takes two parameters, which are specified when creating the scenario in the ACC.

The first parameter, which is required, is `marshalObject`. This parameter is the object that is to be serialized to XML. Typically, this object comes from the source event.

The second parameter, which is optional, is `marshalKey`. This is a key object, provided by the user, that is passed to the `ObjectMarshallerDispatcher` to determine which `ObjectMarshaller` processor will perform the marshalling.

For more information, see the *Using the Configuration Reporter* section in the *Monitoring Site Performance* chapter of the *ATG Installation and Configuration Guide*.

In addition to configuring `marshalObject` and `marshalKey`, you can also modify the behavior of the `SendObjectAsXML` action. Most of its properties are configurable via the `atg.projects.b2bstore.soap.SOAPResources` file.

You can find the following resource strings in the `SOAPResources` file:

- `SimpleSOAPClient` determines the Nucleus location of the `SimpleSOAPClient` to use when sending messages via SOAP. Its default is `/atg/projects/b2bstore/soap/SimpleSOAPClient`.
- `MarshalServicePath` determines the Nucleus location of the `ObjectMarshallerDispatcherService` to use to marshal objects into XML documents. Its default is `/atg/dynamo/service/xml/ObjectMarshallerDispatcher`.
- `SOAPParameterName` determines the name of the parameter that is embedded in the SOAP payload. Its default is `xmlDocument`.

For more information on creating custom actions for scenarios, see the *Adding Custom Events and Actions to Scenarios* section in the *ATG Personalization Programming Guide*.

Server Pieces of SOAP in Motorprise

There are two pieces to the SOAP server in the store: the server itself and the SOAP service that gets registered with it.

SOAP Server

The SOAP server in the store listens for incoming SOAP messages and routes them to a particular destination. The server (sometimes referred to as a SOAP router) is really a Java Servlet that routes messages to a particular SOAP service.

First, we created a Nucleus component to act as the SOAP server: `/atg/projects/b2bstore/soap/RPCRouterServlet`. The properties file for this component looks like:

```
$class=atg.server.soap.DynamoRPCRouterServlet
$scope=global
serviceManager=ServiceManager
RPCRouter=RPCRouter
```

The `RPCRouterServlet` component requires a configured `ServiceManager` and an `RPCRouter` component.

The `ServiceManager` component maintains a list of all the exposed services that are available to a particular `SOAPRouter`. The single service that gets registered with this `ServiceManager` is the `DOMWriterService`, which is documented in the next section.

This is the properties file for `/atg/projects/b2bstore/soap/ServiceManager`:

```
$class=atg.apache.soap.server.ServiceManager
$scope=global
deployedServicesFile={atg.dynamo.server.home}/data/motorprise/
D6DeployedServices.ds
```

We created an RPCRouter component, located at /atg/projects/b2bstore/soap/RPCRouter, that is specific to Motorprise. The RPCRouter component routes a particular RPC request to a particular service. This component can handle incoming SOAP RPC requests independent of the method of the request's transport, such as HTTP or e-mail.

The RPCRouterServlet receives a request over HTTP, extracts out the SOAP message, and then forwards it to the RPCRouter component. The RPCRouter component then examines the SOAP message and determines for which service the message is designated.

Finally, we needed to ensure that messages sent to a particular URL would be routed to the RPCRouterServlet. We wanted HTTP requests that are sent to /Dynamo/solutions/B2BStore/soap (as defined in the SOAP client) to be automatically forwarded to RPCRouterServlet. We used the dispatcherServiceMap property of the /atg/dynamo/Configuration component to forward these requests by adding a Configuration.properties file at <ATG10dir>/Motorprise/config/atg/dynamo/ with the following contents:

```
dispatcherServiceMap+=\
     /Dynamo/solutions/B2BStore/soap=/atg/projects/b2bstore/soap/
       RPCRouterServlet
```

Thus, when a SOAP client sends messages to a location of /Dynamo/solutions/B2BStore/soap, the messages are forwarded to the SOAP server.

SOAP Service

The SOAP service has a particular method call invoked on it to handle the payload of a SOAP message. In Motorprise, we wanted a service that would print an XML document to the Dynamo console. We created a class that prints a given XML document: atg.projects.b2bstore.soap.DOMWriterService. We then created an instance of this class, /atg/projects/b2bstore/soap/DOMWriterService, with the following properties:

```
$class=atg.projects.b2bstore.soap.DOMWriterService
$scope=global
RPCRouterServlet=/atg/projects/b2bstore/soap/RPCRouterServlet
methods=receiveDocument
serviceId=urn\:motorprise-display-xml
```

The /atg/projects/b2bstore/soap/DOMWriterService class has a method called doStartService that registers the SOAP service with the SOAP server. When the DOMWriterService component starts up, the doStartService method determines the SOAP server through the DOMWriterService component's configured property RPCRouterServlet. The doStartService method registers the SOAP service with the SOAP server so that the server knows to route requests for service urn:motorprise-display-xml with a method call of receiveDocument to the SOAP service registered at /atg/projects/b2bstore/soap/DOMWriterService.

11 Merchandising

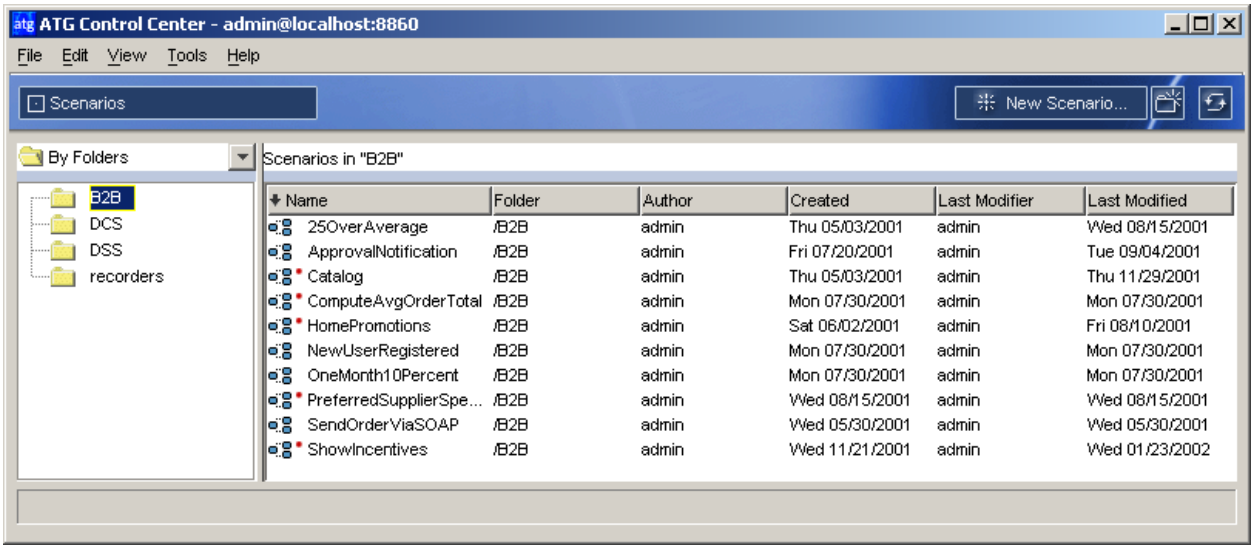
Motorprise uses the scenario and analysis functionality of ATG Business Commerce to provide personalized merchandising to its customers. We created specific scenarios for Motorprise; in addition, Motorprise makes use of core ATG Commerce scenarios. We also created a custom condition in Motorprise. For analysis, Motorprise uses the standard Business Commerce charts.

This chapter contains the following sections:

Motorprise Scenarios

You can view the scenarios we created for Motorprise in the ACC in the B2B folder in the Scenarios > Scenarios section.

Not all Motorprise scenarios are enabled by default (the red dot indicates those that are enabled). Those that send e-mail have been disabled and can be enabled once an e-mail server is set up. See the Setting up e-mail section in the Getting Started chapter of this guide for more information.



Motorprise scenarios in the ACC

25OverAverage

This scenario checks to see if the total cost of an order placed is 25% greater than the average of her previous orders. If it is, the user receives a promotion for 10% of her next order. This scenario relies on values of the Number of Orders and Person's Average order price, which are recorded by the `ComputeAvgOrderTotal` scenario into the buyer's profile. See the description of that scenario below for details.

The first step in this scenario checks when an order is completed. When an order is completed, its sub-type changes to "Order is finished". We check for this rather than for when the order is submitted to avoid giving promotions for orders that are later cancelled. The next step checks whether this buyer has placed five orders to be sure that she has an established average order total.

Next, we check if the order is 25% greater than the buyer's average order total. This is a custom condition written for Motorprise called Order Comparison. It allows you to enter the percentage difference you are looking for. If the order is 25% greater than the average order total, then the buyer receives a promotion for 10% off their next order. He or she also receives an e-mail about the promotion. Because Motorprise has both English and German speaking customers we send the e-mail in the appropriate language by checking their locales before choosing the appropriate e-mail template.

We added a property to the user profile in Motorprise that indicates the number of orders placed, `numOfOrders`, and a property that tracks the average total of all the user's orders, `avgOrderAmt`. Whenever she submits a new order, these two properties are updated with the appropriate values.

We created a new scenario action, Compute Average Order Total, to calculate average order total of user is created. This action takes order total as input and calculates the average order total from `avgOrderAmt` and `numOfOrders`.

Based on this action, a scenario is created with `OrderSumitted` as the event and the amount of the order as input to the action. Also `numOfOrders` is increased by one whenever an order is created. Whenever an order is submitted, this action is invoked with the order amount as input; it then calculates the new average of order total s and stores that value in `avgOrgAmt`.

For more information on creating scenario actions, refer to *ATG Personalization Programming Guide*.

ApprovalNotification

There are three situations when someone should be notified about an order that requires approval:

- When an order requiring approval is submitted, the approver is notified.
- When an approver approves and places an order, the buyer is notified.
- When an approver rejects an order, the buyer is notified.

When an order is submitted that requires approval an Approval Required event is sent and an e-mail is sent to the approver. We first check the locale of the user and then send the appropriate e-mail. When the approver either approves or reject the order an e-mail is sent to the buyer telling him or her what happened to the order.

This is a b2bcommerce scenario that we override in Motorprise in order to add the German language branch.

Catalog

When an anonymous session starts, this scenario assigns them the base catalog. This allows us to give a catalog to known users, which is based on their parent organization's contract with Motorprise.

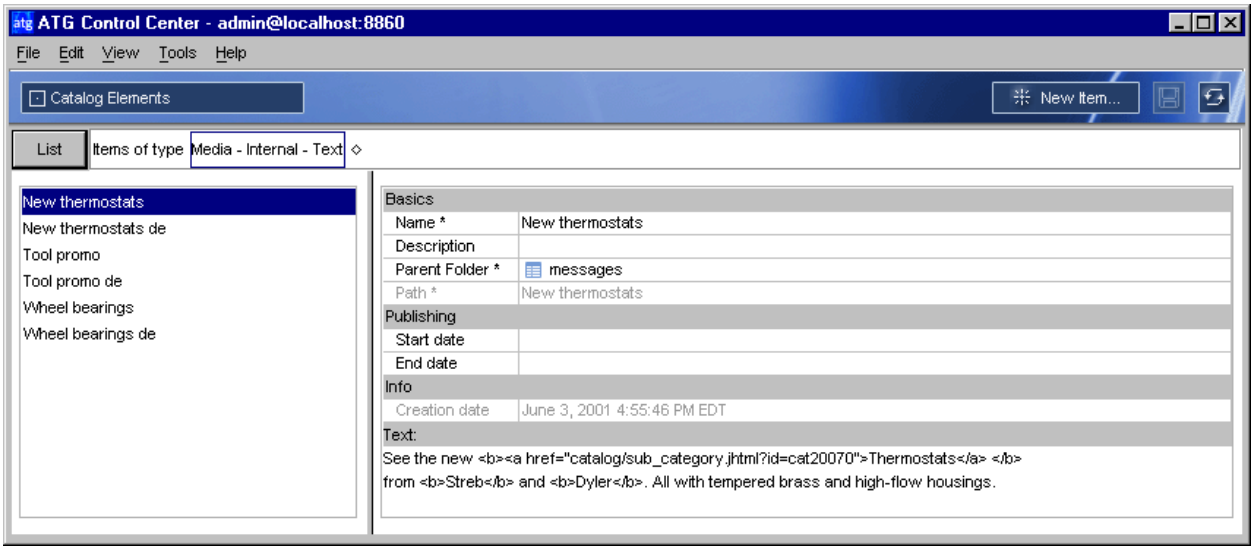
ComputeAvgOrderTotal

This scenario records information that is used in the 25OverAverage scenario. When an order is submitted, it increases the Number of orders of the buyer by one and then computes the buyer's average order total using a custom action.

HomePromotions

When a user logs into the Motorprise home page, certain promotions are displayed to him.

There are three promotions made up of two slots each, a text slot and an image slot, on the home page that are used to deliver text messages and images to users. The text messages are internal messages that are stored in the database. To see the messages go to Catalog Management > Catalog Elements and view items of type `media-internal-text`.



Text media used in the HomePromotions scenario.

The images are the `small images` stored in each product that can be viewed on the corresponding product in the catalog.

For each slot, when the user visits the home page, a text message is added to the text slot and then a corresponding image to the image slot.

NewUserRegistered

When a site administrator registers a new user using `admin/new_user.jsp` or `admin/create_multiple_users.jsp`, an e-mail is sent to the user with her user name and a link to site.

OneMonth10Percent

When a customer submits an order, a scenario instance is created that waits for thirty days and checks whether the user has submitted any orders. If he hasn't, then an e-mail is sent to him notifying him of a 10% off promotion.

PreferredSupplierSpecials

When a user visits the Motorprise home page, certain products from her preferred vendors are displayed in the Preferred Vendor Specials section.

When a buying organization is set up by Motorprise, the customer indicates from which vendors or suppliers the company prefers to buy. This is entered in the organizational profile and so for each buyer that enters the site Motorprise can identify that company's preferred vendors. This allows Motorprise to promote products they know will more likely interest the buyers based on the vendor. These products are shown to the buyer on the home page.

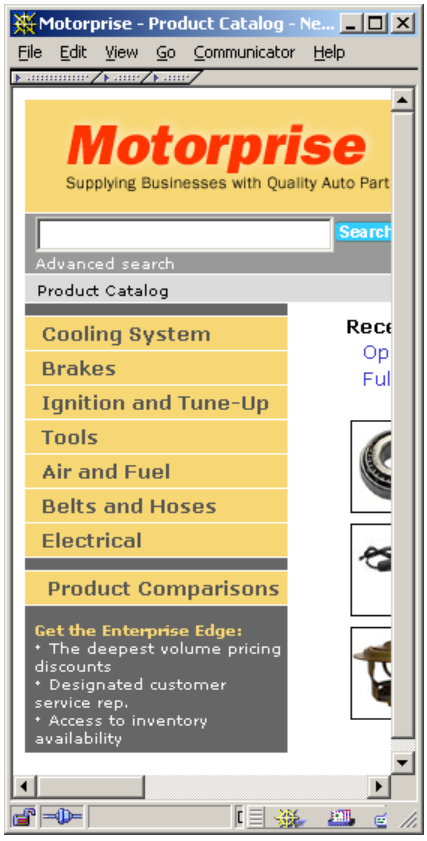
When a buyer visits the home page, either in English or German, we first remove any items that may be in the slot. This is in case a user was browsing the site anonymously before logging in and so might have the default products that are shown to anonymous users in the slot rather than those we intend them to see. Then we check the preferred vendor property and for each vendor show the products we want to promote. At the end of the vendor branches we have an Otherwise branch that shows a default list of products to anonymous users and any users whose parent organization don't have a preferred vendor defined.

SendOrderViaSoap

This scenario sends an order via SOAP so that other systems, such as fulfillment houses or systems of Motorprise partners, can receive and read the order. For more information on this scenario, see the SOAP Support chapter in this guide.

ShowIncentives

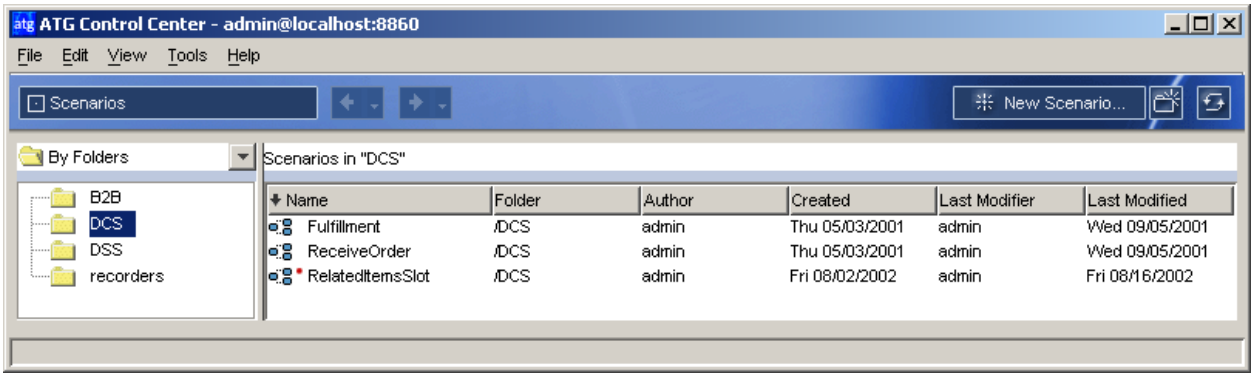
This scenario displays different incentive content under the Catalog Categories, depending on whether or not the user is anonymous, an NDAP user, or a USMW user.



Users from NDAP see an incentive encouraging them to be come an Enterprise customer.

Commerce Scenarios

Motorprise uses the core ATG Commerce scenarios that can viewed in the ACC in the DCS folder under Scenarios > Scenarios.



Motorprise uses core ATG Commerce scenarios.

By default, the RelatedItemsSlot scenario is enabled. You may have enabled the Fulfillment and ReceiveOrder scenarios are enabled to send e-mail. See the *Setting up e-mail* section in the Getting Started chapter of this guide for more information.

Fulfillment

The Fulfillment scenario sends an appropriate email to a customer when any of the following events occur:

- when the order is shipped
- when an order has unavailable items
- when an order is cancelled
- when an item is removed from the order
- when payment information has changed

ReceiveOrder

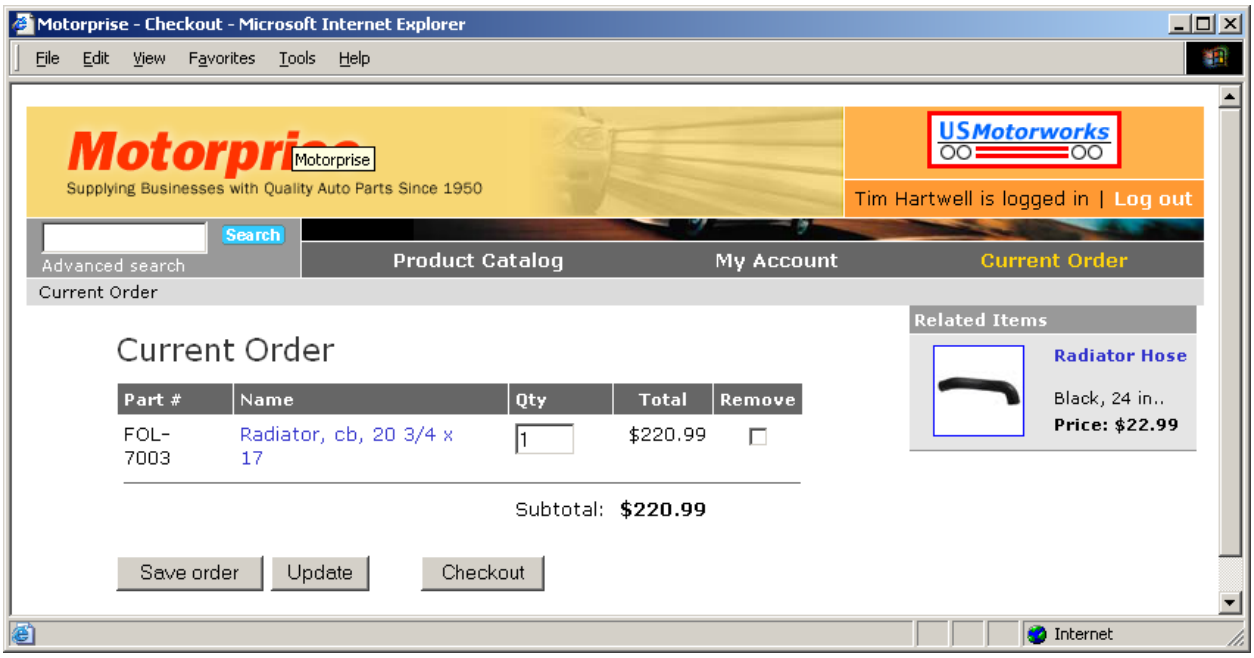
The ReceiveOrder scenario waits for an order to be submitted, and then sends an email to that customer notifying him or her that the order has been received.

For more information on the Fulfillment and ReceiveOrder scenarios, see the *ATG Commerce Programming Guide*.

RelatedItemsSlot

Motorprise uses the RelatedItemsSlot scenario to cross-sell products related to those in the customer's current shopping cart. The scenario determines the contents of a slot named RelatedItemsOfCart, which is used on the shopping cart page, `cart.jsp`, to display items related to the product(s) in the customer's order.

For example, when a customer like Tim Hartwell places a radiator in his order, the slot displays a radiator hose. Each product has a `relatedProducts` property that is used to determine which products to display in the slot on the order page.



The RelatedItemsOfCart slot is used on the shopping cart page.

For more information on the RelatedItemsSlot scenario, see the *Using Scenarios to Cross-Sell and Up-Sell Products* section o f the *Using Commerce Elements in Scenarios* chapter in the *ATG Commerce Guide to Setting Up a Store*.

Creating a Custom Condition in Motorprise

Motorprise offers a special promotion to customers whose current order price is at least 25% greater than their average order price. We implemented a custom condition in DSS to handle this requirement, and we extended the expression grammar in the ACC's scenario editor to provide a customized user interface for creating and editing these conditions.

The *Adding Custom Events, Actions, and Conditions to Scenarios* chapter in the *ATG Personalization Programming Guide* explains how to create custom conditions and expression grammars.

We implemented our custom condition with the class `atg.projects.b2bstore.scenario.PercentageComparisonFilter`, which extends the standard `ExpressionFilter` class from DSS.

`PercentageComparisonFilter` supports conditions of the type:

```
testValue [is at least / is at most]
   percentage percent [greater than / less than / of] baseValue
```

where `testValue` and `baseValue` are numeric expressions and `percentage` is a placeholder for a literal percentage value.

Motorprise, for example, uses this condition class for conditions of the type "Order's price is at least 25% greater than person's average order price". In this case, `testValue` is "order's price", `baseValue` is "person's average order price", and `percentage` is 25.

Our custom condition expects to receive five operands when invoked:

- `Operand 1` is the test value, and must be a numeric value or expression.
- `Operand 2` indicates whether the comparison type is "at least" or "at most." A string constant represents each possible value for this operand.
- `Operand 3` is the literal percentage value in the condition, which must be a number.
- `Operand 4` indicates whether the condition looks for the test value to be some percentage greater than, some percentage less than, or some percentage of the base value. As with the comparison type, a string constant represents each possible value.
- `Operand 5` is the base value, which must also be a numeric value or expression.

We specified the number and types of the operands when we registered our custom condition in `scenarioManager.xml`:

```
<condition-registry>
  <condition>
    <condition-name>orderPricePercentComparison</condition-name>
    <filter-class>atg.projects.b2bstore.scenario.
      PercentageComparisonFilter</filter-class>
    <resource-bundle>atg.projects.b2bstore.scenario.UserResources
    </resource-bundle>
    <display-name-resource>orderPricePercentComparison.displayName
    </display-name-resource>
    <description-resource>orderPricePercentComparison.description
    </description-resource>

<!--
    The action-parameter tags specify positional parameters that will
```

```
    be passed to the filter class's initialize() method.
-->

    <action-parameter>
      <action-parameter-name>
        testValue
      </action-parameter-name>
      <display-name-resource>
        orderPricePercentComparison.testValue.displayName
      </display-name-resource>
      <action-parameter-class>
        java.lang.Number
      </action-parameter-class>
      <required>
        true
      </required>
      <description-resource>
        orderPricePercentComparison.testValue.description
      </description-resource>
    </action-parameter>

    <action-parameter>
      <action-parameter-name>
        comparisonType
      </action-parameter-name>
      <display-name-resource>
        orderPricePercentComparison.comparisonType.displayName
      </display-name-resource>
      <action-parameter-class>
        java.lang.String
      </action-parameter-class>
      <required>
        true
      </required>
      <description-resource>
        orderPricePercentComparison.comparisonType.description
      </description-resource>
    </action-parameter>

    <action-parameter>
      <action-parameter-name>
        percentage
      </action-parameter-name>
      <display-name-resource>
        orderPricePercentComparison.percentage.displayName
      </display-name-resource>
      <action-parameter-class>
        java.lang.Integer
      </action-parameter-class>
      <required>
```

```
        true
      </required>
      <description-resource>
        orderPricePercentComparison.percentage.description
      </description-resource>
    </action-parameter>

    <action-parameter>
      <action-parameter-name>
        percentageType
      </action-parameter-name>
      <display-name-resource>
        orderPricePercentComparison.percentageType.displayName
      </display-name-resource>
      <action-parameter-class>
        java.lang.String
      </action-parameter-class>
      <required>
        true
      </required>
      <description-resource>
        orderPricePercentComparison.percentageType.description
      </description-resource>
    </action-parameter>

    <action-parameter>
      <action-parameter-name>
        baseValue
      </action-parameter-name>
      <display-name-resource>
        orderPricePercentComparison.baseValue.displayName
      </display-name-resource>
      <action-parameter-class>
        java.lang.Number
      </action-parameter-class>
      <required>
        true
      </required>
      <description-resource>
        orderPricePercentComparison.baseValue.description
      </description-resource>
    </action-parameter>

  </condition>
</condition-registry>
```

We also created a custom grammar extension to control how the ACC scenario editor displays and edits instances of this condition. The expression grammar generally follows the pattern described in Extending

the Expression Editor in the *Adding Custom Events, Actions, and Conditions to Scenario* chapter in the *ATG Personalization Programming Guide*, but with two features that require extra explanation.

First, instead of using the standard `<sequence>` tag to begin our condition grammar we used the DCS `<required-order-sequence>` tag, as follows:

```
<required-order-sequence id="condition-orderPricePercentComparison">
```

Using this tag tells the scenario editor that our custom condition can only appear in a scenario segment that already contains a DCS order event.

Otherwise, there would be no way to evaluate an expression like "Order's price" as part of the condition.

Second, we used the custom grammar to constrain the condition's first operand (the test value) to one of four predefined values. Even though the `PercentageComparisonFilter` class accepts any numeric expression as the test value, we wanted to limit the business user to selecting one of the following values:

- Order price
- Order price before discounts
- Order tax
- Order shipping cost

We accomplished this by using a `<choice>` tag as the first operand, where each token within the choice represents one of the four possible values:

```
<choice>
  <token>
    <description>Order price</description>
    <xml-template>
      <event-property>
        <property-name>order</property-name>
        <property-name>priceInfo</property-name>
        <property-name>total</property-name>
      </event-property>
    </xml-template>
  </token>

  <token>
    <description>Order price before discounts</description>
    <xml-template>
      <event-property>
        <property-name>order</property-name>
        <property-name>priceInfo</property-name>
        <property-name>rawSubTotal</property-name>
      </event-property>
    </xml-template>
  </token>
```

```
  <token>
    <description>Order tax</description>
    <xml-template>
      <event-property>
        <property-name>order</property-name>
        <property-name>priceInfo</property-name>
        <property-name>tax</property-name>
      </event-property>
    </xml-template>
  </token>

  <token>
    <description>Order shipping cost</description>
    <xml-template>
      <event-property>
        <property-name>order</property-name>
        <property-name>priceInfo</property-name>
        <property-name>shipping</property-name>
      </event-property>
    </xml-template>
  </token>
</choice>
```

Each token contains the description that appears in the scenario editor and an XML template for the corresponding event property using SDL, the DSS scenario definition language. Each event property definition specifies one property of a DCS order event. For example, the event property definition is the SDL form of the event's "order.priceInfo.total" subproperty:

```
<event-property>
  <property-name>order</property-name>
  <property-name>priceInfo</property-name>
  <property-name>total</property-name>
</event-property>
```

Motorprise Analysis

Motorprise gives ACC users the ability to generate reports at the organizational level with drill down at the cost center and individual level. Motorprise has the following Business Commerce chart templates.

- Dollars spent:
  - By buying organization (alphabetical)
  - By buying organization/by buyer (alphabetical)
  - By buying organization/by part number
  - By buying organization/by cost center

    - By part number
- Orders/invoices:
    - By date (can specify a specific date or date range)
    - By buying organization
    - By buying organization/by buyer (alphabetical)
- Parts Purchased

12 Creating ACC Users and Setting Privileges

Motorprise has several business users who need access to various sections of the site content. We set up roles for these users so that they could use the ACC to perform various administrative tasks. Certain users manage the catalogs and administer price lists. Another user can set up new customer accounts by creating organizations, contracts, etc.

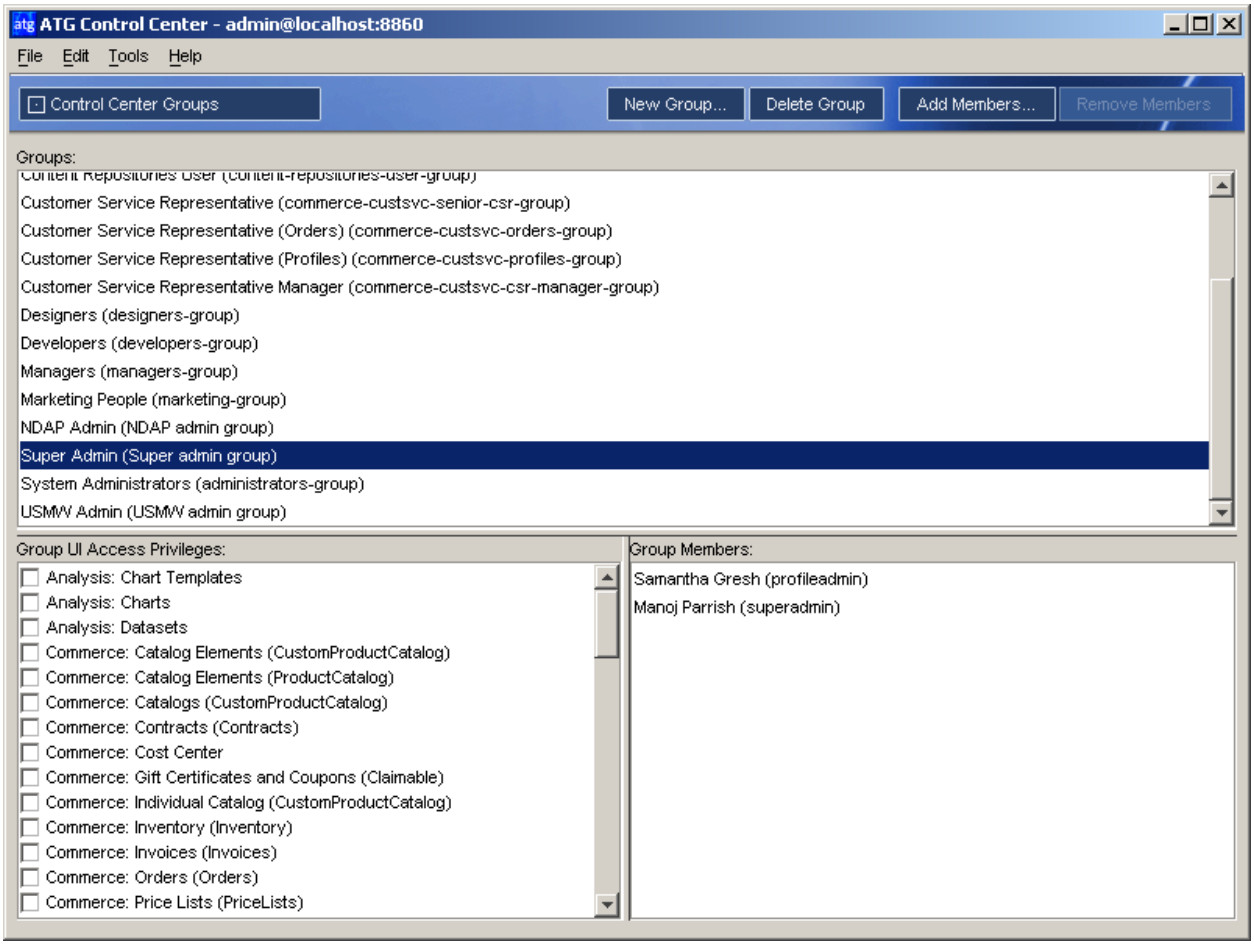
We created these users and defined their access privileges by logging into the ACC as an administrator, using "admin" as the user name and password.

We set up the Motorprise ACC users by:

- Creating five users: Manoj, Eddie, Sam, Manny, and Samantha
- Creating four roles: Superadmin, Content QA, Account Manager for NDAP, Account Manager for USMW, and Profileadmin
- Assigning these roles to specific users
- Setting up access for the various roles

Creating Groups in the ACC

We used the Control Center Groups section in the People and Organizations task menu to create our Motorprise ACC groups. We simply clicked the New Group button and entered a name and description for each group.



Control Center Groups in the People and Organizations section of the ACC.

We created four groups:

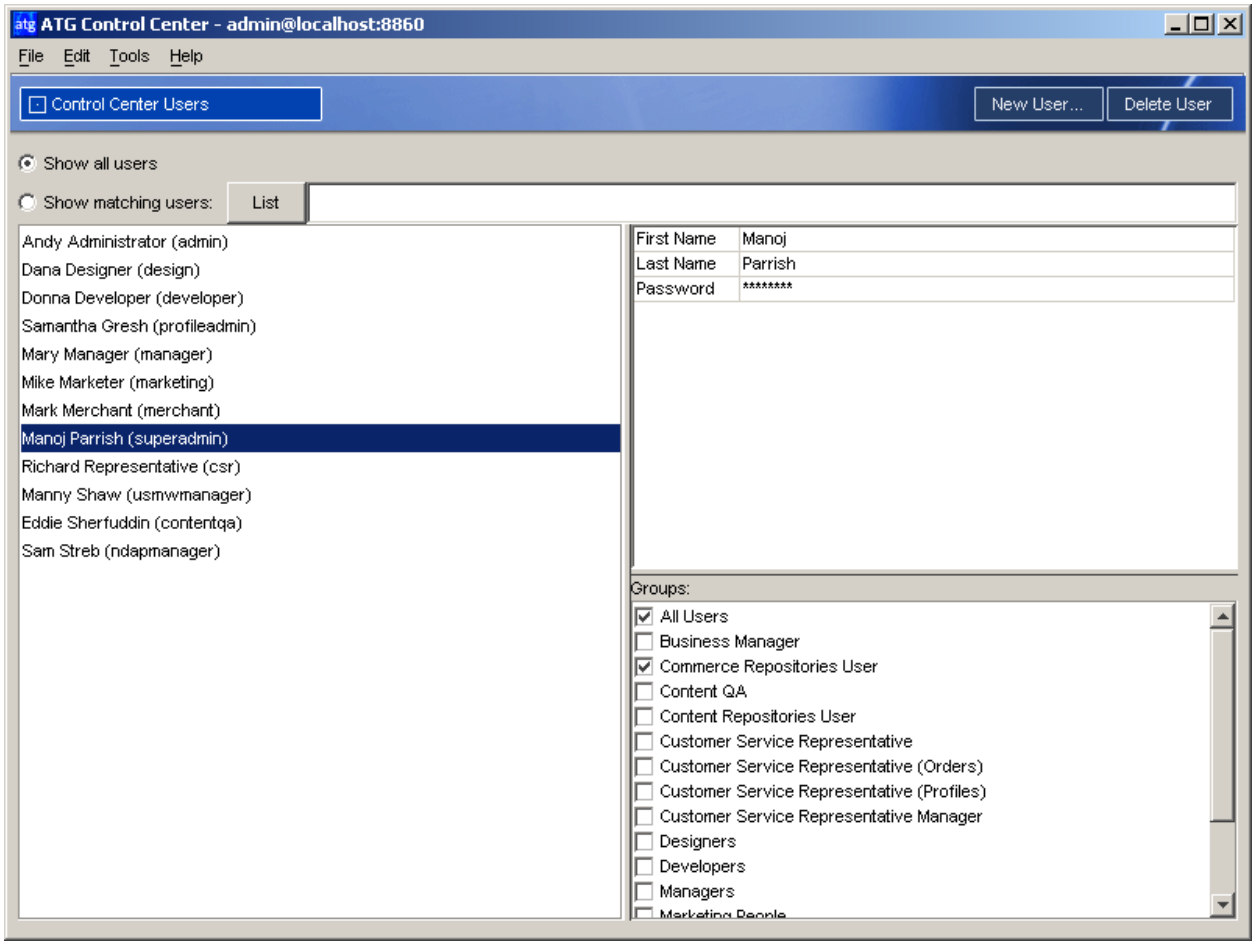
- Content QA
- NDAP Admin
- Super Admin
- USMW Admin

You can also specify certain Group UI Access Privileges for each group. For example, Commerce Repositories Users only have access to the commerce task menus in the ACC.

Creating Users in the ACC

In the People and Organizations task menu > Control Center Users tab, we created our new Motorprise ACC users by clicking the **New User** button and entering the name and password information.

Then, we added each user to groups with the appropriate privileges. For example, we made Manoj Parrish a Super Admin and a Commerce Repositories User. (He is a member of the All Users group by default.)



ACC User Manoj Parrish is assigned to the groups All Users, Commerce Repositories, and Super Admin.

User	Role	Groups
Manoj Parrish	Motorprise Superadmin	All Users Commerce Repositories User Super Admin

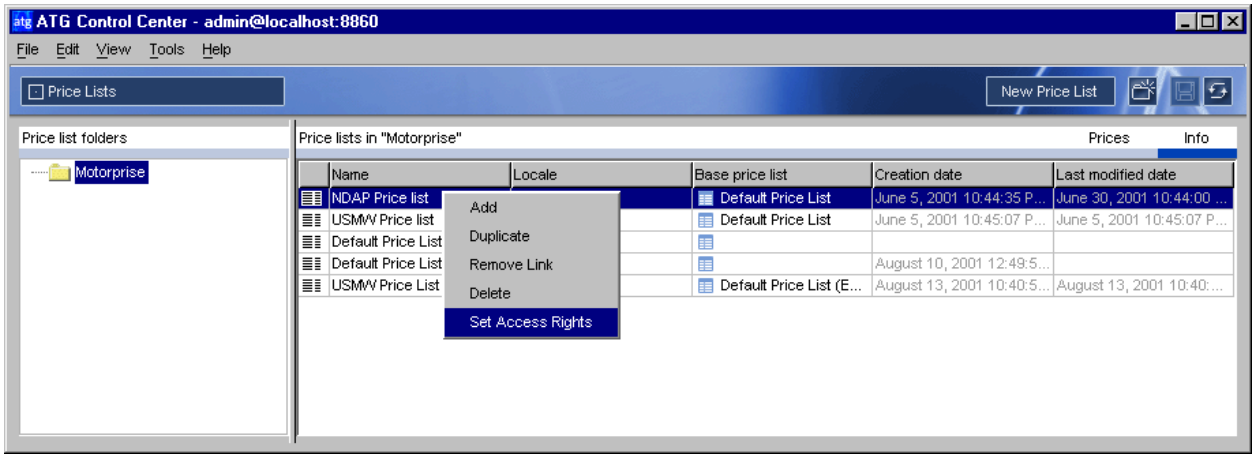
Eddie Sherfuddin	Motorprise Content QA	All Users Commerce Repositories User Content QA
Sam Streb	Motorprise Account Manager for NDAP	All Users Commerce Repositories User NDAP Admin
Manny Shaw	Motorprise Account Manager for USMW	All Users Commerce Repositories User USMW Admin
Samantha Gresh	Motorprise Profileadmin	All Users Commerce Repositories User Content Repositories User Super Admin

Assigning Access Rights to Price Lists

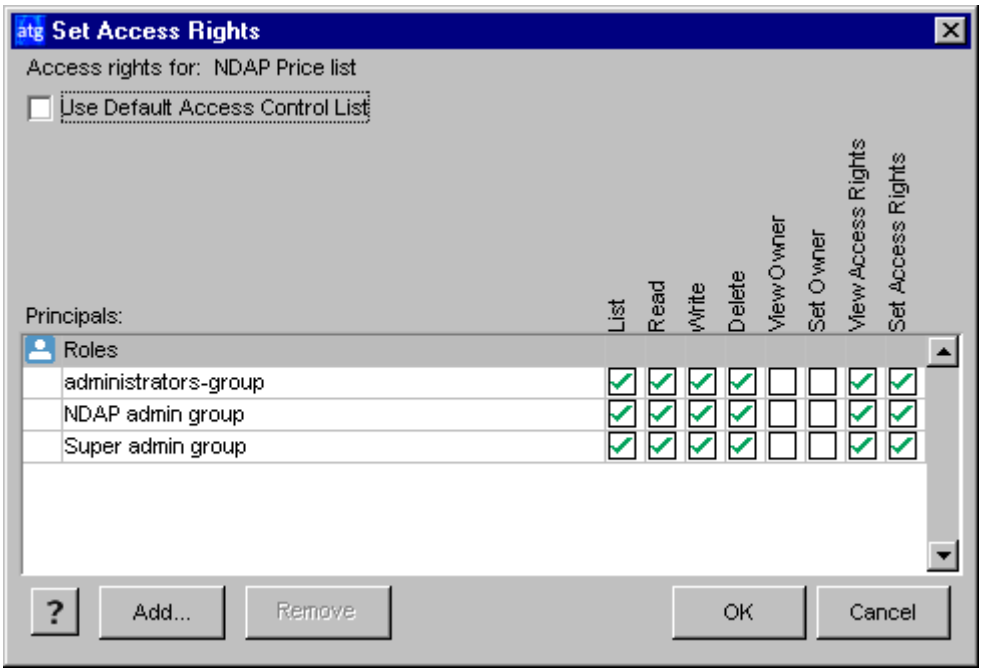
After we created the groups and assigned users to them, we wanted to give certain users access rights to specific price lists. These access rights are inherited by every price and complex price in that particular price list.

For example, to set access rights for the Content QA administrator on the NDAP Price list, you would do the following:

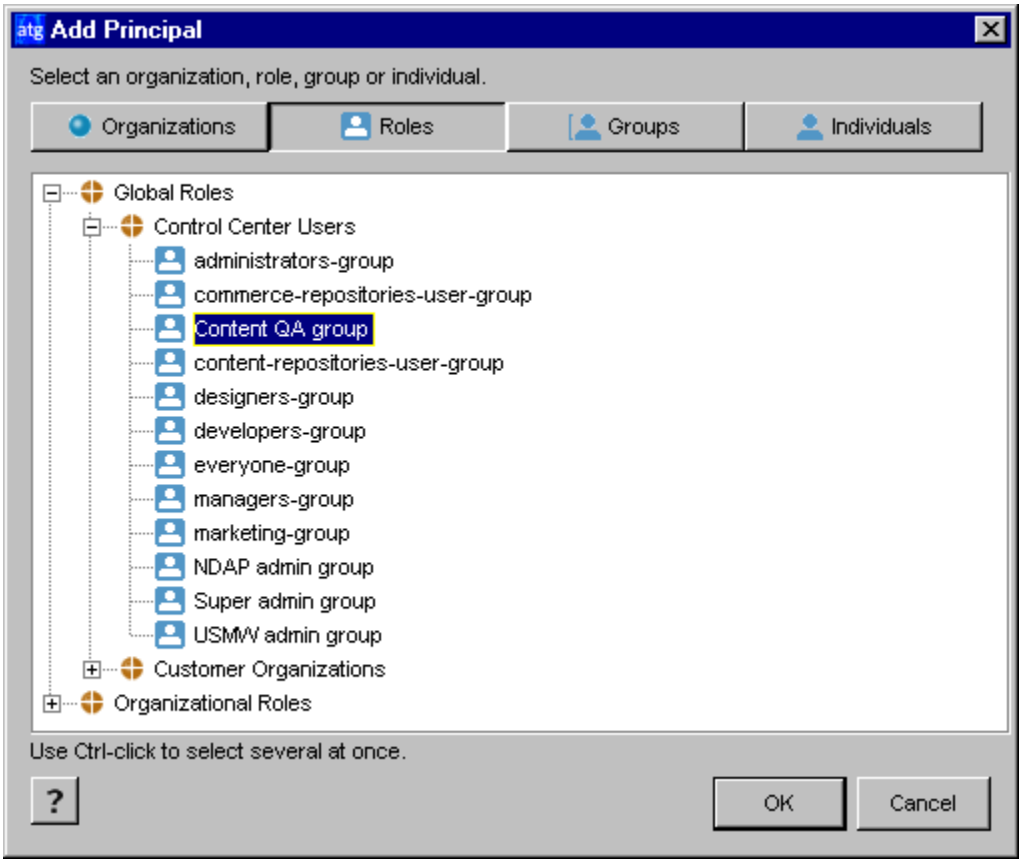
1. In the Pricing > Price Lists task section of the ACC, select the "Info" tab in the top right corner to view the Motorprise price lists: NDAP PriceList, USMW PriceList, USMW PriceList (EUR), the Default Price List and the Default Price List (EUR).



2. Select the NDAP PriceList, right-click on it, and select "Set Access Rights."



3. In the Set Access Rights dialog box, click the **Add** button.



4. Select the Roles tab.
5. Expand the Global Roles and then the Control Center Users to show all the previously created Control Center Groups.
6. Select the Content QA group and click OK. This adds Content QA to the Access Rights dialog box.
7. In the columns that list access rights, select the "List" and "Read" boxes.

For more information on access rights, see the *ATG Personalization Programming Guide*.

Assigning Access Rights to Catalogs

In addition to securing price lists, we also created secure catalogs in Motorprise. By securing catalogs, we allowed only certain users to view and edit some catalogs. For example, the NDAP Account Manager has read and write access to the NDAP catalog, but no access rights to the USMW catalog.

See the next chapter, Exploring Motorprise as a Selling Organization, to understand how each user's access rights work.

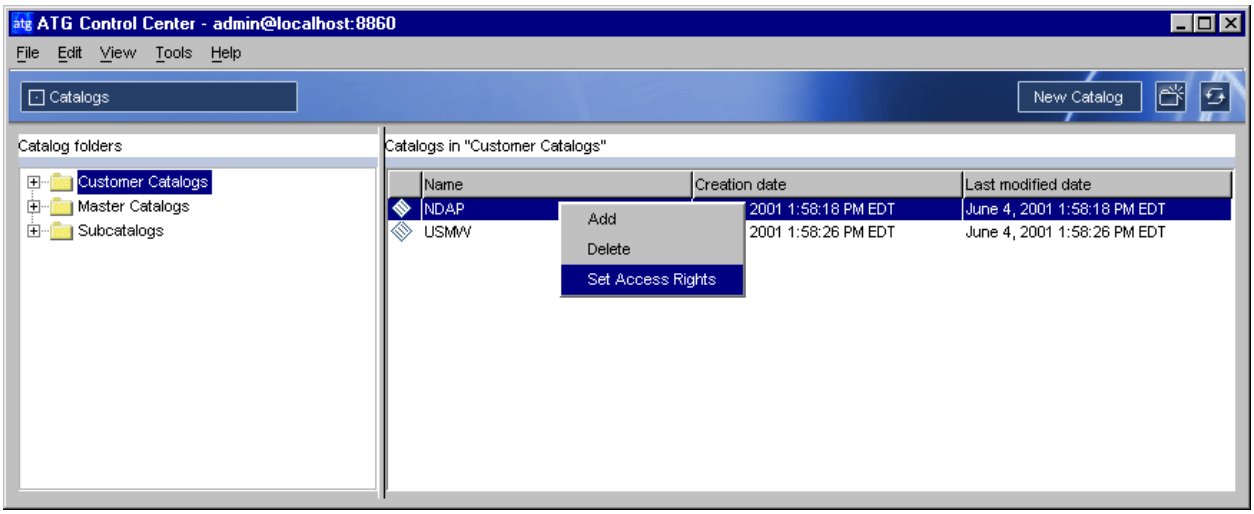
When a user attempts to view or edit a price list or a catalog, the security system checks the security information associated with the object and grants or denies access based on the information. For example, if a user does not have write access to a particular item, then the ACC will display the item in gray characters. Additionally, certain objects might not be visible to certain users. The ACC is capable of checking this security information for all items contained in the price list repository:

See the *ATG Commerce Programming Guide* for more information.

Access rights to catalogs are set in the same way as those in price lists.

For example, to set access rights for the Content QA administrator on the NDAP catalog, you would do the following:

1. In the Catalog Management > Catalogs task section of the ACC, select the Customer catalogs folder
2. Select the NDAP catalog, right-click on it, and select "Set Access Rights."



3. In the Set Access Rights dialog box, click the **Add** button.
4. Select the Roles tab.
5. Expand the Global Roles and then the Control Center Users to show all the previously created Control Center Groups.
6. Select the Content QA group and click OK. This adds Content QA to the Access Rights dialog box.

In the columns that list access rights, select the "List" and "Read" boxes.

13 Exploring Motorprise as a Selling Organization

You can use the ACC to understand how to create new Motorprise customers and to see how security access is set.

This chapter includes the following sections:

Creating New Accounts
Explains how to set up a new organization; assign it a contract, catalog, and pricelist; and create roles and users for it in the ACC.

Exploring Security Privileges
Demonstrates how we created Motorprise ACC users with security access to different catalogs and pricelists.

Creating New Accounts

To set up a new Motorprise customer account, a Motorprise employee must first use the ACC to create the new organization and a company administrator -- an employee of that organization. Once that administrator exists, he or she can create new suborganizations and users.

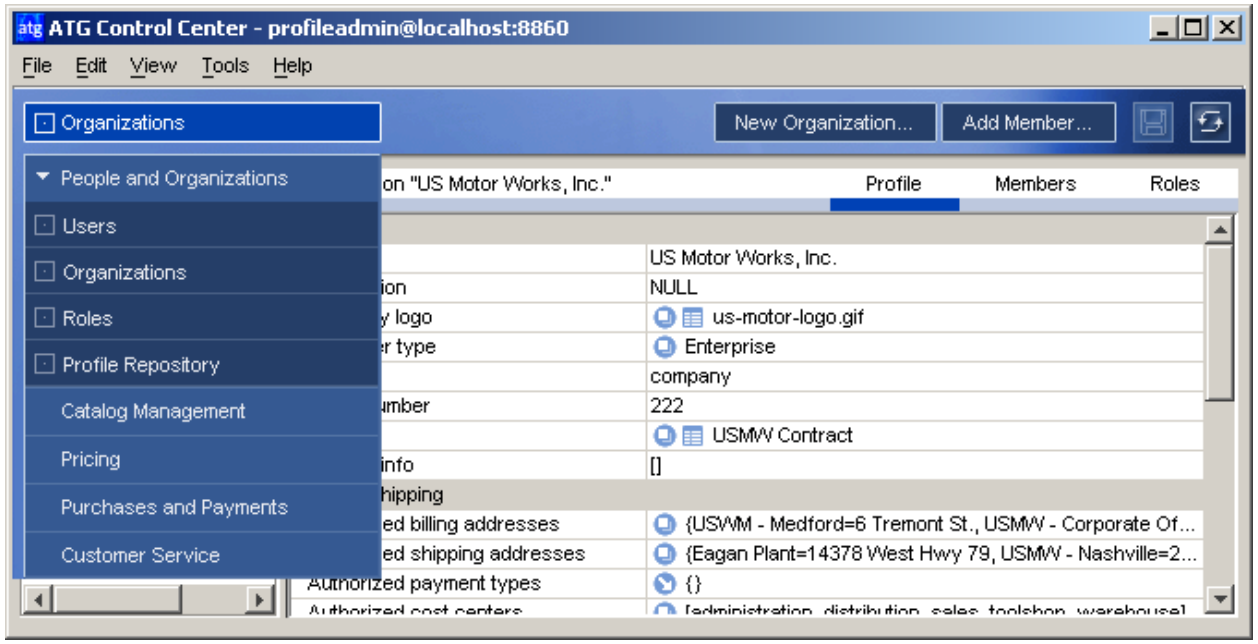
Users inherit most information, such as addresses and payment methods, from the organization profile. Some of these properties can be overridden. However, personal user information must be entered individually. Either the individual or the administrator can do this on the Motorprise site or it can be done by a Motorprise employee in the ACC. All organization information must be entered either in the ACC or in the Company Admin section of the Motorprise site.

Adding a New Organization

We created an administrator for Motorprise, Samantha Gresh, who can create new organizations. You can log in as her and explore the process for adding new organizations.

1. Log into the ACC using “profileadmin” as your user name and password.

   Samantha only has access to certain areas of the ACC. For example, she cannot view and edit pages and components or scenarios.



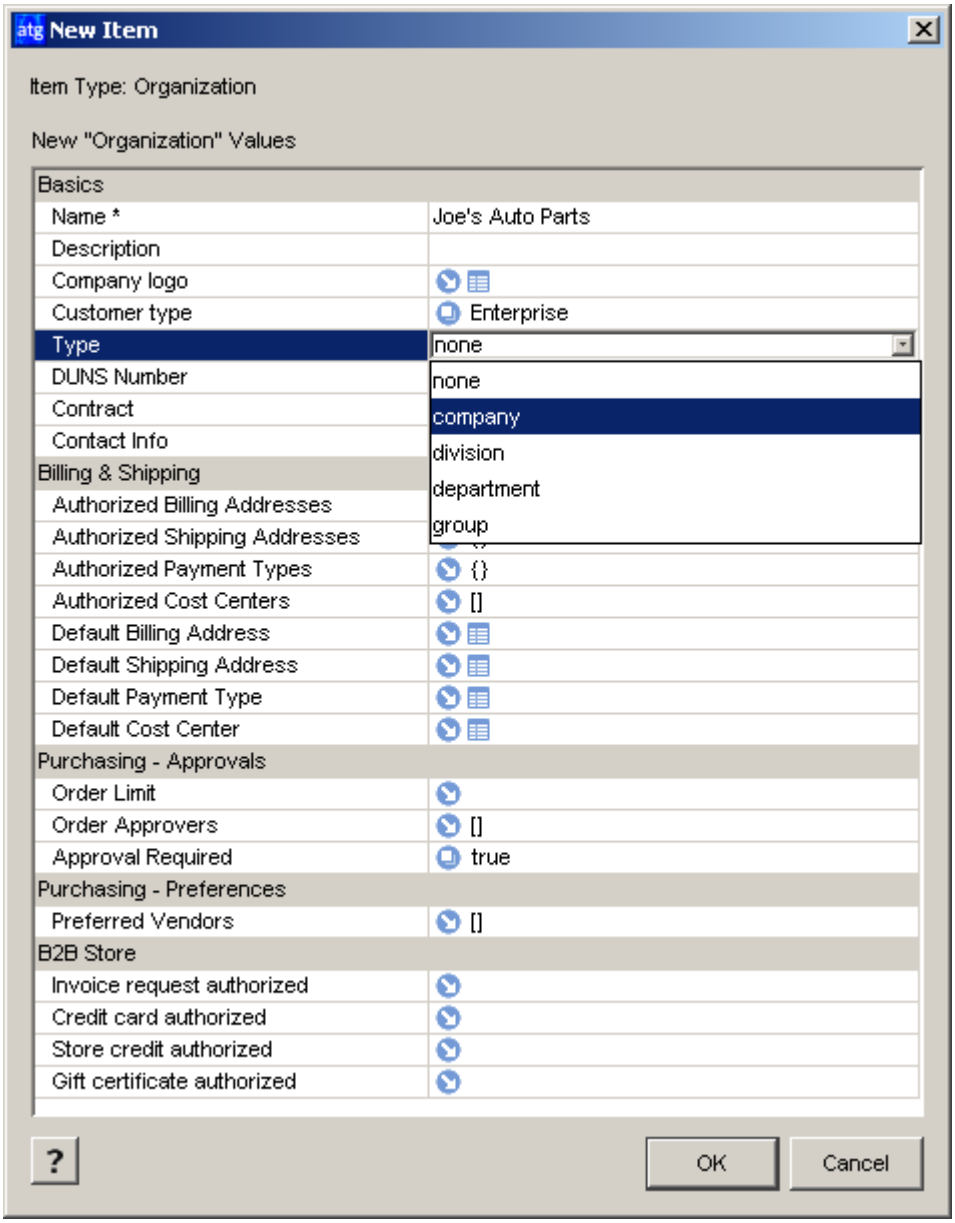
Samantha only has access to certain parts of the ACC.

2. Select People and Organizations > Organizations. The Organizations window appears.
3. In the left pane of the window, select Customer Organizations from the organization tree on the left.

   You must select Customer Organizations because the new organization account will be a top-level organization. It will be the parent organization of any subsequently created suborganizations. If you had selected an existing organization, such as US Motorworks, Inc., then the organization you are creating would be a sub-organization of USMW.
4. Click the "New Organization" button on the top right of the screen. The New Item dialog box appears, listing the organization attributes.
5. Specify values for this organization's profile properties. Properties with an asterisk are required. Other properties are optional.

   Enter the following information:

   - Company name
   - Customer type (Enterprise, Preferred, or Standard)
   - Type (Since this is a new account, select Company. The other options are for sub-organizations.)



6. Click OK.

Notice the icons next to the "Company logo" field.

identifies a derived property with an inherited value. Sub-organizations will inherit this property from the parent organization. In this case, the organization you are creating is itself the parent, so it will not inherit this property.

identifies a property that references (links to) another repository item. This field will be filled by a repository item rather than a string you enter.

Notice the icon next to Customer Type:

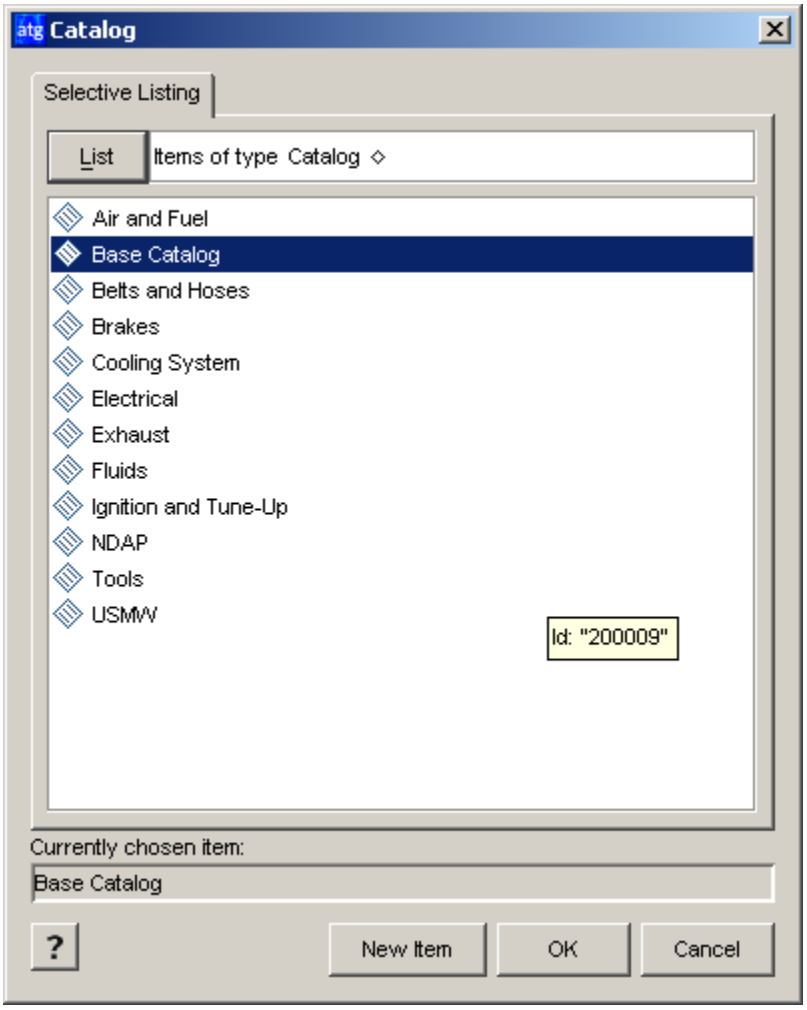
 identifies a derived property value that has been overridden locally. This property is set specifically for this organization and is not inherited.

When there is no icon on a property field, it means that the property cannot be inherited from a parent organization.

Creating a contract

A contract associates a customer with a catalog and a pricelist.

1. Select the organization you just created.
2. Highlight the "Contract" field and click the "**...**" button.
3. Click New Item from the dialog that comes up. Fill in the following properties:
    - Name
    - Catalog : Click the "**...**" button, then the **List** button in the dialog that comes up. Select the Base Catalog, or create a new customer catalog by adding subcatalogs to the new one you create. See the B2B Personalization chapter of this guide for more details.



- Pricelist: Click the "**...**" button, then the **List** button in the dialog that comes up. Select the Default Price List. See the *Creating Price Lists* section in the B2B Personalization chapter of this guide for more details.

4. Click OK.

You can now enter additional information to the organization profile.

Entering Addresses

1. Select an address field such as "Authorized Billing Addresses" and click the "**...**" button.
2. Click the **Add** button and then the **New Item** button.
3. Enter the address information and then click OK.
4. Enter a name for the new address. This name will be a key in the address table.
5. Click OK on the address dialog.

Entering Payment Types

In Motorprise, the payment types used are credit cards and invoices. If the new organization uses credit cards, you can enter them. If you enter a credit card, you must also set the organization's "Credit Card Authorized" field to true. If the company uses only invoices, do not enter any credit cards and set "Invoice Request Authorized" to true and "Credit Card Authorized" to false.

If the company uses cost centers, you can enter those. (Don't select from the existing cost centers because they belong to other organizations. In the Company Admin section of the Motorprise site only cost centers that apply to that organization will appear.)

If the company uses an approval process, set the "Approval Required" field to true and enter a dollar amount in the "Order Limit" field. An order limit of 0 will make all orders require approval. You will specify the order approvers later after you have created those users and assigned their roles.

Creating a Company Administrator

The next step is to create a company administrator for the new organization.

1. Click the Add Member button and on the dialog that comes up click New Item.
2. Fill in the following fields:
   - Login name
   - Password
   - First name
   - Last name
   - Member: set to true
   - Locale: Motorprise supports English (en_US) and German (de_DE).
   - Email address
   - Receive email

   All other information is inherited from the organization or the user can enter it themselves later.
3. Click OK.

Creating Roles

Next, you need to create roles for the new organization.

1. Select People and Organizations > Roles.
2. Open Organizational Roles and select your new organization from the list.
3. Click the New Role button at the top right.

   Motorprise uses three roles: Administrator, Buyer, and Approver. If the new company doesn't require approvals you only need to create the first two for this organization.
4. First create the administrator role by entering the following:

- Name – Admin
- Description – Company administration
- Function – admin (This must be lower case because we use it to target content on the Motorprise site.)

5. Next create the buyer role:
   - Name – Buyer
   - Description – Makes purchases on the site
   - Function – buyer (must be lower case)
6. Create the approver role:
   - Name – Approver
   - Description – Approve orders over a buyer's order limit
   - Function – approver (must be lower case)

When an administrator creates suborganizations in the Company Admin section of the Motorprise site, these roles will be created automatically.

Assigning Roles

Now you assign the role of Admin to the administrator user you created.

1. In the People and Organizations > Roles window, select the role you want to assign.
2. Select File > Add Member. The Add Member dialog box appears.
3. Select the organizations tab and the organization to which you want to assign the role and click OK.
4. Do the same thing for the user.

Now your users should be able to log in to Motorprise and make purchases. To be certain, check the organization's profile and be sure they have at least one billing and shipping address and either Invoice Request or Credit Card Authorized is set to true. Then check the profile of the buyer to be sure he is inheriting this information from the organization.

See the *Creating Organizations* section of the *Setting Up Visitor Profiles* chapter of the *ATG Personalization Guide for Business Users* for more information.

Exploring Security Privileges

We created four Motorprise employees who have specific privileges for accessing price lists and catalogs. You can log in as these Motorprise employees to see how we created different levels of security.

We chose to demonstrate secure repositories with the price list and catalog repository. You could set other security levels, even properties, depending on your business needs.

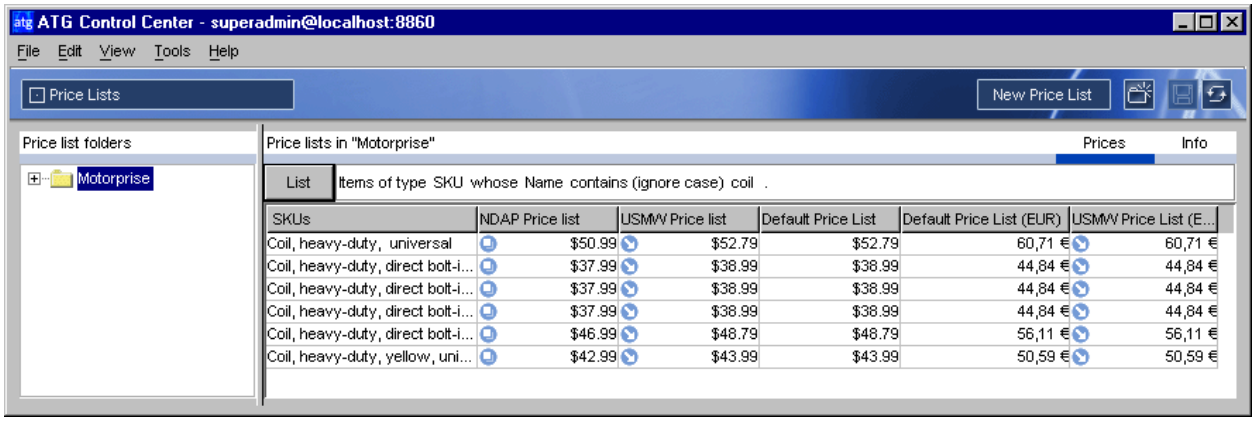
Motorprise Employees

User	Role	Privileges	Username/password
Manoj Parrish	Motorprise Superadmin	Can read and edit all price lists and catalogs	superadmin
Eddie Sherfuddin	Motorprise Content QA	Can read only all price lists and catalogs	contentqa
Sam Streb	Motorprise Account Manager for NDAP	Can read and edit the NDAP price list and catalog; can only read the default price list and catalog; cannot view other customer price lists or catalogs	ndapmanager
Manny Shaw	Motorprise Account Manager for USMW	Can read and edit the USMW price list and catalog; can read the default price list and catalog; can read the default price list for Europe (because they manage the German Division); cannot view other customer price lists or catalogs	usmwmanager

Log in to the ACC as Manoj Parrish (using "superadmin" as your username and password). You will notice that you only have access to the Catalog Management, Pricing, Purchases and Payments and Customer Service sections. (If you had logged on using "admin" as your username and password, you would have access to all sections of the ACC and be able to access components, media, etc.)

Go to the Pricing > Price Lists section and click on the Motorprise folder. Manoj has access to defaultPriceList, defaultPriceList_EUR, NDAP Price list, and USMW Price List.

Do a search for SKUs whose name contains Coil. Here you can see the different prices for the different organizations. Click on any price and modify it. Note that Manoj can edit any price list.



Editing pricelists in the ACC.

Log out of ACC and log back in as a different user to see their price list and catalog privileges.

14 Request-Handling Pipeline Servlets

This chapter describes the two request-handling pipeline servlets we created for Motorprise.

CheckSessionExpiration
This pipeline servlet checks to see if a user's session has expired.

SetCurrentLocation
This pipeline servlet highlights the top navigation bar of the site according to a user's location.

CheckSessionExpiration

A session is an object created at a user's request; it continues to be available to the user for the lifetime of the interaction. Sessions are used to maintain states and user identities across multiple page requests.

When a user first accesses a page, a session is created. For every subsequent page request, the same session is maintained.

Session Expiration from Lack of User Activity

The session expires, or becomes invalid, when there is no activity from the user for a predefined length of time. By default, session timeout is thirty minutes; it is set by the `sessionInvalidationTime` property of `/atg/dynamo/servlet/session/SessionManager`. Once the session expires, if the user tries to access any page, he or she is not recognized and is not served the expected pages. For example, if a user requests the checkout page and then takes no actions for more than thirty minutes, when he or she returns and tries to submit an order, an error is generated.

In Motorprise, we wanted to avoid these session expiration errors. We used the `CheckSessionExpiration` servlet, an instance of the class `atg.projects.b2bstore.servlet.WACheckSessionExpiration`, that is invoked in a request-handling pipeline. It checks the session of the user. If the session is valid, it passes the request to the desired page. If the session is expired, instead of generating an error, it redirects the user to a login page that instructs him or her to log in again. The `relativeExpirationURL` property of this servlet specifies the page to which to redirect the user.

We used this approach to session expiration for both registered users and anonymous users.

Session Expiration from Failover

When a server serving a session fails and another server handles that session, the session ID of the request becomes invalid. A new session is created that restores all the state and identity information of old session. The `sessionRestored` property of the `DynamoHttpServletRequest` is set to the old session ID. Thus, when a session ID is invalid in Motorprise, we check `sessionRestored` to ensure that it is not a result of this session failover condition.

The properties file of `CheckSessionExpiration` at `<ATG10dir>Motorprise/config/atg/projects/b2bstore/servlet/` sets the `insertAfterServlet` property to `/atg/dynamo/servlet/pipeline/SessionSaverServlet`.

This is the `CheckSessionExpiration` properties file at `<ATG10dir>MotorpriseJSP/j2ee-apps/motorprise/config/atg/projects/b2bstore/servlet/CheckSessionExpiration`:

```
$class=atg.projects.b2bstore.servlet.WACheckSessionExpiration

# Specify where in the servlet pipeline this servlet should appear
insertAfterServlet=/atg/dynamo/servlet/pipeline/SessionSaverServlet

# Specify the web application registry where Motorprise is registered
webAppRegistry=/atg/registry/webappregistry/ServletContextWebAppRegistry

# Specify the name under which the Motorprise web app is registered
webApplicationName=MotorpriseJSP

# Specify the root path (relative to the web app context root) of URLs'
# that we should check for session expiration

relativeExpirationPath=/

# Specify the URL (relative to the web app context root) that we should
# redirect to if an expired session is detected.

relativeExpirationURL=/
```

The property `relativeExpirationPath` and `relativeExpirationURL` properties of `CheckSessionExpiration` are configured to use the URLs relative to the web application's context root and convert them to absolute URLs at runtime.

SetCurrentLocation

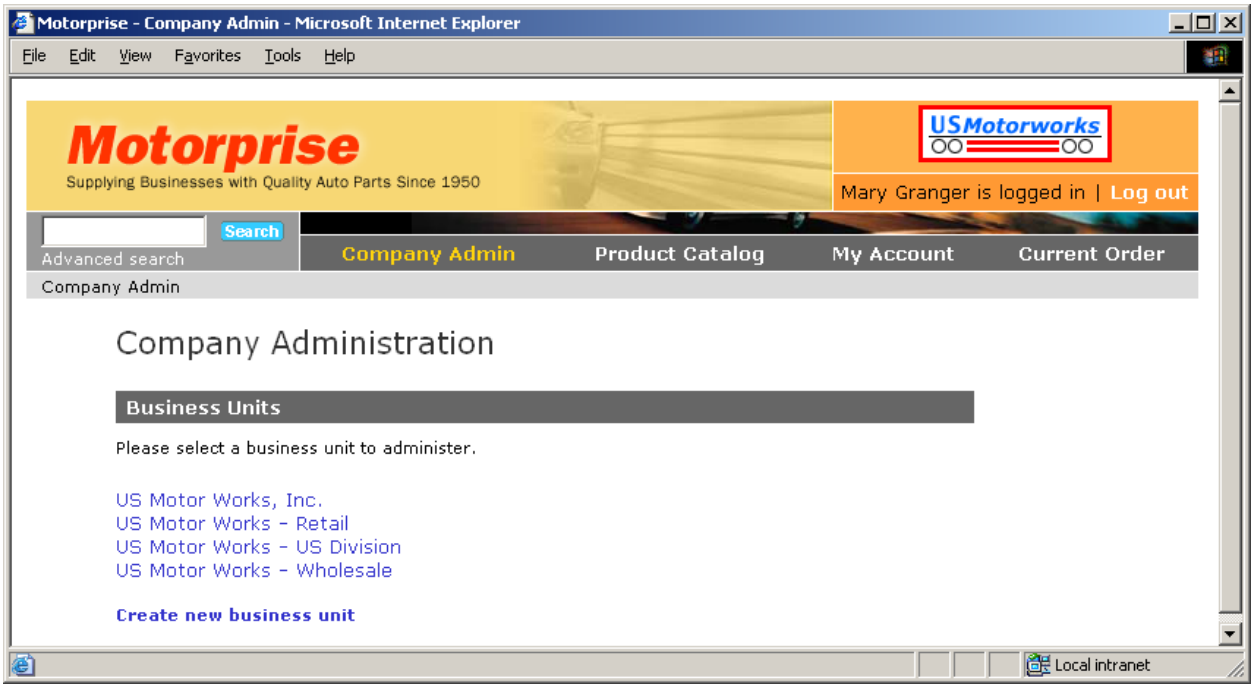
Motorprise has different page sections such as Company Admin, Product Catalog, My Account, and Current Order. Each section contains different pages. When a user is browsing one of these sections, its title on the top navigation bar is highlighted.

In order to identify this location of the user and highlight the corresponding section title, we developed a servlet called `SetCurrentLocation`, an instance of the class `atg.projects.b2bstore.servlet.WASetcurrentLocation`, that is invoked during the request-handling pipeline.

We created a new transient property for the user, `currentLocation`, that stores the section he or she is accessing. The snippet below from `/en/common/BrandNav.jsp` shows how we check this property to highlight the appropriate section:

```
<dsp:droplet name="Switch">
  <dsp:param bean="Profile.currentLocation" name="value"/>
  <dsp:oparam name="admin">
    <td align="center"><dsp:a href="../admin/business_units.jsp">
    <b><font color="#FDD30E" size=-1>Company Admin</font></b></dsp:a></td>
  </dsp:oparam>
  <dsp:oparam name="default">
    <td align="center"><dsp:a href="../admin/business_units.jsp">
    <b><font color="#FFFFFF" size=-1>Company Admin</font></b></dsp:a></td>
  </dsp:oparam>
</dsp:droplet>
```

If the location is equal to `admin`, we highlight the Company Admin section of Motorprise in yellow as shown in the screenshot below.



When a user is in the Company Admin section, its title is highlighted in yellow.

SetCurrentLocation has a map property, relativePathMap, that specifies the mapping between URLs and virtual parts of the store, using URLs relative to the web application context root. It converts these to absolute URLs at runtime. This property contains the relative document paths for each section (Company Admin, Product Catalog, My Account, and Current Order) and the value to be set in the location property if the user accesses any page from one of those paths. This is the properties file for MotorpriseJSP/j2ee-apps/Motorprise/config/atg/projects/b2bstore/servlet/SetCurrentLocation:

```
$class=atg.projects.b2bstore.servlet.WASetCurrentLocation

# Specify where in the servlet pipeline this servlet should appear
insertAfterServlet=/atg/userprofiling/ProfileRequestServlet

# Specify the web application registry where Motorprise is registered
webAppRegistry=/atg/registry/webappregistry/ServletContextWebAppRegistry

# Specify the name under which the Motorprise web app is registered
webApplicationName=MotorpriseJSP

# Set URL's relative to the Motorprise context root.

relativePathMap=/en/catalog/=product_catalog,\
            /en/admin/=admin,\
            /en/home.jsp=product_catalog,\
            /en/checkout/=current_order,\
            /en/user/=my_account,\
            /de/catalog/=product_catalog,\
            /de/admin/=admin,\
            /de/home.jsp=product_catalog,\
            /de/checkout/=current_order,\
            /de/user/=my_account,\
            /ja/catalog/=product_catalog,\
            /ja/admin/=admin,\
            /ja/home.jsp=product_catalog,\
            /ja/checkout/=current_order,\
            /ja/user/=my_account
```

When the user requests a new page, the URL is compared to the paths in the relativePathMap property. The servlet sorts the keys of the map, which are document paths, in ascending order, and checks whether the directory path exists in the request. If there is an exact match, then the corresponding value is assigned to the location property. If there is not an exact match, the closest directory to which the URL belongs is found and the value of that directory is added to location.

Starting the Servlets

In order to insert `SetCurrentLocation` and `CheckSessionExpiration` into the servlet pipeline, they must start at the same time as the ATG platform. To accomplish this, we added the following lines to the `/atg/dynamo/servlet/Initial` component:

```
initialServices+=/atg/projects/b2bstore/servlet/SetCurrentLocation,\
            /atg/projects/b2bstore/servlet/CheckSessionExpiration
```

These servlets are located in `<ATG10dir>/Motorprise/src/Java/atg/projects/b2bstore/servlet/`.

Index