# BEA AquaLogic Commerce Services

## Developer Guide

Version 6.0
February 2008

# Copyright



# Restricted Rights Legend



# Trademarks and Service Marks

# Contents

# Overview

This document is intended to help developers build sophisticated online stores by providing detailed technical explanations of all areas of the AquaLogic Commerce Services product.

Sections include:

- **Setting up your development environment**
    - recommendations for setting up an efficient development environment to work with AquaLogic Commerce Services applications
- **Programming with AquaLogic Commerce Services**
    - helps a developer get familiar with the **core concepts** needed to build e-commerce applications on the AquaLogic Commerce Services platform
- **Architecture reference**
    - explains the underlying architectural aspects of AquaLogic Commerce Services that apply to a wide range of features

# 1 - Setting up your development environment

Here we provide our recommendations for setting up an efficient development environment to work with AquaLogic Commerce Services applications. This configuration is used by our own product development team and is therefore our recommended setup, but you are of course free to use a different IDE, application server and database server for development.

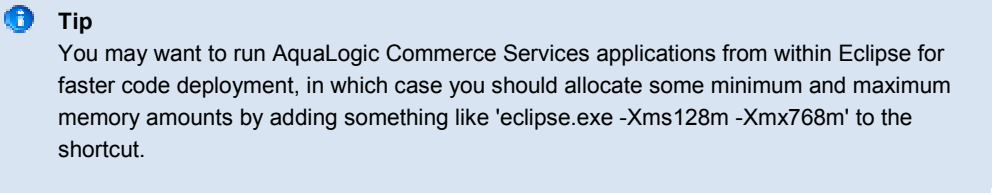
## Setting up your IDE

### *Downloading and installing Eclipse*

> **Note**
> If you are using BEA WorkSpace Studio, you will already have tight integration with WebLogic, and will not need to install the Eclipse WTP plugins.

If you are not using WorkSpace Studio, but are using Eclipse IDE as your development environment:

1. Download and unzip the Eclipse SDK from: http://www.eclipse.org/europa/. Eclipse 3.3 is required to develop on AquaLogic Commerce Services 6.0.
2. Start Eclipse

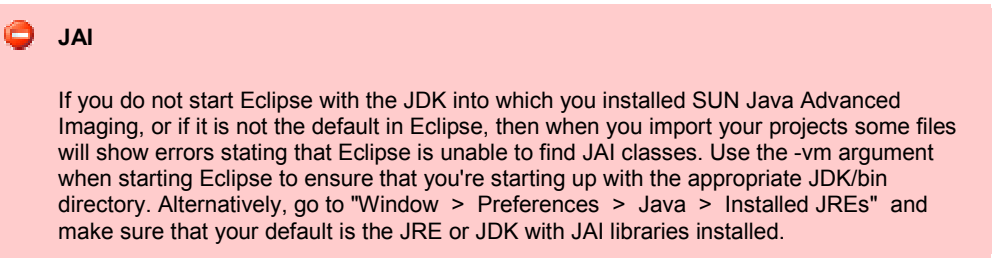



3. Create or Select a workspace.

## *Installing Eclipse plugins*

1. After the startup of Eclipse open the Install / Update dialog via the menu entry "Help > Software Updates > Find and Install ...".
2. Select "Search for new features to install".
3. Check the box for "Europa Discovery Site", and then click the "Finish" button.
4. After picking your mirror(s), you will have the opportunity to select the plugins you wish to install. We recommend installing, at a minimum:
    - Charting and Reporting - > Eclipse BIRT Report Designer Framework
    - Graphical Editors and Frameworks - > Graphical Editing Framework
    - Java Development - > J2EE Standard Tools (JST) Project
    - Testing and Performance - > (ALL) (Note: Please see TPTP for more information)
    - Web and J2EE Development - > Web Standard Tools, J2EE Standard Tools
5. Then click the "Select Required" button to resolve any dependencies, and click "Next" to continue with the installation process.

## *Setting up your environment*




## *BEA WorkSpace Studio and Eclipse Plugins*

This section describes several plugins for BEA WorkSpace Studio and Eclipse that can be useful when customizing AquaLogic Commerce Services.

## Agent Controller

The standalone Agent Controller is necessary for using the TPTP profiling tool to handle remote profiling. TPTP has an integrated agent controller that will work for most people when profiling on a localhost. However, if you want to install that standalone Agent Controller, you still can.

### Installation

1. Download Agent Controller
   Visit the TPTP web site
   Find, download, and unzip the agent controller .zip file into a directory.
2. At the bottom of the download page there are links to documentation, including install docs.

## TPTP plugin for Eclipse

The Eclipse Test & Performance Tools Platform (TPTP) Project is an open source Top Level Project of the Eclipse Foundation, and is installed as part of the Callisto plugin install process. TPTP is divided into four projects:

- TPTP Platform: the TPTP Platform Project encompasses a large amount of common infrastructure and capability which the other TPTP projects expand and specialize. It provides common user interface, standard data models, data collection and communications control, as well as remote execution environments
- Monitoring Tools: it addresses the monitoring and logging phases of the application lifecycle.
- Testing Tools: it addresses the testing phase of the application lifecycle.
- Tracing and Profiling Tools: it addresses the tracing and profiling phases of the application lifecycle.

(Above content obtained from: http://www.eclipse.org/tptp/)

### Agent Controller

The standalone Agent Controller is necessary for using the TPTP profiling tool to handle remote profiling.
As of the Callisto Eclipse bundle, TPTP has an integrated agent controller that will work for most people when profiling on a localhost.
However, if you want to install that standalone Agent Controller, you still can.

### Installation

- Download Agent Controller
  Visit the TPTP web site at http://www.eclipse.org/tptp/.
  Find, download, and unzip the agent controller .zip file into a directory.

- At the bottom of the download page there are links to documentation, including install docs.

## Velocity plugin for Eclipse

Velocity is a Java-based template engine. It permits anyone to use a simple yet powerful template language to reference objects defined in Java code.

When Velocity is used for web development, Web designers can work in parallel with Java programmers to develop web sites according to the Model-View-Controller (MVC) model, meaning that web page designers can focus solely on creating a site that looks good, and programmers can focus solely on writing top-notch code. Velocity separates Java code from the web pages, making the web site more maintainable over its lifespan and providing a viable alternative to Java Server Pages (JSPs) or PHP.

(Above content obtained from: http://jakarta.apache.org/velocity/)

### Installation

1. Start Eclipse.
2. Select Help > Software Updates > Find and Install...
3. Click Next >.
4. Click New Remote Site....
5. Enter the name and URL:

| | |
|---|---|
| Name: | **Veloeclipse** |
| URL: | http://propsorter.sourceforge.net/veloeclipse/ |

6. Click Finish.
7. Select the latest version
8. Click Next >.
9. Click I accept the terms in the license agreements.
10. Click Next >.
11. Click Finish.
12. Feature verification: click Install.

### Usage

Any *.vm* file will be opened in the Velocity Editor and allow you to use (limited) code completion. See the website for more details.

## Amateras XML Editor plugin for Eclipse

Eclipse HTML Editor is an Eclipse plugin for HTML/JSP/XML Editing. It works on Eclipse 3.0 (or higher), JDT and GEF. It has following features.

- HTML/JSP/XML/CSS/DTD/JavaScript Highlighting
- HTML/JSP Preview
- JSP/XML Validation
- Contents Assist (HTML Tags/Attributes, XML based on DTD and JSP taglib and more)
- Wizards for creating HTML/JSP/XML files
- Outline View
- Editor Preferences
- Editor Folding
- Web Browser (It works as an Eclipse's editor)
- Image Viewer
- Tag Palette
- CSS code completion and outline
- DTD code completion, outline and validation
- JavaScript code completion, outline and validation

(Above content obtained from:
http://amateras.sourceforge.jp/cgi-bin/fswiki_en/wiki.cgi?page=EclipseHTMLEditor)

### Installation

1. Close Eclipse.
2. Go to https://sourceforge.jp/projects/amateras/files/.
3. Download the latest version of EclipseHTMLEditor.
4. Unzip the zip file into the directory that holds your plugins directory for Eclipse.
5. Start Eclipse.

### Usage

If you make sure you add a DTD to the XML file then the Amateras XML editor (you may have to use *Open With* to start using it) will create an *Outline* view and enable you to use code completion.

## *BEA WorkSpace Studio and Eclipse Tips*

- Put your cursor on the interface name or method name, press "CTRL+T", it will show a drop down of all implementations. You can select one you want to go. This is very helpful with all the Spring interfaces going around.

# Other Development Tools

This section describes other tools that may be helpful for developing AquaLogic Commerce Services applications.

## *Web browser tools*

- Web developer extension for Firefox/Mozilla. This plug-in provides a variety of handy features for developing web pages, including W3C compliance checking. http://chrispederick.com/work/webdeveloper/
- Live HTTP Headers. Shows real-time HTTP header data http://livehttpheaders.mozdev.org/

# 2 - Programming with AquaLogic Commerce Services

This section is intended to help a developer get familiar with the **core concepts** needed to build e-commerce applications on the AquaLogic Commerce Services platform.

The explanations and examples provided in this section are deliberately **brief and concise** because the aim is **to get you productive as quickly as possible** without you having to learn all the details of what's going on under the hood. For more details on the underlying architecture, refer to the Architecture Reference.

# Tutorial 1 – Customizing orders

This tutorial conveys the general idea of how the system can be customized and a certain feature implemented without making changes to the product's core code. Starting with the domain model, going through the store front and finishing with the Commerce Manager (CM) client we will briefly describe how to do a small change to the order domain object and represent this change to customers (in the storefront) and CSRs (in the Commerce Manager application).

An imaginary software company needs to sell its anti-virus products. Customers should be able to choose the subscription level and there has to be an option on checkout saying "Renew my subscription when it expires". This will be later on used to automatically charge the customer by a certain amount of money in order to renew their license subscription.

Here's what you are going to learn by following the tutorial:

- How to make simple modifications to the database schema.
- How to modify a domain model object and have its new data automatically persisted and retrieved from the database.
- How to change the Store Front to make domain model changes visible to customers of the store, this includes changes to emails sent to the customer.
- How to change the CM Client making domain model changes visible to the store's CSRs.

## *Modify the database to suit the domain model needs*

At this step the changes are related to the idea of extending the existing order object. We are going to use an additional table that will act as an extension to the TORDER

table. By joining the two tables we will be able to represent the original table along with the one we will create as one object.

## Adding AUTOBILL column to a new table

All database related SQL files are located in your installation under dbscripts/<your_db_type> folder. The file that we are interested in is dbscripts/<your_db_type>/schema.sql. In this step we assume that the database is MySql and therefore the SQL syntax is MySql related.

Open schema.sql with your favorite editor and add to the end of the file the following SQL entry. The new order extension table declaration should look like:

```
...
# ----------------------------------------------------------------------
# TORDEREXT
# ----------------------------------------------------------------------
CREATE TABLE TORDEREXT
(

UIDPK BIGINT NOT NULL,
AUTOBILL INTEGER default 0,
PRIMARY KEY(UIDPK)

);
```

The new table column is of type integer which will represent a boolean value in the database as values '0' = false and '1' = true.

If you only want to change an already existing database you can commit the above query for adding that new table to the database directly using a database client.

## Additional column to TORDER table

One more change is necessary in order to have JPA working correctly. And that is adding a column to the TORDER table.
The idea behind that change is to have a discriminator value that acts as an ID to the object that will extend the *OrderImpl* class. Using different discriminators allows for distinguishing different subclasses of the original *OrderImpl* class.

This might be done in the *schema.sql* file or directly executing a query to the database.

```
ALTER TABLE TORDER ADD COLUMN TYPE VARCHAR(20);
```

## *Extend the domain object model*

### Setting up BEA WorkSpace Studio Projects

Before going to the next steps we will have to create WAR files out of the subfolders in the *commerceApp* exploded application. This step is necessary because BEA WorkSpace Studio only supports import of WAR files and we have our *commerceApp* application exploded by default. The ALCS installer creates the *commerceApp* in the following folder *<BEA_HOME>\commerce_6.0\samples\commerce*. We have to zip the *commerceServices* folder contents to *commerceServices.war*, the *commerceServicesManager* folder contents to *commerceServicesManager.war* and the *commerceServicesSearch* folder contents to *commerceServicesSearch.war*.

Next we have to import the required projects. Here's what has to be done step-by-step:

1. Go to **File**->**Import...**
2. Choose **Web/WAR file** from the tree. Click **Next**.
3. For **WAR file** you have to locate the *commerceServices.war* file.
4. Select **Next**.
5. On the next page find the jar file named *com.bea.alcs.core-6.0.jar* and check it.
6. Select **Finish**

The result of following those steps would be the creation of two projects named *storefront* and *com.bea.alcs.core-6.0*. The *storefront* project is a web project and should have reference to the *com.bea.alcs.core-6.0* project.

The same procedure has to be performed for the *commerceServicesManager.war* file.

1. Go to **File**->**Import...**
2. Choose **Web/WAR file** from the tree. Click **Next**.
3. For **WAR file** you have to locate the *commerceServicesManager.war* file.
4. Select **Next**.
5. On the next page find the jar file named *com.bea.alcs.core-6.0.jar* and check it.
6. Select **Finish**

As an option you can also have the *commerceSerivcesSearch.war* imported in the same manner.

### Setting up the server runtime environment

It is possible to launch the web projects using a BEA WebLogic Server supported by BEA WorkSpace Studio.

1. Go to **Window**->**Show View**->**Other...**
2. Browse for **Server** category and select **Servers** view. Click OK.

3. Right click over the main view area and choose **New**->**Server**
4. In the dialog that came up choose your BEA WebLogic Server release and then **Next** button.
5. Browse for the BEA WebLogic Server domain home that you are going to use.
6. Add the *commerceServicesManager* and *commerceServices* projects (also *commerceServicesSearch* if you imported it) to the ones to the **Configured Projects** section.
7. Select **Finish**

Before giving a try to start the server we have to setup a Data Source so that the application is able to access the database. For reference on how to do that please refer to the Deployment Guide.

The BEA WebLogic Server configuration has to be refreshed and this could be done in the **Servers** view by right clicking over the server node and selecting **Publish**. This will build the *com.bea.alcs.core.jar* and build all the WAR files and deploy them to BEA WebLogic Server.

As we imported all the jars located in the war files it is possible that you may get a problem saying that the *com.bea.alcs.core.jar* file already exists in the web projects. You will have to go and delete the jar file from each web project's WEB-INF/lib folder.

So now we can start our web server and see if everything is OK. In the **Servers** view select **Start the server** button. All the output will be dumped in the **Console** view. For troubleshooting you might need to go and check the web project log files in case the log4j logger is set to type *FILE*. The folder path can be seen in the Server configuration editor opened by double click over the server node in **Servers** view.

**Useful Information**

If you get exception starting the server this could be a problem with the license. You can fix it by modifying the **commerce-config.xml** file and pointing the property **catalog.asset.path** to the folder where assets folder is on your hard drive:

<property name="catalog.asset.path" value="c:/alcs6/assets"/>

This will be necessary to be changed for all the projects imported in the previous step.

## Creating a new class that extends the Order class

Adding the additional field to the order means creating additional class extending the already existing *Order* class. In the newly created class we will have to add the accessors for the *autoBill* field.
All the changes for the core model will be done in the *com.bea.alcs.core-6.0* project.
So first let's create the interface. Here's what it should be like:

### com.example.extendorder.ExtOrder.java

```
package com.example.extendorder;
```

```
import com.bea.alcs.domain.order.Order;

public interface ExtOrder extends Order {

        /**
         * Checks whether this order has been set as an auto bill order.
         *
         * @return true in case the order is of type auto bill
         */
        boolean isAutoBill();

        /**
         * Sets auto bill flag on this order.
         *
         * @param autoBill the auto bill flag
         */
        void setAutoBill(boolean autoBill);
}
```

The interface implementation would be like the following listing:

### com.example.extendorder.impl.ExtOrderImpl

```
package com.example.extendorder.impl;


import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

import com.bea.alcs.domain.order.impl.OrderImpl;
import com.example.extendorder.ExtOrder;

@Entity
@Table(name = ExtOrderImpl.TABLE_NAME)
@PrimaryKeyJoinColumn(name="UIDPK", referencedColumnName="UIDPK")
```

```
@DiscriminatorValue("extOrder")
public class ExtOrderImpl extends OrderImpl implements ExtOrder {

	/**
	 * Serial version id.
	 */
	public static final long serialVersionUID = 5000000001L;

	public static final String TABLE_NAME = "TORDEREXT";

	private boolean autoBill;

	/**
	 * Checks whether this order has been set as an auto bill order.
	 *
	 * @return true in case the order is of type auto bill
	 */
	@Basic
	@Column(name = "AUTOBILL")
	public boolean isAutoBill() {
		return autoBill;
	}

	/**
	 * Sets auto bill flag on this order.
	 *
	 * @param autoBill the auto bill flag
	 */
	public void setAutoBill(final boolean autoBill) {
		this.autoBill = autoBill;
	}
}
```

Having all those annotations is probably somewhat confusing. All we want to tell JPA is that this class has its *UIDPK* column which is the primary for the *TOREDEREXT* table and the ID of this class in the parent table *TORDER* will be *extOrder*. This will allow JPA to distinguish between the parent class and its subclass. We also define the class as an entity being mapped towards the *TORDEREXT* table.

The serial version UID is used for serialization/deserialization purposes.

## Creating a new class that extends the shopping cart class ExtShoppingCartImpl.java

```
package com.example.extendorder.impl;

import com.bea.alcs.domain.shoppingcart.impl.ShoppingCartImpl;

public class ExtShoppingCartImpl extends ShoppingCartImpl {

	/**
	 * Serial version id.
	 */
	public static final long serialVersionUID = 5000000001L;

	private boolean autoBill;


	/**
	 * Checks whether this order has been set as an auto bill order.
	 *
	 * @return true in case the order is of type auto bill
	 */
	public boolean isAutoBill() {
		return autoBill;
	}

	/**
	 * Sets auto bill flag on this order.
	 *
	 * @param autoBill the auto bill flag
	 */
	public void setAutoBill(final boolean autoBill) {
		this.autoBill = autoBill;
	}

}
```

You might notice that the created classes do not compile. This is fixed by going to the project **Properties**->**Java Build Path**->**Add JARs...** and selecting/adding all the jar files from *storefront/WEB-INF/lib* folder.
Now the two class files should compile properly.

Next we need to register the implementation *ExtOrderImpl* class to be a valid entity so that JPA recognizes it at runtime.
In *com.bea.alcs.core-6.0/src/META-INF* folder you will find the *persistence-renamed.xml* file holding all the entity class declarations. We need to add the following line to declare our new class:

### persistence-renamed.xml

```
...
        <class>com.example.extendorder.impl.ExtOrderImpl</class>
...
```

For the new extension to take affect we have to tell JPA that the extended class (*OrderImpl*) accepts extensions. Overriding the entity mapping is performed using a reference from the *persistence-renamed.xml* file to a specified mapping file. The change for the *persistence-renamed.xml* is as follows:

### persistence-renamed.xml

```
...
        <mapping-file>META-INF/order-mapping.xml</mapping-file>
...
        <!-- class declarations follow here -->
```

The mapping file reference has to be before the entity class declarations.
The contents of the *order-mapping.xml* file located in *com.bea.alcs.core-6.0/src/META-INF/* would be like the next listing.

### order-mapping.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
   version="1.0">

  <entity class="com.bea.alcs.domain.order.impl.OrderImpl">
   <inheritance strategy="JOINED"/>
   <discriminator-column name="TYPE" discriminator-type="STRING" />
  </entity>
```

```
</entity-mappings>
```

This tells JPA to add to the exising annotation of *OrderImpl* class the inheritence of type *JOINED*. This will make JPA join the *TORDER* and *TORDEREXT* tables at runtime and they will be used by the new entity called *OrderExtImpl*.

## Enhance classes with OpenJPA

JPA uses runtime or build time enhancement procedure to build the byte code metadata for each class having a persistence nature. For enhancing our newly created classes we are going to use the build time enhancement. This will be performed by an ANT task which will do the task after each Eclipse compilation has been performed.
A file called *build.xml* has to be created in the *src* folder of *com.bea.alcs.core-6.0* project.

### build.xml

```
<project name="enhance" default="enhanceJava">


        <!-- define the openjpac task; this can be done at the top of the    -->
        <!-- build.xml file, so it will be available for all targets      -->
      <taskdef name="openjpac"
classname="org.apache.renamed.openjpa.ant.PCEnhancerTask">
                <classpath>
                <fileset dir="../../storefront/WebContent/WEB-INF/lib">
                    <include name="*.jar"/>
                </fileset>
                </classpath>
      </taskdef>


      <target name="enhanceJava">
        <!-- invoke enhancer on all .java files below the model directory -->
        <openjpac>
                <config propertiesFile="./META-INF/persistence-renamed.xml"/>
          <fileset dir="../build/classes">
            <include name="**/*.class" />
          </fileset>
                <classpath>
                        <pathelement location="../build/classes"/>
                        <pathelement location="../ImportedClasses"/>
                        <fileset dir="../../storefront/WebContent/WEB-INF/lib">
                            <include name="*.jar"/>
```

```
                    </fileset>
                </classpath>
            </openjpac>
        </target>
</project>
```

It is important that the XML file is located in the *src* folder because it has relative path to all the jar files that are needed for running the ANT task. If you want it to be in a different folder, the paths have to be updated.

The task can be either linked to the already existing builders list of the project or run manually after each change in the project. Here's how you can link the ANT task as an automatic builder.

1. Open the context menu over the *com.bea.alcs.core-6.0* project and select the **Properties** menu item.
2. Go to the **Builders** section and select the **New...** button
3. Choose **Ant Builder**
4. In the opened dialog select the already created *build.xml* file from the *com.bea.alcs.core-6.0* project and set the **Base Directory** to be the *src* folder of the project.
5. On the **Refresh** tab check the option **Refresh resources upon completion** and select the option **The project containing the selected resource**
6. On **Targets** tab choose the **Manual Build** and **Auto Build** targets to be the *enahanceJava* target
7. Select OK on both of the open dialogs

An automatic build should be triggered and you should see the result of the JPA enhancer in the **Console** view.

# Tutorial 2 - Customizing One Page checkout

In this tutorial we will see how to customize several different aspects of the One Page checkout. We will begin with some simple layout and style changes and move our way up to more complex modifications. Finally, we will walk through an example of creating a new complex form object and configuring it to be automatically rendered by the javascript code.

**Note**
For instructions on rebuilding OnePage after editing the CSS and JS files, please see the

section Building AquaLogic Commerce Services Applications below.

## Changing the overall layout

The layout of One Page is specified in the checkout.vm Velocity template found in the following Storefront directory:

WEB-INF/templates/velocity/onepage

You can change the positions of different sections here such as the arangement of the cart on the left and the checkout accordian on the right as shown below. Try moving some of the sections around to see how this works.

```
<div style="float:left;width:35%;">
  <h2>#springMessage("checkout.heading.cart")</h2>
  <div id="carteditor-container" style="height:350px;overflow-y:auto;">
    <!-- This is rendered in JavaScript by the CartEditor -->
  </div>
  <div id="summary" style="position:relative;height:210px;">
    . . .
  </div>
</div>

<!-- THE CHECKOUT ACCORDION -->
<div id="checkout_container" style="width:480px; float:right;">
  . . .
</div>
```

All of the forms in the checkout accordian are also specified in this file inside the checkout_container div and can be modified here.

## Customizing the style of elements

Most of the styling is done with CSS in the files found in the Storefront template-resources/stylesheet/onepage directory. The various CSS files are all concatenated together during the build into one file called onepage.css in the directory just previously mentioned. Most of the files you would want to edit are located in the ep/global and ep/ui subdirectories and are organized by the various functional units they correspond to. For now, locate the onepage.css file and make some simple modifications to the colors and positioning of the various elements.

 **Be Careful**

> For quick testing you can edit the styles directly in the onepage.css file. Be sure to make any permanent changes to the other files however, otherwise your changes will be overwritten during the next build.

## *Altering the sequence of steps taken during checkout*

The default behavior during the checkout process is to autofill as many of the accordian panes as possible when the customer logs in (or on page load if the customer is already signed in). If the customer then makes changes to the information in one of the panes the pane directly under it will be opened.

This behavior is specified in the checkout.js file found in the template-resources/js/onepage/ep/ui directory. The logic for setting the pane states on page load and on customer login is shown below. To always open the Shipping pane on login or page refresh try commenting out all the code inside the main if block except for the line that reads "this.getShippingPane().open()".

```
ep.ui.Checkout.prototype.refreshPaneStates = function()
{
  var shippingVerified = false;
  var billingVerified = false;

  var customer = ep.session.customer.getCustomer();
  if (customer)
  {
    if (customer.preferredShippingAddress) {
      shippingVerified = true;
      ep.session.shoppingCart.getCart().selectedShippingAddressUidPk =
        customer.preferredShippingAddress.uidPk;
      this.renderShippingSummary();
    } else {
      this.getShippingPane().open();
    }

    if (customer.preferredBillingAddress) {
      billingVerified = true;
      ep.session.shoppingCart.getCart().selectedBillingAddressUidPk =
        customer.preferredBillingAddress.uidPk;
      this.billingAddressSummary();
    } else if (shippingVerified) {
      this.getBillingPane().open();
```

```
    }

    if (shippingVerified && billingVerified) {
      this.showPaymentPane();
    }
  }
};
```

## *Adding a new complex form object*

Complex form elements are specified in their own javascript classes which handle the tasks of rendering, user interaction, and form element updating. In this example we will create a new complex form element for editing a multi-SKU cart item. This would assume that we did not already have the MatrixSelector class found in matrixselector.js in the template-resources/js/onepage/ep/ui directory.

We would first create the file called matrixselector.js and define the constructor for the form element as shown below.

```
nitobi.lang.defineNs('ep.ui');

ep.ui.MatrixSelector = function(id, item, options, availability, choice, skuGuid)
{
  this.item = item;
  this.options = options;
  this.availability = availability;
  this.setId(id);
  this.skuGuid = skuGuid;
  this.choice = choice || new Array();

  nitobi.event.EventManager.publish('ep.ui.MatrixSelector<[' +
    this.getId()+']>.setChosen');
  nitobi.event.EventManager.subscribe('ep.ui.MatrixSelector<[' +
    this.getId()+']>.setChosen', this, this.userDidChoose);
};

nitobi.lang.extend(ep.ui.MatrixSelector, nitobi.ui.InteractiveElement);
```

The next thing we need to do is create the method that will render the html for the element and setup all the javascript events. The main function for this can be seen below.

```
ep.ui.MatrixSelector.prototype.render = function()
{
  var renderString = "";
  for (var i = 0; i < this.options.length; i++)
  {
    var listId = this.getId() + '_list_'+this.options[i].optionKey;
    renderString +=
      . . .
    for (var j = 0; j < this.options[i].optionValues.length; j++)
    {
      renderString += . . . ;
        . . .
    }
    renderString += . . . ;
  }
  renderString+= . . . ;
  var container = this.getHtmlElementHandle();
  container.innerHTML = renderString;
  this.attachEvents();
  this.updateAvailability();
  this.updateChosen();
  this.updateFormElements();
};
```

A number of other functions to handle the logic controlling user interaction would then need to be written, but we will ignore those for now. To see the full details the matrixselector.js file mentioned above can be referenced.

To have this form element dynamically created we need to tell the form renderer about it. This can be done in the form.js file located in the template-resources/js/onepage/nitobi/ui directory. The method we need to edit is shown below along with the code to use our new form component.

```
nitobi.ui.Form.renderComplexFormElements = function(elements, arguments)
{
  for (var i=0; i < elements.length; i++)
  {
    if (elements[i].childNodes.length > 0)
    {
      nitobi.ui.Form.renderComplexFormElements(elements[i].childNodes, arguments);
```

```
  }

  if (nitobi.html.Css.hasClass(elements[i], "ep-ui-swatchselector-container"))
  {
    . . .
  }
  else if (nitobi.html.Css.hasClass(elements[i], "ep-ui-matrixselector-container"))
  {
    if (!elements[i].javascriptObject)
    {
      matrixSelector = new ep.ui.MatrixSelector(elements[i].id, {}, . . . );

      elements[i].javascriptObject = matrixSelector;
      matrixSelector.render();
    }
  }
  . . .
}
 };
```

# Tutorial 3 – Working with Web Services

## *Overview*

In this tutorial we will see how to modify an existing web service to add new API methods. First, we will design the service API, including creating the message objects, new service methods, and domain data objects. Then we will validate our code with unit tests. Finally, we will build the web service project and verify that the generated client-side library contains the new methods and that the deployed web service works.

The first step in designing a new web service is to consider the service as something that fulfills a business process. It is important to understand that the service should ideally be:

- **Coarse-grained** - it should perform a business task without the need for the client to make many, if any, other service calls
- **Flexible** - it should allow the client to specify what they want to happen and what they want to receive according to well-defined criteria
- **Business-oriented** - it should answer some need of the business and not expose functionality for the sake of exposing it
- **Abstracted** - it should not directly expose core domain objects because those will change without regard for Web Services clients

An example of a business process is "Update an order shipment". It is clear that this process could be realized with many fine-grained service methods, i.e.

1. Find the order by order number and customer id
2. Set the order total on the order
3. Get the shipment from the order
4. Set the new carrier, tracking code and shipment date on the shipment
5. Send shipment confirmation email

However, it would be much better to model this business process in a single service method, which takes a request parameter that has flexible criteria: Order number, Customer id, Order total, Carrier, Tracking code, Shipment date. Internally the method would perform the above workflow.

**Thinking in terms of services**

It can be different to think of in terms of services. It might help to consider the service as a desk clerk in an office. On the desk is an inbox tray and an outbox tray. People fill out specific forms depending on the tasks they need done for them then they drop the forms in the inbox tray. When the clerk has time, they take a form and perform the task using the criteria in the form. When they're done they may or may not place another form in the outbox for delivery back to the person who requested the task.

In this way, you can see that a service is a course-grained black box with varying levels of performance. Clients usually would want to make as few trips to the desk in order to complete their task as possible. Different clients might use the same service with different criteria to achieve their unique needs..

## *Adding a new service API - Retrieve order history*

We will step through adding a fake service API to retrieve a user's order history by their user ID and store ID. The business need in this case is to display a user's order history in a portal channel. In reality, we would probably model a generic getOrders method however we will make a more specific method in this tutorial. We will be working in the commerceServicesWS project, which you can check out and build according to the usual methods described in the section 1 - Setting up your development environment (ie. run ant eclipse-setup-all, ant libsetup, etc)

### Create the message and domain data objects

We will have to add a method to retrieve an order history according to **some** criteria. It is good practice to provide a typed parameter that is specific to the request. In this case, let's create a message object called OrderHistoryRequest. We will have to annotate this object with JAXB annotations so that it can be marshaled to and from XML, as follows:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "OrderHistoryRequest", propOrder = { "userId", "storeCode" })
```

```
public class OrderHistoryRequest {

        @XmlElement(name = "UserId", required = true)
        private String userId;

        @XmlElement(name = "StoreCode", required = true)
        private String storeCode;

        // getters and setters follow

}
```

Our method will return an OrderHistory domain data object, which also needs JAXB annotations:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "OrderHistory", propOrder = { "orders" })
@XmlRootElement(name = "OrderHistory")
public class OrderHistory {

        @XmlElement(name = "Orders")
        private Orders orders;

        public OrderHistory () {
                // no-arg constructor
        }

        public OrderHistory(final List<com.bea.alcs.domain.order.Order> orders) {
                this.orders = new Orders(orders);
        }

        public List<Order> getOrders() {
                return orders;
        }

        public void setOrders(final Orders orders) {
                this.orders = orders;
        }
```

```
}
```

You can see that our OrderHistory will reuse an existing domain data object, Orders. In general, reuse like this is desireable as it keeps our catalog of exported XML schemas to a minimum.

Also, OrderHistory has a constructor that takes a List of domain Order objects. This is a typical pattern for translating internal domain objects into service proxy objects and is the fundamental bit of logic that the proxy objects perform, and which we will test.

## Create tests for the domain data objects

We will create JUnit tests for the translation of domain objects into proxy objects. It is good practice to consider edge cases such as null inputs. An example test method might look like:

```
public void testCreateWithOrders() throws Exception {
                        Date date = new Date();
                        com.bea.alcs.domain.order.Order order1 = OrderTest.createOrder("1234",
OrderStatus.APPROVED, new BigDecimal("00.00"), date);
                        com.bea.alcs.domain.order.Order order2 = OrderTest.createOrder("5678",
OrderStatus.APPROVED, new BigDecimal("20.00"), date,
                                                "Hungry Hungry Hippo", "Pong");
                        domainOrders.add(order1);
                        domainOrders.add(order2);
                        OrderHistory orderHistory = new OrderHistory(domainOrders);
                        assertEquals(2, orderHistory.getOrders().size());
                        OrderTest.verifyOrder(orderHistory.getOrders().get(0), "1234",
OrderStatus.APPROVED, new BigDecimal("00.00"), date);
                        OrderTest.verifyOrder(orderHistory.getOrders().get(1), "5678",
OrderStatus.APPROVED, new BigDecimal("20.00"), date, "Hungry Hungry Hippo", "Pong");
            }
```

You'll notice that we are reusing some creation and assertion methods from the OrderTest. Have a look at the other existing unit tests to get an understanding of what we are trying to test in the domain data objects. Run the test in Eclipse and verify that it passes.

## Create the service interface

Locate the relevant service interface; as of this writing, it is OrderWebService. Add a method to that service: getOrderHistory(OrderHistoryRequest), with the required JAXB annotations as follows:

```
@WebMethod(operationName = "getOrderHistory")
```

```
        OrderHistory getOrderHistory(@WebParam(name = "orderHistoryRequest")
OrderHistoryRequest request);
```

## Create the service implementation

Once the method has been added to the service interface, it will need to be implemented in the service implementation class. Add an implementation as follows:

```
public OrderHistory getOrderHistory(final OrderHistoryRequest request) {
                        CustomerSession session =
customerSessionService.findByCustomerIdAndStoreCode(request.getUserId(),
request.getStoreCode());
                        if (session == null) {
                                return new OrderHistory();
                        }
                        List<com.bea.alcs.domain.order.Order> orders =
orderService.findOrderByCustomerGuid(session.getCustomer().getGuid(), true);
                        return new OrderHistory(orders);
        }
```

In this example of the service implementation, customerSessionService and orderService have been injected by Spring. In our test of the service implementation, they will be injected manually as mock objects and set up using the jMock framework.

## Create a test for the service implementation

Locate the unit test for the service implementation and add the following test case:

```
public void testGetOrderHistory() throws Exception {
                        List<com.bea.alcs.domain.order.Order> domainOrders = new
ArrayList<com.bea.alcs.domain.order.Order>();
                        domainOrders.add(OrderTest.createOrder("1111",
OrderStatus.APPROVED, new BigDecimal("10.00"), new Date(), "Red Big-Wheel"));


                        mockOrderService = mock(OrderService.class);


        mockOrderService.stubs().method("findOrderByCustomerGuid").will(returnValue(domai
nOrders));
                        service.setOrderService((OrderService) mockOrderService.proxy());


                        OrderHistoryRequest request = new OrderHistoryRequest();
                        request.setUserId("user@localhost");
                        request.setStoreCode("SNAPITUP");
                        OrderHistory orderHistory = service.getOrderHistory(request);
```

```
        assertEquals(1, orderHistory.getOrders().size());
        Order order = orderHistory.getOrders().get(0);
        assertEquals("10.00", order.getTotal());
        // etc...
    }
```

Run the test inside Eclipse and verify that it passes.

### Create client acceptance tests

Now that we have generated a WSDL and the client libraries, we can write client acceptance tests, which will invoke the web service that is deployed and running on the app server. As of this writing, we don't have a good framework for writing these tests and because they hit the database, anyone who runs the tests would need to have specific data populated or they will fail. However, until a framework is in place, we can still write tests to validate our services. Take a look at the test, GetOrdersTest in src/src-client-gen/test/java for examples of such tests and write a test case for the new service api as follows:

```
public void testGetOrderHistory() throws Exception {
        OrderHistoryRequest request = new OrderHistoryRequest();
        request.setUserId("user@localhost");
        request.setStoreCode("SNAPITUP");
        OrderWebServiceOperations port = service.getOrderWebServiceImplPort();
        OrderHistory orderHistory = port.getOrderHistory(request);
        assertEquals(2, orderHistory.getOrders().size());
    }
```

You can see that this test assumes that your database contains a store with code "SNAPITUP", a user with id "user@localhost" and that the user has placed 2 orders. You will need to populate that data, probably in the storefront and Commerce Manager applications as it is difficult to do directly in the database.

## *Notes for real-world development*

- Our new service method used the core OrderService. That service is injected into the web service by Spring, and is configured in WEB-INF/conf/spring/service/serviceConnect.xml. If you need to add new core services, you'll need to configure Spring.
- You might be required to create a new web service altogether, which will have its own WSDL. This has implications on the build, spring bean and servlet configurations:
    - A new build file will be needed and the main build file, build.xml will need to have various targets updated to call the new build file. See build_shoppingcart.xml for an example of a service buildfile.

    - You will need to modify WEB-INF/conf/spring/service/serviceConnect.xml to have a bean and jax-ws service binding for your new service. Specify the dependent beans you want Spring to inject into your service.
    - You will also need to modify WEB-INF/web.xml.vm and include new servlet and servlet-mapping elements for your web service.

- When running the ant target to build the web services, it is not always easy to detect why a particular task might have failed. One tip is to attempt to start your app server after a build failure. Usually an exception will be reported during startup if some aspect of your web service configuration is wrong (i.e. missing annotations) and the exceptions are generally verbose enough to help you correct the problem.

### *References*

**JAX-WS**
Java API for XML Web Services (JAX-WS) makes it easy to develop Web services and clients through ease-of-development features, support for W3C and WS-I standards such as SOAP and WSDL, asynchronous client and server, and databinding through JAXB 2.0.

- reference implementation - https://jax-ws.dev.java.net/
- JAX-WS specification - http://jcp.org/aboutJava/communityprocess/final/jsr224/index.html

**JAXB**
Java Architecture for XML Binding (JAXB) provides ease of development by binding an XML document to JavaBean objects based on the XML document's XML schema.

- Reference implementation - https://jaxb.dev.java.net/
- Introduction to the technology - http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

**jMock**
jMock is a library that supports test-driven development of Java code with mock objects. Mock objects help you design and test the interactions between the objects in your programs.

- jMock website - http://www.jmock.org

# Unit Testing

AquaLogic Commerce Services code in the core project is extensively unit tested using the JUnit testing framework. This test coverage provides a quick and effective way to

detect bugs that may have been introduced while making modifications. When customizing AquaLogic Commerce Services, it is good practice to write JUnit tests for new or modified code and run all JUnit tests prior to committing changes. The majority of the JUnit tests in AquaLogic Commerce Services cover code in the com.bea.alcs.core project, which the web projects depend on. Within the core project, test coverage by line is approximately 70%. Domain model classes in the com.bea.alcs.domain package are particularly well covered.

## *Running JUnit tests*

JUnit tests can be run via an ant task or from within the Eclipse development environment.

### Running tests with WebLogic WorkSpace Studio

JUnit tests can also be run from within Eclipse. To run an individual test case, right-click on the file in the Package Explorer and select Run as... > JUnit Test. You can also run all tests in an entire project by right-clicking on the project in the Package Explorer and selecting Run As... > Junit Test. JUnit tests run from Eclipse will run several times faster than the ant task, but several unit tests that pass when run by ant will fail in Eclipse. This is due to a JUnit configuration issue and the tests cases will be updated in an upcoming release of AquaLogic Commerce Services so that they will pass using the default Eclipse JUnit configuration.

**Setting memory options in Eclipse**

The default memory for launching applications in Eclipse is insufficient for executing all unit tests in the core project. You can increase the default memory settings by navigating to Window > preferences > Java > Installed JREs > Select your JRE > Edit... and set your memory settings in the "Default VM Arguments" input box. The recommended setting is "-Xmx512m".

## *Creating unit tests*

### Adding a new unit test

The JUnit test case for a class is typically the name of the class being tested with "Test" appended to it. Test cases should be in the same package as the class they test so that they have protected access to the class's members. However, to avoid cluttering the production code with unit test classes, unit tests are placed in separate directory structure that mirrors the "WEB-INF/src" source folder but is rooted at "WEB-INF/test".

Within the test case, a constructor is not required unless there is initialization that must be performed only once for the entire test case. A main method that runs the text UI or

Swing UI is also not required as it is not used by the Eclipse JUnit runner the or Ant JUnit task.

Test cases in AquaLogic Commerce Services typically use the setup() method to instantiate the object tested by each test in the test case. The setup() method is also frequently used to configure any mock objects that are not specific to any one test.

## Mocking objects

AquaLogic Commerce Services uses the JMock framework to facilitate testing a single class in isolation. JMock eases the creation of mock objects that can be referenced by the class under test instead of an instance of the real class used in production. This allows the output of the collaborating class to be controlled in so that specific situations can be tested in the class under test. Furthermore, the mock objects can be given expections of which methods will be invoked on it by the class under test. These expections can optionally specify the parameters that should be passed as well as the number of times the method should be invoked. Since JMock integrates with JUnit, the test case will fail if the mock object's expectations are not met.

The following annotated code shows a typical mock object usage pattern. In this example, the Product class is being tested and we use a mock ProductType object to test Product's interaction with ProductType.

```
//Create a mock object to mock a product type
final Mock mockProductType = mock(ProductType.class);

//Pass the mock object to the product class, casted to a ProductType
this.productImpl.setProductType((ProductType) mockProductType.proxy());

//Specify that the product type's getProductAttributeGroup() method must
//be called one time and that it will return to the product the set of
//attribute groups we wish to test the product with (returned by the
//createAttributeGroup() method)
mockProductType.expects(once()).method("getProductAttributeGroup").will(returnValue(createAttributeGroup()));

//Invoke the method being tested, which will interact with the mock object
this.productImpl.performAttributeRelatedOperation();

//Check the results of the operation here
```

# Customizing and Extending AquaLogic Commerce Services Applications

The following sections contain many useful suggestions and best practices for customizing AquaLogic Commerce Services. It is highly recommended that you read these before you start making significant changes.

## *General Customization Practices*

### Making updates simple

In order to avoid having to merge code changes when you receive patches or updates we recommend that you touch as few of the core java classes as possible. The sections that follow contain suggestions for how to go about this.

**Tip**

To avoid confusion you may want to create your extension package structure to mirror the AquaLogic Commerce Services package structure.

**Example:**

AquaLogic Commerce Services class:
com.bea.alcs.domain.customer.impl.CustomerImpl.java

your extending class:
com.yourcompany.domain.customer.impl.YourCustomerImpl.java

### Customizing domain objects

Depending on whether you need to customize a domain object application-wide or just certain instances of that class, say inside of a new service method that has been created, you may wish to choose from one of the options below.

**Tip**
Before customizing a domain object, check to see if it allows for localized attributes to be added in the Commerce Manager. Many of the objects that clients usually wish to extend have already been configured to allow dynamic attribute creation.

#### Extending domain objects

If you need to add properties to some of the domain object classes you can do so by extending them with your own java classes. Any accessor or convenience methods that perform some logic that you need to change can be overriden in your extending class. If you find that you need to make a large amount of changes to a domain object, but wish

to maintain the same interface (recommended), you can create a new implementation of the domain object's interface.

### Wrapping domain objects: the Decorator pattern

To customize a domain object only in certain scenarios while leaving it unchanged in the rest of the application consider using the Decorator pattern. Create a new implimentation of the interface that the domain object uses and add a property to it that references the original domain object instance. Impliment all the property accessors and other methods required by the interface and have them delegate the call to the reference of the domain object that you have wrapped, adding custom logic where needed. In the service/controller method where you need to use this customized wrapper simply create a new instance of the wrapper object and insert the original domain object into it using the setter method you created. Since the interface is identical between the wrapper and the original object it will be indistinguishable to most of the application.

**Please Note**

If you are dealing with large lists of domain objects that you need to wrap then you may want to extend the object instead. Iterating over the list and wrapping each object may cause a slight performance loss. If you only need to wrap a single object or a small collection of objects then using the Decorator pattern to wrap them should be fine.

## Customizing service methods

There are a number of ways to customize service methods depending on what changes in logic are required.

### Spring hooks: method interception & pre/post execution hooks

If the core logic inside a service method does not need to be altered, but additional routines need to be run before or after the method call then consider using the spring AOP framework.

With a few changes to the spring configuration files you can add hooks to an exisiting service method and perform additional logic before or after the method's execution. If you need access to the parameters passed into the service method you can use a method interceptor. After configuring a proxy class in the spring configuration file you can have the method call rerouted to your own method, perform the additional logic desired, and then pass the call along to the original method.

**Example:** method interceptor for validating objects passed in as method paramaters

**service.xml - adding a validation interceptor**

```
<!--
```

```
Note:
the referenced class needs to implement
org.aopalliance.intercept.MethodInterceptor
-->

<bean id="validationInterceptor"
  class="com.bea.alcs.service.interceptor.ValidateInterceptor">
 <property name="defaultBeanValidator">
  <ref bean="defaultBeanValidator" />
 </property>
</bean>
```

**service.xml - adding a validation advisor**

```
<bean id="validationAdvisor"
  class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
 <property name="advice">
  <ref bean="validationInterceptor" />
 </property>
 <property name="patterns">
  <list>
   <value>.*add.*</value>
   <value>.*update.*</value>
  </list>
 </property>
</bean>
```

**service.xml - adding a proxy wrapper**

```
<bean id="customerAjaxController"
  class="org.springframework.aop.framework.ProxyFactoryBean">
 <property name="proxyInterfaces">
  <value>com.bea.alcs.sfweb.ajaxservice.CustomerAjaxController</value>
 </property>
 <property name="interceptorNames">
  <list>
```

```
    <value>validationAdvisor</value>
    <value>customerAjaxControllerTarget</value>
  </list>
 </property>
</bean>
```

### Extending service classes

In order to change the internal logic of a service method you can extend the service class with your own. If you need to change the logic of a significant number of methods in a service class then you can create a new implimentation of the service class's interface. After doing this edit the spring service configuration files to point to your newly created service instead of the original one.

## *Customizing the Storefront*

## Customizing the UI

AquaLogic Commerce Services has been built to allow maximum flexibility in customizing the look and feel of the store front.

If you do not need to customize any core logic then you will probably only need to edit the velocity and CSS files. Velocity itself handles most of the display logic and quite a lot of customization can be done just using the velocity macros. All of the html styling is handled in CSS and can be easily customized by edditing the various CSS files.

## Customizing the checkout process

The default checkout process is handled by a collection of Spring controllers, form beans, and configuration files. In general, the service layer methods should not need to be edited. If, however, they do need to be altered, refer to the previous section for suggestions on how to go about doing this.

### Maintaining the default checkout sequence

If the logical sequence of checkout steps is being maintained then you can probably just copy the default checkout-related Spring controllers into your own package directory and alter them as required. If the UI for the checkout process is being customized as well, the Velocity templates linked to the controllers in the spring configuration files will need to be edited as well. For any controllers that have been copied, you will need to update the Spring URL-mapping configuration file to use your newly created controller classes instead of the default ones.

### Changing the checkout sequence

If you need to change the sequence of steps taken through the checkout process you will have to do a fair bit more work. You still shouldn't have to change the service

methods, but you will need to create your own Velocity templates and Spring controllers. You will also need to edit the Spring URL-mapping configuration file to link all these up.

Regardless of what you need to do, you can probably save a fair bit of time looking at the default checkout Velocity templates and Spring controllers.

# Useful Technology Tutorials and Resources

AquaLogic Commerce Services is built on a large list of open source frameworks. In order to develop on the AquaLogic Commerce Services platform you should be familiar with the core frameworks. This Developer Guide provides some training on these frameworks, but for more detailed information, we recommend the resources below.

## *Core 3rd-party frameworks*

- Velocity: http://jakarta.apache.org/velocity/
- Spring: http://www.springframework.org/
    - including: Spring MVC, ACEGI (security) and Commons Validator integration
    - recommended book: "Spring in Action"
    - Intro to the Spring Framework - May 2005 (Intro level) - http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework

## *Other 3rd-party frameworks*

- ACEGI Security Solution - http://acegisecurity.org/;
- Ant (build tool): http://ant.apache.org/
- Axis (Web Services): http://ws.apache.org/axis/
- Java Advanced Imaging API (JAI): http://java.sun.com/products/java-media/jai/iio.html
    - used for dynamic image resizing
- Jboss Rules (rules engine): http://www.jboss.com/products/rules
- jUnit (unit testing): http://www.junit.org
    - recommended book: "jUnit in Action"
    - also recommend jMock for mock objects: http://www.jmock.org/
- Log4J (logging): http://logging.apache.org/log4j/docs/
- Lucen/Solr (search engine): http://lucene.apache.org/java/docs
    - recommended book: "Lucene in Action"
- Quartz (scheduler): http://www.opensymphony.com/quartz/
- Xdoclet (annotations): http://xdoclet.sourceforge.net/xdoclet/index.html
    - recommended book: "Xdoclet in Action"

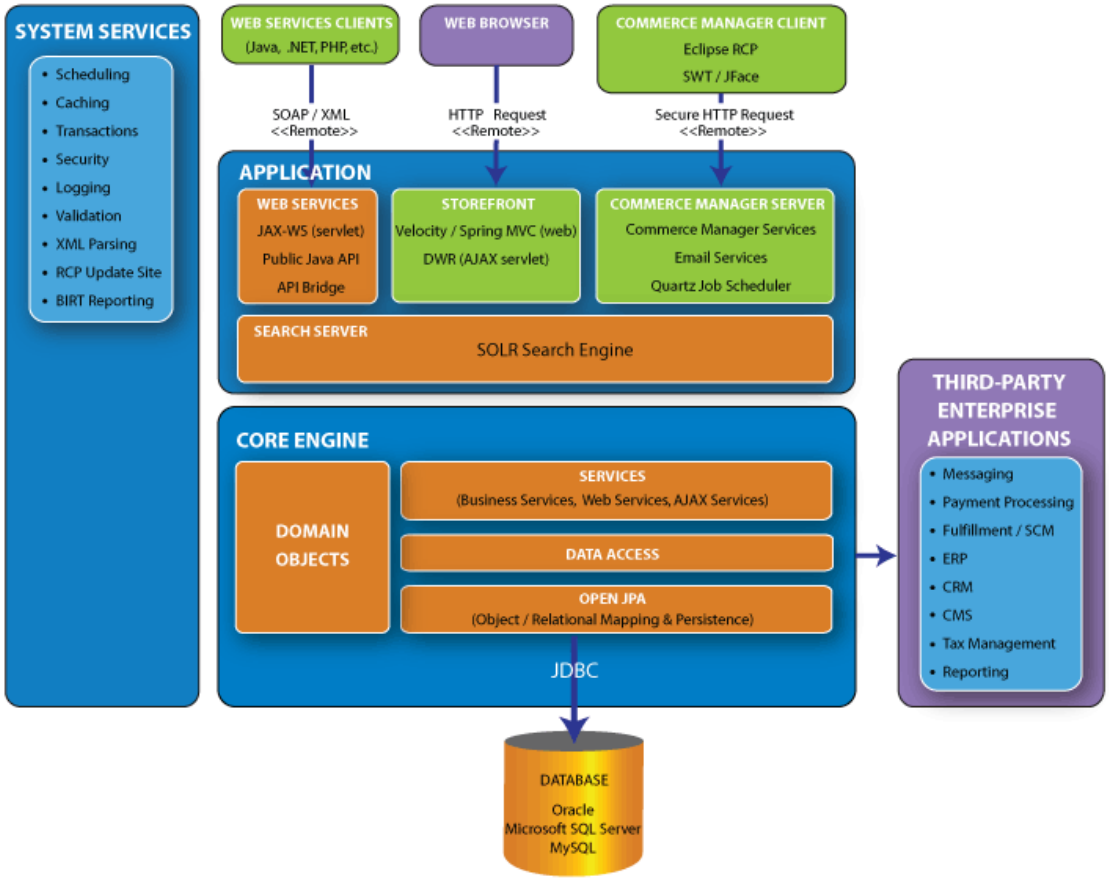
# 3 – Architecture Reference

The purpose of this section is to explain the underlying architectural aspects of AquaLogic Commerce Services that apply to a wide range of features. For a quick introduction to programming with AquaLogic Commerce Services, see the “Programming with AquaLogic Commerce Services” section.

## Application Layers

At a high-level, AquaLogic Commerce Services can be viewed as a set of layers with the data access layer at the bottom and the view layer at the top. This section describes the key design concepts and technologies used in each layer.

The architecture diagram below presents the key components of the system, including the technologies used. The details of these components and how to work with them are described in this section.



## *View*

The view layer is responsible for presenting content to the user in the browser. The view layer also collects user input to be processed by lower layers of the architecture. The view layer is composed of the following technologies.

- Velocity - A template processing technology that embeds information from Java objects within HTML pages
- CSS - Cascading Style Sheets provide formatting and positioning for elements across all HTML pages
- Javascript - Javascript is used to create rich user interfaces in the browser

## Velocity Templating Engine

Velocity is a template engine that serves as an alternative to JSP that separates Java from the presentation tier. When working with Velocity, you will typically begin with a static HTML page and then add Velocity directives (or "code") to access and display the properties of Java objects. Files containing static text and Velocity code are called templates and have a .vm file extension. A Velocity engine is invoked to process the template, rendering pure text without any Velocity code. In addition to the UI layer, Velocity is used in AquaLogic Commerce Services to generate email messages and generate configuration files during the build process.

### Syntax reference

The following subset of Velocity directives demonstrates the key funtionality provided by Velocity and also serves as a quick reference.

- $!customer.address – Displays the address property of the customer object
- $!customer.getAddress() – Displays the result of invoking the getAddress() method on the customer object
- #if [#elseif] [#else] #end – Synax for Conditionals
- #foreach( $ref in arg ) statement #end – Syntax for iteration (For Each loops)
- #set($variable = "value") – Sets the value of a variable. The variable does not need to be declared
- #include - Renders files that are not parsed by Velocity
- #parse - Renders files that are parsed by Velocity
- #macro - Runs a Velocity macro

### Specifying paths and URLs in Velocity templates

The AquaLogic Commerce Services Search Engine Optimization (SEO) feature rewrites the URL of some pages in the StoreFront. For this reason, pages using SEO cannot use paths relative the the page's path. In this case, you must use the absolute path to

images or URLs. By including the #templateInit() directive at the top of your page, a Velocity variable namved $baseUrl will contain the base portion of the absolute path you will need. For example, use syntax below to link to index.ep.

```
<a href="$baseUrl/index.ep">#springMessage("bc.home")</a>
```

### Best practices

- Presentation logic of any significant complexity should be invoked from a Velocity macro
- Velocity templates should be formatted so that they are easy to read
- A Velocity template should consist of mostly HTML sprinkled with Velocity macros and functions
- Use parameters to help abstract logic within a macro
- Templates should NOT contain business logic
- Most macros should be in the global macro library
- Only macros that are highly-specific to a particular page should be defined at the top of the page

There is a flag called "autoload" that determines whether Velocity will automatically reload global library macros. During development, this should be set to "on" so that you do not need to restart your app server when making a change to the global Velocity macro library.

## Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) is a mechanism for separating the style of a web page (fonts, colors, spacing, positioning, etc.) from the content of the page. When using CSS, a CSS file defines the style for various HTML elements such as hypertext links. CSS files also define "classes" which define style information for arbitrary HTML elements. In this case, elements in the HTML declare the name of their CSS class to have that particular style applied to them. CSS class names should refer to what the HTML element is (e.g. Order Summary) rather than suggest the style that should be applied to it.

By using CSS, style information can be changed in a single CSS file and it will automatically "cascade" across all HTML pages on the site.

### CSS in AquaLogic Commerce Services

The main CSS file that defines various styles in AquaLogic Commerce Services is called master.css and can be found in template-resources/stylesheet.

## JavaScript

JavaScript is a cross-platform, object-oriented scripting language. JavaScript is a small, lightweight language; it is not useful as a standalone language, but is designed for easy embedding in other products and applications, such as web browsers. Inside a host environment, JavaScript can be connected to the objects of its environment to provide programmatic control over them.

### JavaScript limitations

- JavaScript has a concept of objects and classes but no built-in concept of inheritance.
- Javascript only has a few built-in types (boolean, float, string, object, array) and it converts from one to another quite freely.
- JavaScript objects (some call them associative arrays) are like maps in a O-O style with map keys of type String.
- Use the following to find out whether a JavaScript object supports a certain property or function:

```
if (typeof (myObj,someProperty) != undefined) {
        ...
    }
if (myObj instanceof MyObj) {
  ...
        }
```

**Note:** The instance instantiated from JSON is always of type Object Array, therefore it does not work well with the "instanceof" operator.

## *Web*

The web layer is responsible for managing the interaction between the view layer and the lower layers of the application. The two main technologies used in the AquaLogic Commerce Services web layer is Spring MVC.

### Spring MVC

Spring MVC is a web framework used to separate Velocity templates in the view layer from the underlying domain model and services. Refer to Spring MVC - Web Framework for more information.

## Spring MVC Web Framework

Spring MVC is a web application framework that performs a similar role to that of Apache Struts. The purpose of Spring MVC is to separate view layer logic from the

underlying domain objects and services to maintain loose coupling and application maintainability.

### Model View Controller (MVC)

The MVC pattern is used to achieve the separation between view and domain. In the Spring MVC layer, the "Model" is the domain objects and services in lower layers of the system. The "View" is the Velocity templates that render information about the model and collect user input. The "Controller" component represents a set of Java classes called controllers that manage the interaction between the View and the Model. At a high level, the intent of the Controllers is to prevent coupling between the domain (and services) and the Velocity that is used to present them.

### Spring MVC Controllers

The Spring MVC Controllers typically manage a single page request and serve one or more of the following roles in displaying the page.

- Specify the Velocity template that will be displayed to satisfy the request.
- Obtain and optionally prepare domain objects for display by the Velocity template.
- Declare a validator to validate field input.
- Invoke methods on domain objects or services as required to satisfy the request.
- Return an appropriate response after performing the required task.

Spring MVC Controllers are declared and configured in the url-mapping.xml file.

- This file declares and configures the controller classes as beans.
- The URL mapping section at the top of the file specifies which controller should be invoked for a given URL.

Spring MVC Controllers should contain only a small amount of logic to initiate operations in lower layers. Work flow logic should appear in services, not in Spring Controllers.

- The problem with implementing workflow logic in controllers is that only Spring MVC has access to it – Web services will not be able to reuse the logic.
- Therefore, even when it doesn't look like there's much need for a service layer call (ie. only a few methods will be executed in the service layer method) logic should still be implemented in the service layer.

### AquaLogic Commerce Services Controller hierarchy

Controllers in AquaLogic Commerce Services will typically inherit from one of the following base classes. The base class to extend depends on the purpose of the controller.

- **SimplePageController** - Use this controller to load a static page that does not invoke any service layer methods such as a "Return Policy" page. There is no need

to implement a new controller class when using SimplePageController, you can just use Spring configuration in url-mapping.xml to create an instance of SimplePageController and specify the Velocity template to be displayed.

- **AbstractEpController** - Extend this controller when your page does not use a form, but you want to perform an action on the service layer or a domain object. For example, a page that changes the user's locale might read a local parameter from the request and change the locale in the user's profile. When extending AbstractEpController, override handleRequestInternal() to implement the logic that is performed when a user visits the page.
- **AbstractEpFormController** - Extend this controller to create pages with HTML forms. For example, consider a "forgotten password" page that requests the user's email address and then emails them a new password. When using AbstractEpFormController, a "form-backing" object is specified as the "command" property in url-mapping.xml. This form-backing object will receive the form input specified by the user. Override onSubmit() to perform the action that occurs when the user submits the form. If your form-backing object requires further preparation before display, you can perform this initialization by overriding formBackingObject() and returning a prepared object.

**Specify a Validator**

If your form input requires validation, specify a validator in the Controller's Spring configuration block. Validation for new form-backing objects can be specified in validation.xml.

## *Service*

The service layer provides services to various consumers in the web layer as well as web service consumers. There are several types of services that serve different roles in the application.

### Persistence Services

Persistence Services provide the capability to save and retrieve domain objects. Persistence Services extend from AbstractEpPersistenceServiceImpl and offer methods for adding new domain objects, retrieving domain objects by their identifier, and searching for domain objects with specific criteria. PersistenceServices are named XService where X is the class name of the objects that it can save and retrieve.

### Domain Services

Domain services typically implement the logic for a use case that is inappropriate for encapsulation by any one domain object. For example, a service that performs a checkout will contain logic for the flow of the interaction between several domain objects. This domain service logic is typically at a higher level of abstraction than the fine-grained domain logic that is implemented by an individual domain object. Domain services will often use other services in combination with domain object logic to accomplish a task.

For example, the checkout service uses domain logic to check for sufficient inventory while using the InventoryService persistence service to persist inventory levels.

> In AquaLogic Commerce Services, domain service logic specific to a particluar domain object is often implemented within the Persistence Service for that object.

### Integration Services

Integration services implement functionality that is invoked by domain services but considered outside the domain of an ecommerce application and typically integrate with other systems or technologies. The following are examples of integration services.

- EmailService - Sends email on behalf of other services.
- CustomerIndexBuildService - Constructs a search index used by the Lucene search feature.
- BirtReportService - Provides access to the BIRT reporting engine.

### System Services

System services handle various concerns that cut across many parts of the application. These services are typically provided by Spring and configured in Spring configuration files. Examples of System services include object lifecyle management, caching, transactions, security, and scheduling.

### Web Services

Web services expose service layer functionality to web services clients. These services are ultimately delivered to external systems via SOAP.

## *Domain*

The Domain layer contains an object model of the ecommerce domain. This object model consists of classes that model real-world entities such as customers and products. The behavior and relationships of these classes should be a reflection of the real-world entities. For example, customers have collections of addresses and products have references to price objects. As much as possible, domain objects should encapsulate their behavior so that the objects who collaborate with them are unaware of internal implementation details. Furthermore, domain objects should be kept relatively free from the constraints of frameworks, persistence, etc. so that they are a relatively pure expression of the domain.

The following sections cover key topics and design considerations in domain model development.

## Domain object inheritance structure

When creating a new domain object, you will need to consider which abstract domain class to inherit from. In most cases, you will need to inherit from one of the leaf nodes of the existing inheritance tree structure: Transient, Entity, or ValueObject. The following interfaces (and their corresponding abstract implementation classes) define the inheritance tree structure.

- **EpDomain** - represents a general domain object with a reference to ElasticPathImpl.
    - **Transient** - Extends EpDomain and represents a transient (not persistent) domain object.
    - **Persistence** - Extends EpDomain and represents a persistent domain object.
        - **Entity** - Extends Persistence and represents a domain object with its own identity.
        - **ValueObject** - Extends Persistence and represents a domain object that is a value with no identity of its own.

## How to create new instances of Domain Objects

In AquaLogic Commerce Services, all new instances are obtained by calling getElasticPath().getBean(BEAN_NAME). All objects are typically "wired" with an instance of ElasticPathImpl and inherit the getElasticPath() method so that this functionality is always available. The BEAN_NAME parameter is a constant bean name defined in ContextIdNames.java.

```
//Create a new instance of Customer
Customer myCustomer = (Customer)
getElasticPath().getBean(ContextIdNames.CUSTOMER);
```

If you are adding a new domain object to the system, you will need to declare the new object in one of the following ways.

### ElasticPathImpl's bean map

In most cases, you will need to add the domain object definition into ElasticPathImpl.PrototypeBeanFactory.BEAN_MAP. This maps bean ID constants from ContextIdNames.java to the fully qualified class name so that ElasticPathImpl can create an instance of the object when clients call getBean(BEAN_NAME). Add new bean definitions in the static block near the top of ElasticPathImpl using an entry such as the one below.

```
//Add a new bean that can be create by and retrieved from ElasticPathImpl
addBeanDefinition(ContextIdNames.CUSTOMER,
  "com.bea.alcs.domain.customer.impl.CustomerImpl");
```

This is the preferred method of declaring new objects because the performance overhead of creating instances is 10% - 15% less than when using Spring.

### Spring configuration

In some cases, you will need to declare multiple domain objects that are initialized with different property values. In this case add the bean definition using standard Spring configuration to domainModel.xml or domainModelCM.xml if the domain model object will only be accessed by the Commerce Manager. Note that this way of declaring instances is used for all singleton service objects.

## Domain object identity - UIDPK vs GUID

Domain objects in AquaLogic Commerce Services have two kinds of identifiers, a UIDPK and a GUID.

The UIDPK is a surrogate key which is generated automatically when a record is added to a table. After it is created, its value cannot be changed. In database tables, UID_PK is a unique primary key.

GUID is the acronym for Globally Unique IDentifier. In AquaLogic Commerce Services it is used as the general name for the identifier of an entity object. In most cases, the GUID is the natury key of an entity object. For example, the natural key of ProductSku is its SKU code, so the SKU code is the GUID for ProductSku objects. In other words, you can think of GUID as a generic name for identifier of entity object. However, some specific entities may have their own name for the same identifier, such as "SKU code."

The following table provides more comparison between the UIDPK and the GUID.

| | UIDPK | GUID | COMMENTS |
|---|---|---|---|
| TYPE | Integer | String | |
| LENGTH | 32 bit or 64 bit | 255 byte (maximum) | |
| SCOPE | One system | Multiple systems | The same product might have different UIDPK in the staging and production database, but they will always have the same GUID. |
| USAGE | Entity object or value object | Entity object | UIDPK is used to identify an entity or a value object in one system. It's also used in associations(foreign key, etc.) between entities and value objects.<br>GUID is only used to identify an entity. It can be used from in CRUD(create, retrieve, update & delete) operations on an entity from other systems (e.g. web services and the import manager). |

| | UIDPK | GUID | COMMENTS |
|---|---|---|---|
| GENERATION | Automatically | Hybrid | You can call the setGuid() method to manually set a GUID. If you don't manually set one, the GUID is to assigned when you create a new entity. The default behavior is to allow the GUID to be automatically assigned. Unlike the UIDPK, a GUID can be changed after creation. |
| ALIASES | N/A | Can have aliases for different entity objects | Examples:<br>ORDER GUID is also called ORDER NUMBER<br>SKU GUID is also called SKU CODE<br>PRODUCT GUID is also called PRODUCT CODE<br>CATEGORY GUID is also called CATEGORY CODE<br>ATTRIUBTE GUID is also called ATTRIBUTE KEY |

## Bi-Directional Relationships

Avoid creating bi-directional relationships between parent objects and the child objects that they aggregate using a collection class. By avoiding bi-directional relationships we eliminate the complexity of maintaining the parent link when a child is added or removed from the collection. In some cases the bi-directional relationship cannot be avoided. For example, ProductSku references Product because it must fall back to the product's price when a client requests a price from the SKU but no price has been defined at the SKU level.

## Domain Object Serialization

Generally, domain objects should be made serializable because they might be replicated from one application server to another in a clustered application server environment. To make a domain object serializable, it must implement the "Serializable" interface and all of its aggregated fields must either be serializable or transient. When you declare a field reference to a service or a utility in a domain object, they should be defined as transient.

```
public class BrandFilterImpl extends AbstractTransientImpl implements BrandFilter {
  private static final String ERROR_MSG = "Invalid brand filter id:";

  private String filterId;
  private int uid;
  private transient Utility utility;
  private transient BrandService brandService;
  private Brand brand;

  ...
}
```

## Setting default field values for domain objects

There are three ways to set default values for domain object fields.

- Set values in the domain object constructor - This technique is seldom used because the field initialization can no longer be controlled.
- Field initializer declaration - This may be used for fields whose default values are cheap to create.
- Set values in the setDefaultValues() method - This is the preferred technique.

Using setDefaultValues() is preferred because it can be used to control when default values are initialized. In production, it is wasteful to set expensive default values when creating new domain objects because they will typically be overwritten by the persistence layer immediately. For example, Maps consume a lot of memory while computing fields like GUIDs and dates are CPU intensive. When running JUnit tests, however, we will need to set the default values so that the functionality can be tested without throwing NullPointerExceptions.

When setting default values in setDefaultValues(), check that the value has not yet been set before initializing fields.

```
public void setDefaultValues() {
  super.setDefaultValues();
  if (this.startDate == null) {
    this.startDate = new Date();
  }
  if (productCategories == null) {
    productCategories = new HashSet();
  }
  if (productPrices == null) {
    productPrices = new HashSet();
  }
  if (promotionPrices == null) {
    promotionPrices = new HashMap();
  }
  if (localeDependantFieldsMap == null) {
    localeDependantFieldsMap = new HashMap();
  }
  if (productSkus == null) {
    productSkus = new HashMap();
  }
```

```
}
```

### *Data Access*

The data access layer is responsible for saving and retrieving data from persistent storage. The majority of persistent data in AquaLogic Commerce Services is stored in the database using the OpenJPA implementation of the Java Persistence API (JPA). A small number of configuration files are persisted directly to the file system using XML and properties files. Objects that are aware of persistence implementation details such as file formats or whether data exists in a database are called Data Access Objects (DAO).

### *OpenJPA object/relational persistence*

OpenJPA is a persistence service that maps objects to tables in a relational database. See the Java Persistence API (JPA) Guide section below for more information.

### *File system persistence*

Two types of data files are used by the file system persistence objects in AquaLogic Commerce Services, XML files and properties files.

#### XML file persistence

AquaLogic Commerce Services uses a file called bea.alcs.xml as the main source of configuration settings. This file is read by ElasticPathDaoXmlFileImpl, which uses the JDOM library to parse the file and store the configuration settings in the ElasticPathImpl singleton. This operation occurs once at system startup and the settings in bea.alcs.xml cannot be changed at runtime.

#### Properties file persistence

Some configuration data is stored in a small number of properties files. These files are stored in conf/resources and are read into the system by PropertiesDaoImpl. Examples of data stored in properties files include the list of countries and country codes, and the status of Lucene index builds.

## Java Persistence API (JPA) Guide

This section is a guide for configuring persistence using the Java Persistence API (JPA). We have chosen to use the Apache OpenJPA implementation.

**Package renaming applied to OpenJPA source**

As a workaround for a classloading conflict issue with WebLogic 10.0 which embeds its own version of OpenJPA, it was necessary to rename the packages of the OpenJPA library from the usual org.apache.openjpa.* to org.apache.renamed.openjpa.*. It was also necessary to rename the `persistence.xml` file to `persistence-renamed.xml`.

## *A little bit about JPA*

Originally part of the JSR-220 Enterprise JavaBeans 3.0 specification, JPA was separated from EJB 2.0 to be a standalone specification for Java persistence. There are many successful implementations of JPA, including

- Apache OpenJPA
- BEA Kodo
- TopLink JPA
- JPOX 1.2
- Hibernate EntityManager

Features of JPA include

- Supports a POJO (Plain Old Java Object) persistence model.
- Rich inheritance support.
- Support for annotations or XML based mapping files.
- Supports pluggable persistence providers.
- Supported by Spring
- Supports native SQL as well as the Java Persistence Query Language (JPQL)

## *Online documentation*

As a starting point, you should take a look at the following documentation

- OpenJPA User's Guide which is a great reference for all aspects of JPA.
- Java Persistence API docs
- OpenJPA API docs.
- Spring Object Relational Mapping guide - JPA section.
- Spring JPA integration API docs

## *Entities and Identities*

### Entity Characteristics

Any application defined object with the following characteristics can be an Entity:

- It can be made persistent
- It has a persistent identity (i.e. a unique key)
- It is not a primitive, a primitive wrapper of build-in object.

So in most cases a concrete or an abstract class can be an Entity.

Mixed inheritance is supported - the superclass and/or subclass of an Entity does not have to be an Entity.

## Entity Identification

All Entities must have a persistent id (i.e. a database primary key). This is known as the entity identity (or persistence identity) JPA supports

- Single field identity (like our long UidPk field)
- Composite primary keys (provided by an Identity class)

The Entity Identity must be defined on the root Entity or mapped superclass of the hierarchy. In our domain model, we have the UidPk identity defined on the Persistence interface. Persistence classes need to implement the getUidPk() method with the @Id annotation and setUidPk() method.

JPA supports several strategies for primary key generation:

- Auto - leave the decision up to the JPA implementation.
- Identity - the database will assign an identity value on insert
- Sequence - use a datastore sequence to generate a value
- Table - use a sequence table to generate a field value

Table is the generator used in our object model, as this will work with all databases, and offers the best overall performance. In our persistence classes we define the generator table JPA_GENERATED_KEYS with columns ID and LAST_VALUE, with a name and primary key value matching the name of the DB table that the class will be persisted to. JPA uses this table to keep track of the last identifier value used to ensure the next one used is unique. You can configure the number of values to allocate in memory for each trip to the database (default is 50). Allocating values in memory allows the JPA runtime to avoid accessing the database for every sequence request.

The annotations on the getUidPk() method are as follows:

```
@Id
@Column(name = "UIDPK")
@GeneratedValue(strategy = GenerationType.TABLE, generator = TABLE_NAME)
@TableGenerator(name = TABLE_NAME, table = "JPA_GENERATED_KEYS",
        pkColumnName = "ID", valueColumnName = "LAST_VALUE", pkColumnValue =
TABLE_NAME)
```

## *Packaging*

JPA requires one XML file META-INF/persistence-renamed.xml which

- Defines the name of the persistence unit
- Defines the transaction strategy
- Identifies the entities contained within a persistence unit
- Defines persistence provider configuration
- It can also define the location of XML mapping files

Here is a sample META-INF/persistence-renamed.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
         http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
             version="1.0">

  <persistence-unit name="myopenjpa">

    <provider>org.apache.openjpa.persistence.PersistenceProviderImpl</provider>

    <class>com.bea.alcs.domain.catalog.impl.AbstractPriceImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.BrandImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.CategoryImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.CategoryTypeImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.InventoryImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.ProductAssociationImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.ProductCategoryImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.ProductImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.ProductSkuImpl</class>
    <class>com.bea.alcs.domain.catalog.impl.ProductTypeImpl</class>

    ...

    <properties>
      <property name="openjpa.ConnectionFactoryName"
                value="java:comp/env/jdbc/epjndi"/>
```

```
    </properties>

  </persistence-unit>

</persistence>
```

This example defines a persistence unit named "myopenjpa", identifies the provider as OpenJPA (org.apache.openjpa.persistence.PersistenceProviderImpl, lists the classes contained in the persistence unit, and sets the connection factory to java:comp/env/jdbc/epjndi.

## *OpenJPA Properties*

In our standard persistence unit (META-INF/persistence-renamed.xml) we set the following OpenJPA properties:

```
<properties>
  <property name="openjpa.Log" value="commons"/>
  <property name="openjpa.ConnectionFactoryProperties" value="PrettyPrint=true,
PrettyPrintLineLength=120"/>
  <property name="openjpa.jdbc.EagerFetchMode" value="parallel"/>
  <property name="openjpa.jdbc.SubclassFetchMode" value="parallel"/>
  <property name="openjpa.QueryCompilationCache" value="all"/>
  <property name="openjpa.DetachState" value="loaded(DetachedStateField=true)"/>
  <property name="openjpa.Multithreaded" value="true"/>
</properties>
```

The following table gives a brief summary of the given settings:

| Property | Description |
|---|---|
| Log | "commons" tells OpenJPA to use the Apache Jakarta Commons Logging thin library for issuing log messages. This is used in our system as a wrapper around log4j |
| ConnectionFactoryProperties | This is used to format the log output of SQL debug messages to a more readable form |
| EagerFetchMode<br>SubclassFetchMode | Set to "parallel" which was found to be the most efficient eager fetch mode for our domain model |
| QueryCompilationCache | Tells OpenJPA to cache parsed query strings. As a result, most queries are only parsed once in OpenJPA, and cached thereafter |

| | |
|---|---|
| DetachState | Tell OpenJPA to take advantage of a detached state field to make the attach process more efficient. This field is added by the enhancer and is not visible to your application. The loaded value tells OpenJPA to detach all fields and relations that are already loaded, but don't include unloaded fields in the detached graph. Setting DetachedStateField to true tells OpenJPA to use a non-transient detached state field so that objects crossing serialization barriers can still be attached efficiently. This requires, however, that your client tier have the enhanced versions of your classes and the OpenJPA libraries. |
| Multithreaded | This tells OpenJPA that persistent instances and OpenJPA components other than the EntityManagerFactory will be accessed by multiple threads at once |

## *Class Enhancement*

In order to provide optimal runtime performance, flexible lazy loading, and efficient, immediate dirty tracking, OpenJPA uses an enhancer. An enhancer is a tool that automatically adds code to your persistent classes after you have written them. The enhancer post-processes the bytecode generated by your Java compiler, adding the necessary fields and methods to implement the required persistence features. This bytecode modification perfectly preserves the line numbers in stack traces and is compatible with Java debuggers.

The OpenJPA enhancer is easily added to the build process as an Ant task. Typically the ant task will look something like this:

```
<taskdef name="openjpac"
        classname="org.apache.openjpa.ant.PCEnhancerTask"
        classpathref="classpath">
</taskdef>


<target name="enhance" description="Enhance classes..">
  <openjpac>
    <config propertiesFile="${src.main.java.dir}/META-INF/persistence-renamed.xml"/>
    <classpath refid="classpath"/>
    <fileset dir="${target.classes.main.dir}">
      <include name="**/*.class"/>
    </fileset>
```

```
  </openjpac>
</target>
```

## *Annotations*

### Overview

Most annotations are placed directly into the class files of the objects to be persisted. This makes it easy to maintain consistency between the objects and their persistence definition - there is no need to go find the "mapping file". Annotations are generally quicker to type than XML mapping definitions and IDE's like Eclipse support annotations including syntax checking and auto-completion. The exception to the rule is annotations that define named queries (see below).

Annotations can be at the field level (persistence layer writes directly to the field) or at the property level (persistence layer uses the getter and setter). You cannot mix field level and property level annotations within the same class hierarchy.

As a standard our object model uses property level annotations. So the annotations in the class files are written above the getter method for each property.

**Read the full documentation**

The OpenJPA User's Guide (available at http://openjpa.apache.org/docs/latest/manual/manual.pdf) contains comprehensive documentation on all of the annotations available along with what parameters they take. You **will** need to refer to this often when doing your own mappings.

### Simple annotations

Here's a list of some of the more common simple annotations:

| Annotation | Purpose |
|---|---|
| @Entity | denotes an entity class |
| @Table(name = "*tablename*") | denotes that the entity should persist to the *tablename* table in the schema |
| @Id | denotes a simple identity field |
| @Basic | denotes a simple value that should be persisted as-is |
| @Column(name = | denotes that the value should be persisted to the |

| | |
|---|---|
| "*columnname*") | *columnname* column in the schema |
| @Temporal | defines how to use Date fields at the JDBC level |
| @Enumerated | control how Enum fields are mapped |
| @Transient | specifies that a field is non-persistent |

Note that properties in a class defined as an Entity that do not have any annotations may still be treated as persistable depending on the type. To avoid any confusion, ensure you annotate **all** property getters including @Basic and @Transient properties!

## Relationship management

Here's a list of some of the annotations commonly used for mapping relationships in JPA

| Annotation | Purpose |
|---|---|
| @OneToOne | When an entity A references a single entity B, and no other A's can reference the same B, we say there is a one to one relation between A and B |
| @OneToMany | When an entity A references multiple B entities, and no two A's reference the same B, we say there is a one to many relation from A to B |
| @ManyToOne | When an entity A references a single entity B, and other A's might also reference the same B, we say there is a many to one relation from A to B |
| @ManyToMany | When an entity A references multiple B entities, and other A's might reference some of the same B's, we say there is a many to many relation between A and B |
| @Embedded | Embedded fields are mapped as part of the datastore record of the declaring entity. A class can be marked as embeddable by adding the @Embeddable annotation to the class |

## Inheritance

JPA provides two annotations for supporting inheritance

| Annotation | Purpose |
|---|---|
| @MappedSuperclass | a non-entity class that can define persistent state and mapping information for entity subclasses. Mapped superclasses are usually abstract. Unlike true entities, you cannot query a mapped superclass, pass a mapped |

| | |
|---|---|
| | superclass instance to any EntityManager or Query methods, or declare a persistent relation with a mapped superclass target |
| @Inheritance | Used to indicate the strategy for a hierarchy of entities. There are 3 strategies to chose from<br>• SINGLE_TABLE - maps all classes in the hierarchy to the base class' table<br>• JOINED - uses a different table for each class in the hierarchy. Each table only includes state declared in its class. Thus to load a subclass instance, the JPA implementation must read from the subclass table as well as the table of each ancestor class, up to the base entity class.<br>• TABLE_PER_CLASS - uses a different table for each class in the hierarchy. Unlike the JOINED strategy, however, each table includes all state for an instance of the corresponding class. Thus to load a subclass instance, the JPA implementation must only read from the subclass table; it does not need to join to superclass tables. |

## OpenJPA specific annotations

OpenJPA provides some useful annotations in addition to those provided by the JPA specs. Many of these were required in our system due to the complexity of the object domain model. An overview of the OpenJPA specific annotations commonly used are as follows:

| Annotation | Purpose |
|---|---|
| @ForeignKey | Foreign key definition. It is important this annotation is present when there is a DB foreign key so that OpenJPA can calculate the correct order to issue database statements without violating key constraints. |
| @Dependent | Marks a direct relation as dependent. This means the referenced object is deleted whenever the owning object is deleted, or whenever the relation is severed by nulling or resetting the owning field |
| @ElementJoinColumn | Array, collection, or map element join column |
| @ElementForeignKey | Define a foreign key constraint to the columns of a collection element |

| | |
|---|---|
| @ElementDependent | Marks the entity elements of a collection, array, or map field as dependent. This means the referenced object is deleted whenever the owning object is deleted, or whenever the relation is severed by nulling or resetting the owning field |
| @Factory | Contains the name of a method that will be invoked to instantiate the field from the external form stored in the database. |
| @Externalizer | Sets the name of a method that will be invoked to convert the field into its external form for database storage |

## Example

Here is an example of an annotated class file which shows many of the annotation types in use

```
@Entity
@Table(name = "TPRODUCT")
public class ProductImpl extends AbstractEntityImpl implements Product {


 ...


 @Basic
 @Column(name = "START_DATE")
 public Date getStartDate() {
  return this.startDate;
 }


 @Version
 @Column(name = "LAST_MODIFIED_DATE")
 @Temporal(TemporalType.TIMESTAMP)
 public Date getLastModifiedDate() {
  return lastModifiedDate;
 }


 @ManyToOne(targetEntity = ProductTypeImpl.class)
 @JoinColumn(name = "PRODUCT_TYPE_UID")
 public ProductType getProductType() {
  return this.productType;
```

```
  }

  @Transient
  public ProductSku getDefaultSku() {
    if (defaultSku == null && productSkus != null && productSkus.size() > 0) {
      return (ProductSku) productSkus.values().iterator().next();
    }
    return defaultSku;
  }

  @OneToMany(targetEntity = ProductPriceImpl.class,
          cascade = { CascadeType.PERSIST, CascadeType.REMOVE })
  @MapKey(name = "currencyCode")
  @ElementJoinColumn(name = "PRODUCT_UID")
  public Map getProductPrices() {
    return productPrices;
  }

  ...

}
```

## *Java Persistence Query Language (JPQL)*

### JPQL overview

JPQL executes over the abstract persistence schema (the entities you've created) defined by your persistence unit, rather than over the database like traditional SQL.

JPQL supports a SQL like syntax

```
SELECT [PD:<result>]
   [PD:FROM <candidate-class(es)>]
   [PD:WHERE <filter>]
   [PD:GROUP BY <grouping>]
   [PD:HAVING <having>]
   [PD:ORDER BY <ordering>]
```

JPQL supports dynamic and static (named) queries

## Query basics

A simple query for all CategoryTypeImpl entities

```
SELECT ct FROM CategoryTypeImpl ct
```

The optional where clause places criteria on matching results. For example

```
SELECT j FROM ImportJobImpl j WHERE j.name = '01-SnapItUp'
```

And of course you can specify parameters, for example

```
SELECT c FROM CategoryImpl c WHERE c.code = ?1
```

## Fetch joins

JPQL queries may specify one or more join fetch declarations, which allow the query to specify which fields in the returned instances will be pre-fetched

```
SELECT ct FROM CategoryTypeImpl ct JOIN FETCH ct.categoryAttributeGroupAttributes
```

Multiple fields may be specified in separate join fetch declarations. You may want to use other join types depending on the data, e.g. left outer join fetch.

## Named queries

JPA supports the concept of named queries. While named queries can be defined in the annotations of a class (usually the class in which the named query is most relevant), they can also be defined external to the implementation class source files in XML files. In this case, each Java package has its own named queries XML file in META-INF/, entitled <packagename>-orm.xml. This is useful for keeping all the named queries specific to a package in the same place while allowing easier extensibility, so that queries can be modified without requiring the application to be recompiled.

Here is an example of named queries in an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
   <package>com.bea.alcs.domain.customer.impl</package>
   <entity class="CustomerDeletedImpl">
      <named-query name="CUSTOMER_UIDS_SELECT_BY_DELETED_DATE">
         <query>select pd.customerUid from CustomerDeletedImpl as pd where pd.deletedDate
&amp;gt;= ?1</query>
      </named-query>
   </entity>
```

```
    <entity class="CustomerGroupImpl">
      <named-query name="CUSTOMERGROUP_SELECT_ALL">
        <query>select cg from CustomerGroupImpl cg</query>
      </named-query>
      <named-query name="CUSTOMERGROUP_FIND_BY_NAME">
        <query>select cg from CustomerGroupImpl cg where cg.name = ?1</query>
      </named-query>
    </entity>
</entity-mappings>
```

## *Persistence Coding*

### Entity Manager

One of the key interfaces in the Java Persistence API is the EntityManager. This is similar to Hibernate's Session interface, providing services for CRUD operations (persist, remove, find, merge etc), creating Query instances and interfacing to the Transaction API.

### Spring integration

Spring provides JPA integration, implementing JPA Container responsibilities. It includes beans like LocalEntityManagerFactoryBean which make it easy to configure the Entity Manager Factory through Spring configuration files, and SharedEntityManagerBean which provides a shared EntityManager reference.

### Core Persistence model integration

JPA has been integrated with the AquaLogic Commerce Services core persistence model, which makes considerable use of Spring Inversion of Control and Dependency Injection.

Here is a summary of the core classes that you'll use for persistence. You can find these in the com.bea.alcs.persistence.impl package

| Class | Implements core interface | Purpose | How it works |
|---|---|---|---|
| JpaPersistenceEngineImpl | PersistenceEngine | The main persistence engine class. This is what you would set as the persistenceEngine property in your Spring configuration files for the | This class uses a shared EntityManager provided by the Spring configuration |

| | | | |
|---|---|---|---|
| | | service beans | |
| JpaSessionImpl | PersistenceSession | This is used by code such as the Import Manager which uses the PersistenceSession interface to run queries on the session in a transaction under its control | This class uses the EntityManager methods directly |
| JpaQueryImpl | Query | This is used by code such as the Import Manager which uses the Query interface to manipulate queries within its own transaction | This class uses the javax.persistence.Query methods directly |
| JpaTransactionImpl | Transaction | This is used by code such as the Import Manager which uses the Transaction interface control its own transactions | This class uses the EntityTransaction methods directly |

The majority of the time you'll just use the core PersistenceEngine interface, and configure Spring to inject the JPA implementation, so your code generally doesn't need to know anything about JPA!

## *Eager vs. Lazy Loading*

You can define fields to load lazily or eagerly in the annotations, but be careful! Eager loading everything can seriously affect performance, and is unnecessary in cases when you don't need every linked object.

Lazy loading will defer loading of a field's associate entity until it is accessed - but it must be accessed while the connection is still open. Trying to access that field after the connection is open will get often you a NullPointerException.

Best practice is to define all (non-basic) fields as lazy loading (the default for OneToMany and ManyToMany mappings) and then define named queries with fetch join to eagerly load the relation fields. When you need just the basic data you use a query without the fetch join and when you need everything you use the fetch join query.

For example look at the following named queries

```
@NamedQuery(name = "CATEGORY_SELECT_BY_GUID",
        query = "SELECT c FROM CategoryImpl c WHERE c.code = ?1")
```

and

```
@NamedQuery(name = "CATEGORY_SELECT_BY_GUID_EAGER",
        query = "SELECT c FROM CategoryImpl c "
            + "LEFT OUTER JOIN FETCH c.localeDependantFieldsMap "
            + "LEFT OUTER JOIN FETCH c.attributeValueMap "
            + "WHERE c.code = ?1")
```

You would use the first one when you just want the basic Category information, and the second when you need all the locale dependent fields and attributes.

## *Fetch Groups*

OpenJPA provides the concept of Fetch Groups - sets of fields that load together. They can be used to pool together associated fields in order to provide performance improvements over standard data fetching. Specifying fetch groups allows for tuning of lazy loading and eager fetching behavior. Fetch groups are used in our system for objects that are indexed by our Search Server to define exactly what fields need to be loaded by the indexing jobs.

## *Problems and work-arounds*

JPA is designed to be simple and easy to use, and so it is less complex than Hibernate. Sometimes this means that mappings that were possible in Hibernate are not possible to do in JPA without some modifications to the object model.

### Locale and Currency

JPA does not support the Locale and Currency types so some special OpenJPA annotations are required when using these datatypes.

For example, here are annotated getters for Locale and Currency fields:

```
@Persistent(optional = false)
  @Externalizer("toString")
  @Factory("org.apache.commons.lang.LocaleUtils.toLocale")
```

```
@Column(name = "LOCALE", length = 20)
public Locale getLocale() {
  return locale;
}

@Persistent
@Column(name = "CURRENCY", length = 3)
@Externalizer("getCurrencyCode")
@Factory("com.bea.alcs.commons.util.impl.UtilityImpl.currencyFromString")
public Currency getCurrency() {
  return currency;
}
```

This tells JPA the externalizer method to use to convert the field into its external form for database storage (eg getCurrencyCode()) for Currency), and to the full class and method to instantiate the field from the external form stored in the database.

## Constructors

JPA requires all entities to have a default constructor which it will call when the persistence layer starts up. If there is no default constructor the build-time enhancer will add one for you.

Where you need to be careful is if you have code in your constructor that is expecting other objects to have already been instantiated. For example, this code was originally in the constructor for ShoppingCartImpl

```
public ShoppingCartImpl() {
  super();
  this.shippingServiceLevelList = new ArrayList();
  this.viewHistory = (ViewHistory) getElasticPath().getBean("viewHistory");
}
```

Before this class was annotated as a JPA Entity the constructor wasn't being called until the ElasticPathImpl bean factory added a bean for ShoppingCartImpl, so the call to getElasticPath() would always work. However, once this class was added to the list of persistence entities, the constructor was being called **before** ElasticPathImpl was being instantiated, so the call to getElasticPath() would fail!

The work around is to put any code in the constructor that relies on other objects in a more appropriate place. In this particular case the code to get the viewHistory bean was moved into the getViewHistory method of ShoppingCartImpl.

## *Useful Tools*

OpenJPA includes some useful tools that can help with using JPA.

### Mapping Tool

The mapping tool can generate, modify or validate a schema from your object model. It is very useful for checking if your annotations map the object(s) to the current schema, or for generating the SQL if you add a new object.

You can call the tool from ant (a target openjpa.verify-schema is defined in the core module which should verify mappings against the schema, provided your connection details are in persistence.xml), or you can call it from the command line as follows for a particular class

```
java org.apache.openjpa.jdbc.meta.MappingTool -a validate
com.bea.alcs.domain.catalog.impl.ProductImpl.java
```

Naturally, this requires your classpath to be set up correctly and you need to specify the fully qualified class name.

### Schema Tool

The schema tool can be used to generate an XML representation of a database schema, take an XML schema definition and apply changes to a database to make it match the XML, or reverse map to generate class definitions and metadata for an existing schema.

See the OpenJPA documentation for full details.

## *Logging*

Detailed logging for JPA can be enabled by adding the following to your log4j.properties.vm files

```
log4j.category.openjpa.jdbc.SQL=TRACE
log4j.category.openjpa.Runtime=TRACE
log4j.category.openjpa.MetaData=TRACE
log4j.category.openjpa.Enhance=TRACE
log4j.category.openjpa.Query=TRACE
```

These values should be removed or set to ${ep_log_level} for distribution

## *Required libraries*

The following libraries are required for our use of JPA. They can all be found in EP_LIBS

| Library | Purpose |
|---|---|

| | |
|---|---|
| openjpa-all.1.0.0-renamed.jar | Main Apache OpenJPA API (with renamed packages) |
| geronimo-jpa_3.0_spec-1.0.jar | JPA specifications API |
| geronimo-jms_1.1_spec-1.0.1.jar | JMS specifications API |
| geronimo-jta_1.1_spec-1.1.jar | JTA specifications API |
| serp-serp-1.13.1.jar | Framework for manipulating Java bytecode |
| commons-collections-3.2.jar | Newer release of the commons collections API |
| commons-pool-1.3.jar | Object pooling components |

## Database Compatibility Issues

This page lists database compatibility issues and other database-specific considerations.

### Oracle

#### Empty strings are considered null

Oracle considers the empty String ("") to be a null value. If you assign the empty string to a column that is set to not-null, the database operation will fail.

#### Query Length Limitation

Oracle allows only 1000 expressions in a single query. Therefore, large queries with over 1000 expressions like "select ... from ... where ... in (..., ...)" must be separated into multiple queries. The following code from ProductServiceImpl.findByUids() shows the recommended way to accomplish this.

```
final List queries = this.getUtility().
  composeQueries(Criteria.PRODUCT_BY_UIDS,
  Criteria.PLACE_HOLDER_FOR_LIST, productUids);
final List result = getPersistenceEngine().retrieve(queries);
```

#### Ampersand(&) cannot be used in SQL scripts.

```
INSERT INTO TBRAND (UIDPK,CODE,GUID)
VALUES (1110,'D&H', 'D&H');
```

must be changed to the following.

```
INSERT INTO TBRAND (UIDPK,CODE,GUID)
VALUES (1110,concat('D', concat(CHR(38), 'H')),
```

```
    concat('D', concat(CHR(38), 'H')));
```

**Literal date values must use the TO_DATE function**

Literal date values like '2006-11-11 11:11:11' cannot be used in Oracle SQL scripts. Instead, the TO_DATE function must be used as shown below.

```
TO_DATE('2005-06-10','YYYY-MM-DD')
```

## Service-Layer Database Transactions

Database transaction behavior is defined on a per-method basis in the service.xml Spring configuration file. Service classes with methods that are to be run as transactions inherit the parent bean named "txProxyTemplate" in service.xml. txProxyTemplate has a property called "transactionAttributes" that defines the nature of transactions for methods matching particular name patterns. For example, the property <prop key="update*">PROPAGATION_REQUIRED</prop> declares that any method beginning with "update" in a service method of a bean whose parent is "txProxyTemplate" will be run in a transaction. The following transaction attributes are in use:

- PROPAGATION_REQUIRED - Supports a transaction if one already exists. If there is no transaction a new one is started.
- PROPAGATION_NEVER - Does not execute as a transaction and throws an exception if one already exists.
- PROPAGATION_SUPPORTS - A new transaction will not be started to run the method, however if a transaction is in progress it will propagate to include the call to this method as well.

The following attributes are also available:

- PROPAGATION_MANDATORY - Throws an exception if there is no active transaction
- PROPAGATION_NOT_SUPPORTED - Executes non-transactionally and suppends any existing transaction
- PROPAGATION_REQUIRES_NEW - Always starts a new transaction. If an active transaction exists, it is suspended.
- PROPAGATION_NESTED - Runs as a nested transaction if one exists, or starts a new one.

Note that all methods that access the database must have names that match one of the transaction attribute properties defined in txProxyTemplate or an exception will be thrown at runtime.

# Performance and Scalability

This document provides information and guidelines for developing scalable high-performance applications with AquaLogic Commerce Services. We will focus on issues

and considerations that arise during development. For performance issues relating to deployment and configuration, please see Configuring AquaLogic Commerce Services for optimal performance in the Deployment Guide.

The first section covers general performance issues optimizations of AquaLogic Commerce Services. The Storefront and Commerce Manager sections focus on topics that are specific to those applications. Theses sections are followed by a set of performance tips for general Java development.

## *General*

This section covers general performance topics for the AquaLogic Commerce Services applications.

## Database

### Create indexes on condition columns for faster queries

Creating an index on columns containing values used in query conditions can improve the performance of your queries. While this may introduce some overhead when updating data, query performance is usually more important in AquaLogic Commerce Services.

## OpenJPA

See the Apache OpenJPA Optimization Guidelines for recommendations on OpenJPA options that can be configured to help improve performance.

### Denormalize tables to reduce joins

Normalized table designs reduce duplicate data, but this is usually at the expense of performance. Querying normalized data typically requires queries to join several tables, which can be slow. For this reason, judicious denormalization can be beneficial for performance critical query performance. For example, in AquaLogic Commerce Services a column called LOCALIZED_ATTRIBUTE_KEY exists in the table TPRODUCTATTIRIBUTEVALUE to avoid the need to join with TATTRIBUTE to retrieve the key.

### Use the Commerce Manager to perform batch jobs

Statistics and customer profiling data should be calculated as a batch job run by the Commerce Manager rather than be computed on the fly during Storefront transactions. These jobs are typically scheduled to run during times of low load using the Quartz scheduler. The top seller and product recommendation calculations are good examples of intensive computations that are performed as Commerce Manager batch jobs to conserve Storefront CPU resources.

### Use Lucene for faster searches

Use Lucene index search to achieve faster search results. Lucene is particularly well suited for full text search and will perform much faster than a database query.

## Storefront

### Use load tuners to retrieve only the data you need

Some domain objects such as Product aggregate a large amount of data such as prices, SKUs, attributes, inventory and recommendations etc. For a particular page, it's normally not necessary to load all of the aggregated data. For example, in search result page we only show the main information of a product such as its name and price. The attributes and product recommendations are not used. In this case, a load tuner can be used to instruct the persistence layer on which collections should be loaded. The load tuner is typically passed to the service layer along with the id of the desired parent object. There are several pre-defined load tuners declared in domainModel.xml, but you will often wish to create your own tuner to load exactly the data you need.

### Attribute performance overhead

The AquaLogic Commerce Services attribute system provides a flexible way for business users to add or change category or product information. However, developers should be aware of the overhead of using attributes. A database table join and a Java map lookup is required to retrieve an attribute value for a category or product. Therefore, the number of attributes used should be limited in performance-critical situations.

### Only store critically necessary data in the HTTP session

It's always good for scalability and performance to make the HTTP session lighter because thousands of sessions may get created in an application server at peak time. Consider the following calculations as an example.

Session timeout : 30 minutes

Average customer visit time : 5 minutes

Average customer arrival rate : 10 users / second

Average session size : 100Kbyte / session

Total sessions = 10 users / sec * 60 sec / minute

* (30 minutes + 5 minutes) = 21,000 sessions

Total memory consumed by sessions = 100Kbyte / session

* 21,000 sessions = 2.1GB

From the calculations above, we can see how critical it is to limit the data in the session because of its overall effect on memory usage. Furthermore, many of the objects stored in the session are long-lived and have a high chance of being pushed to older segments of JVM memory where they are more expensive to garbage collect. A site's throughput

to drop severely when the available memory is consumed and the JVN begins performing full garbage collections.

When you consider storing additional data in the session, the following questions can help you determine if this is really necessary.

- How much memory will the data consume when multiplied across all sessions?
- Can this data be shared by customers? If not, is it acceptable to have thousands of copies of this data?
- What is the probability that successive requests will use this data?
- If it is not stored in the session, is there a cheap way for successive requests to generate or retrieve this data when they require it?

## Optimize load balancer polls

If you are using a load balancer that makes periodic requests to check on a server's health, this can be a source of performance problems. Each time a request is made, a CustomerSession object is created to track information about the session and potential customer. If the load balancer poll repeatedly makes requests, a high number of unnecessary customer sessions may be created. For this reason, create a specialized page to handle polls without creating any session data.

### Use load tuners to retrieve only the data you need

Some domain objects such as Product aggregate a large amount of data such as prices, SKUs, attributes, inventory and recommendations etc. For a particular page, it's normally not necessary to load all of the aggregated data. For example, in search result page we only show the main information of a product such as its name and price. The attributes and product recommendations are not used. In this case, a load tuner can be used to instruct the persistence layer on which collections should be loaded. The load tuner is typically passed to the service layer along with the id of the desired parent object. There are several pre-defined load tuners declared in domainModel.xml, but you will often wish to create your own tuner to load exactly the data you need.

### Attribute performance overhead

The AquaLogic Commerce Services attribute system provides a flexible way for business users to add or change category or product information. However, developers should be aware of the overhead of using attributes. A database table join and a Java map lookup is required to retrieve an attribute value for a category or product. Therefore, the number of attributes used should be limited in peformance-critical situations.

### Only store critically necessary data in the HTTP session

It's always good for scalability and performance to make the HTTP session lighter because thousands of sessions may get created in an application server at peak time. Consider the following calculations as an example.

Session timeout : 30 minutes

Average customer visit time : 5 minutes

Average customer arrival rate : 10 users / second

Average session size : 100Kbyte / session

Total sessions = 10 users / sec * 60 sec / minute

* (30 minutes + 5 minutes) = 21,000 sessions

Total memory consumed by sessions = 100Kbyte / session

* 21,000 sessions = 2.1GB

From the calculations above, we can see how critical it is to limit the data in the session because of its overall effect on memory usage. Furthermore, many of the objects stored in the session are long-lived and have a high chance of being pushed to older segments of JVM memory where they are more expensive to garbage collect. A site's throughput to drop severely when the available memory is consumed and the JVN begins performing full garbage collections.

When you consider storing additional data in the session, the following questions can help you determine if this is really necessary.

- How much memory will the data consume when multiplied across all sessions?
- Can this data be shared by customers? if not, is it acceptable to have thousands of copies of this data?
- What is the probability that successive requests will use this data?
- If it is not stored in the session, is there a cheap way for successive requests to generate or retrieve this data when they require it?

## Cache clustering

AquaLogic Commerce Services does not use cache clustering. Cache clustering is complex, error prone, and poor performing. Since most objects are cached as read-only in the Storefront, there is no need to cluster the cache.

## Optimize load balancer polls

If you are using a load balancer that makes periodic requests to check on a server's health, this can be a source of performance problems. Each time a request is made, a CustomerSession object is created to track information about the session and potential customer. If the load balancer poll repeatedly makes requests, a high number of unnecessary customer sessions may be created. For this reason, create a specialized page to handle polls without creating any session data.

## Java code performance tips

### Use Collections.EMPTY_LIST, Collections.EMPTY_SET, Collections.EMPTY_MAP

If a method returns a list, set or map, it should return an empty list, set or map rather than null. If the returned list, set or map is just for reference (which should be in most cases), you should consider using Collections.EMPTY_LIST, Collections.EMPTY_SET and Collections.EMPTY_MAP instead of creating a new collection. Theses Collections constants are immutable instances and can be used to avoid unnecessary memory allocation.

```
public Collection getProductUids() {
  if (topSellerProducts == null) {
    return Collections.EMPTY_SET;
  }
  return topSellerProducts.keySet();
}
```

### Map Iteration

Avoid iterating on maps in the way shown below.

```
for (Iterator keyIter = mapToSort.keySet().iterator(); keyIter.hasNext();) {
  String currKey = (String) keyIter.next();
  String currValue = (String) mapToSort.get(currKey);
  reverseMap.put(currValue, currKey);
  sortedValueSet.add(currValue);
}
```

This method accesses the value of a Map entry using a key that was retrieved from a keySet iterator. It is more efficient to use an iterator on the entrySet of the map to avoid the Map.get(key) lookup. Use the following code instead.

```
for (Iterator iter = mapToSort.entrySet().iterator(); iter.hasNext();) {
  final Entry entry = (Entry) iter.next();
  final String currKey = (String) entry.getKey();
  final String currValue = (String) entry.getValue();
    reverseMap.put(currValue, currKey);
  sortedValueSet.add(currValue);
}
```

### Explicit garbage collection

```
Runtime.getRuntime().gc();
```

Explicit requests for garbage collection as shown above are abnormal. They should only be used in benchmarking or profiling code. Explicitly invoking the garbage collector in routines such as close or finalize methods can lead to extremely poor performance. Garbage collection can be expensive and any situation that forces hundreds or thousands of garbage collections will bring a machine to a crawl.

### Concatenating Strings using + in a loop

Concatenating Strings using + is slower than using StringBuffer.
Slow example:

```
for (int i = 0; i < optionValueCodes.size(); i++) {
  query += "optVal.optionValueKey = '" + (String) optionValueCodes.get(i) + "' ";
  if (i < optionValueCodes.size() - 1) {
   query += "or ";
  }
}
```

Better example:

```
final StringBuffer sbf = new StringBuffer("Select sku.uidPk
from ProductSku as sku inner join sku.optionValueMap as optVal where (");


for (int i = 0; i < optionValueCodes.size(); i++) {
  sbf.append("optVal.optionValueKey = '")
     .append((String) optionValueCodes.get(i)).append("' ");
  if (i < optionValueCodes.size() - 1) {
    sbf.append("or ");
  }
}
```

### Use static inner classes when possible

If you have an inner class that does not use its embedded reference to the object which created it, it can be declared static. The reference makes instances of the class larger and may keep the reference to the creator object alive longer than necessary. If possible, the class should be made static.

# Database Reference

This section serves as a reference guide for the AquaLogic Commerce Services database model.

## *Data Model*

Any given database data model displays the relationship among database tables in the schema, for a particular topic. The data models are grouped in the following categories:

- Catalog
- Customer
- Import
- Orders
- Roles
- Rules
- Shipping
- Taxes

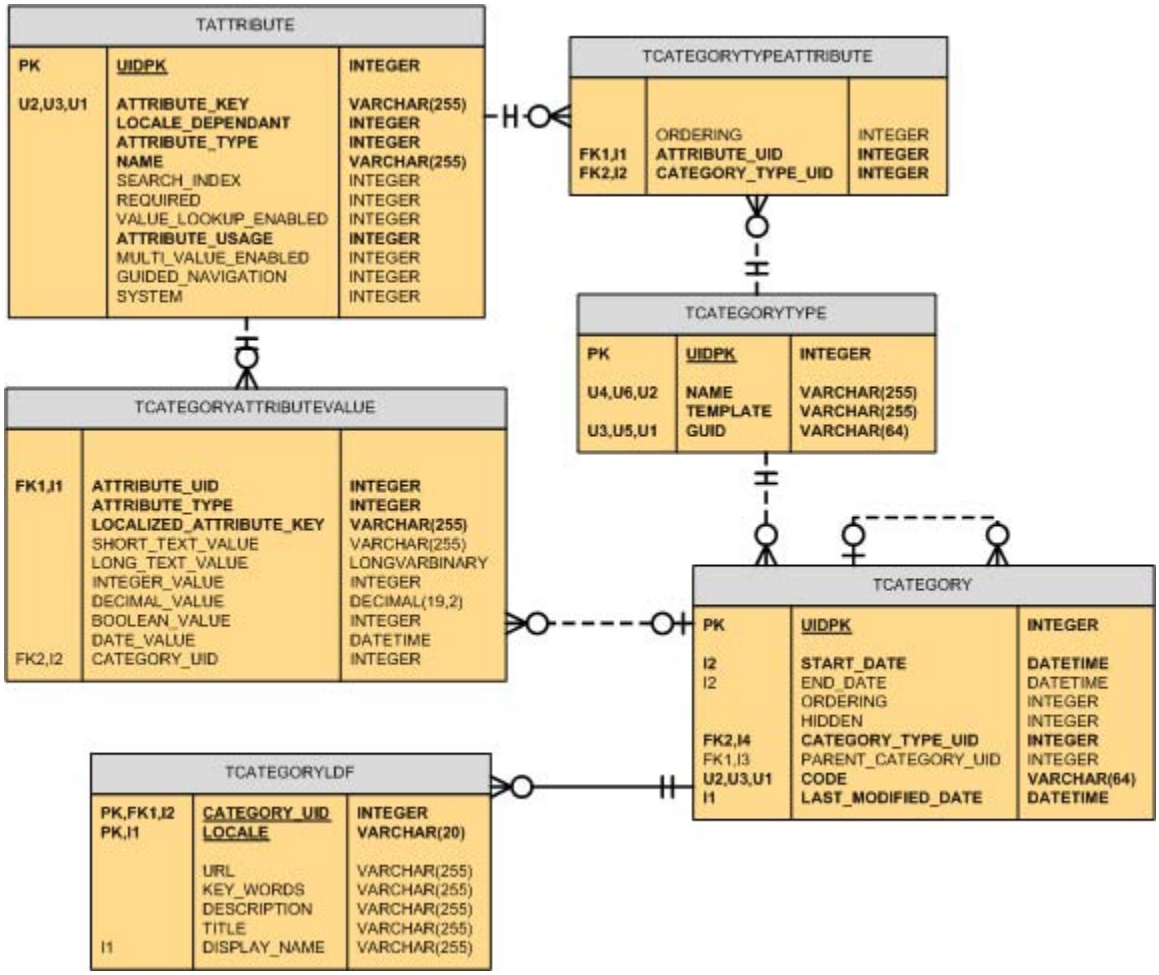
For the larger data model source graphics, see http://edocs.bea.com/alcs/docs51/pdf/alcs_data_model.zip in this document's directory.

## Catalog

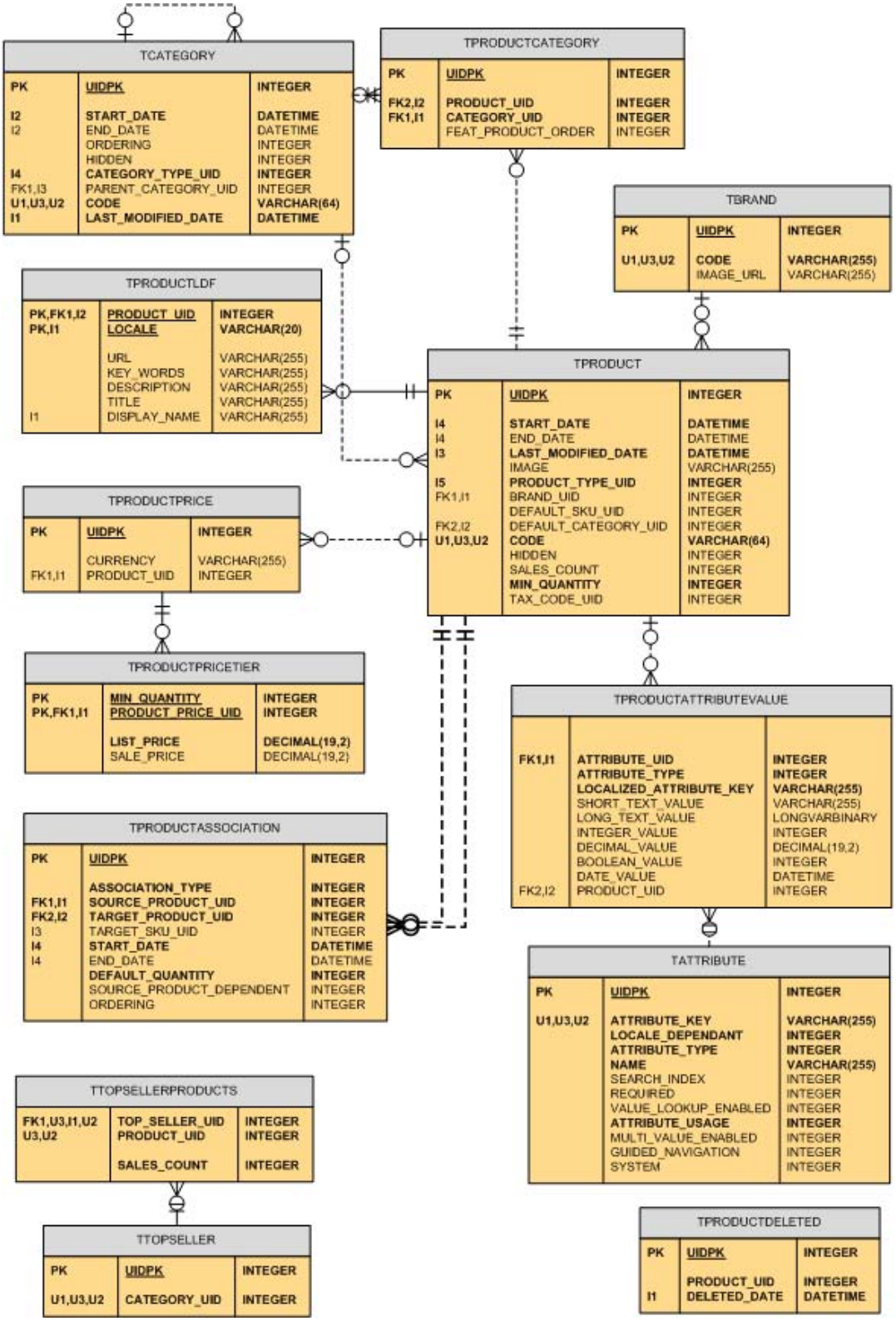
The data model diagrams related to the Catalog subsystem are:

- Category
- Product
- Product SKU

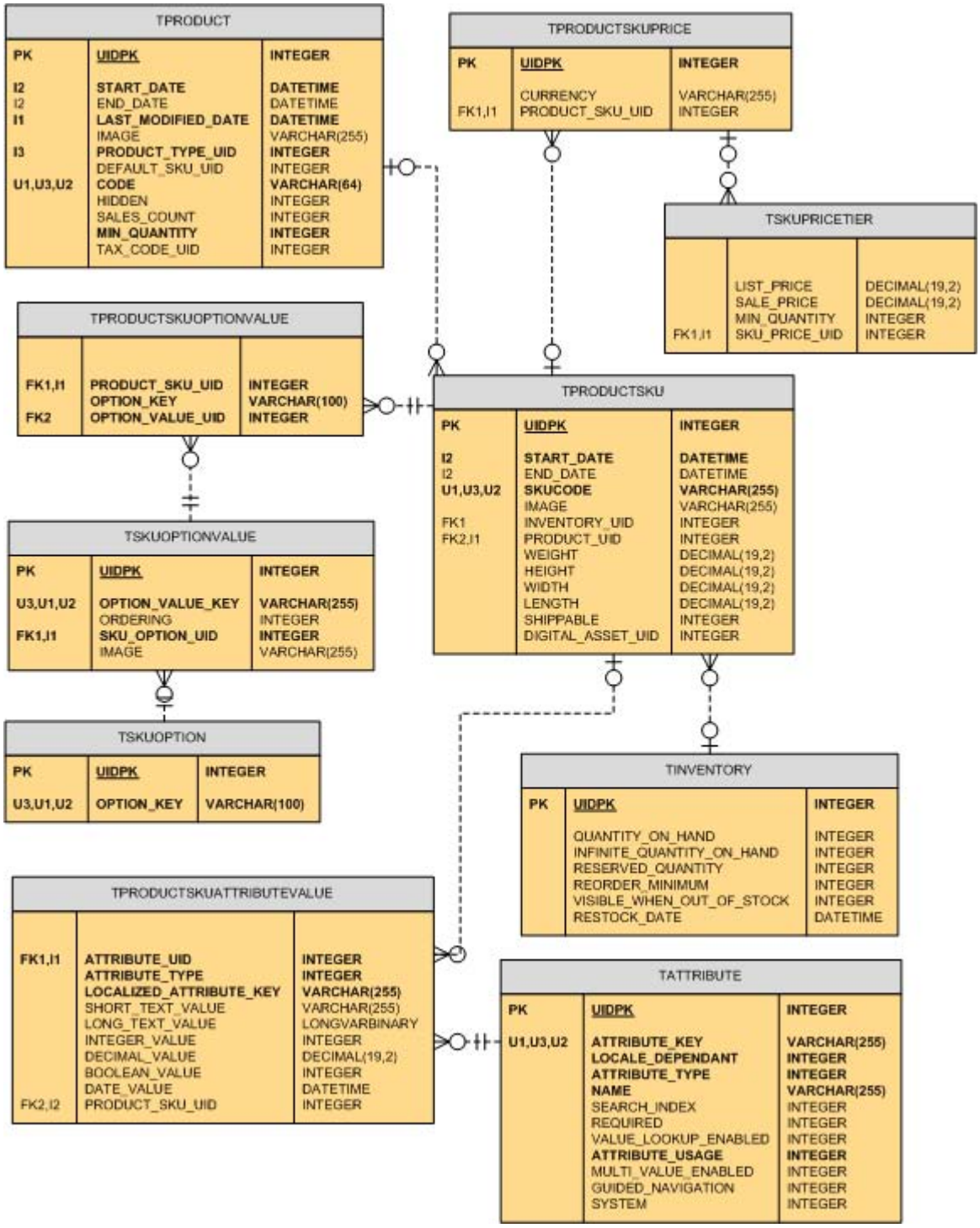
## Category


TATTRIBUTE
PK UIDPK INTEGER
U2,U3,U1 ATTRIBUTE_KEY VARCHAR(255)
LOCALE_DEPENDANT INTEGER
ATTRIBUTE_TYPE INTEGER
NAME VARCHAR(255)
SEARCH_INDEX INTEGER
REQUIRED INTEGER
VALUE_LOOKUP_ENABLED INTEGER
ATTRIBUTE_USAGE INTEGER
MULTI_VALUE_ENABLED INTEGER
GUIDED_NAVIGATION INTEGER
SYSTEM INTEGER
TCATEGORYTYPEATTRIBUTE
ORDERING INTEGER
FK1,I1 ATTRIBUTE_UID INTEGER
FK2,I2 CATEGORY_TYPE_UID INTEGER
TCATEGORYTYPE
PK UIDPK INTEGER
U4,U6,U2 NAME VARCHAR(255)
TEMPLATE VARCHAR(255)
U3,U5,U1 GUID VARCHAR(64)
TCATEGORYATTRIBUTEVALUE
FK1,I1 ATTRIBUTE_UID INTEGER
ATTRIBUTE_TYPE INTEGER
LOCALIZED_ATTRIBUTE_KEY VARCHAR(255)
SHORT_TEXT_VALUE VARCHAR(255)
LONG_TEXT_VALUE LONGVARBINARY
INTEGER_VALUE INTEGER
DECIMAL_VALUE DECIMAL(19,2)
BOOLEAN_VALUE INTEGER
DATE_VALUE DATETIME
FK2,I2 CATEGORY_UID INTEGER
TCATEGORY
PK UIDPK INTEGER
I2 START_DATE DATETIME
I2 END_DATE DATETIME
ORDERING INTEGER
HIDDEN INTEGER
FK2,I4 CATEGORY_TYPE_UID INTEGER
FK1,I3 PARENT_CATEGORY_UID INTEGER
U2,U3,U1 CODE VARCHAR(64)
I1 LAST_MODIFIED_DATE DATETIME
TCATEGORYLDF
PK,FK1,I2 CATEGORY_UID INTEGER
PK,I1 LOCALE VARCHAR(20)
URL VARCHAR(255)
KEY_WORDS VARCHAR(255)
DESCRIPTION VARCHAR(255)
TITLE VARCHAR(255)
I1 DISPLAY_NAME VARCHAR(255)

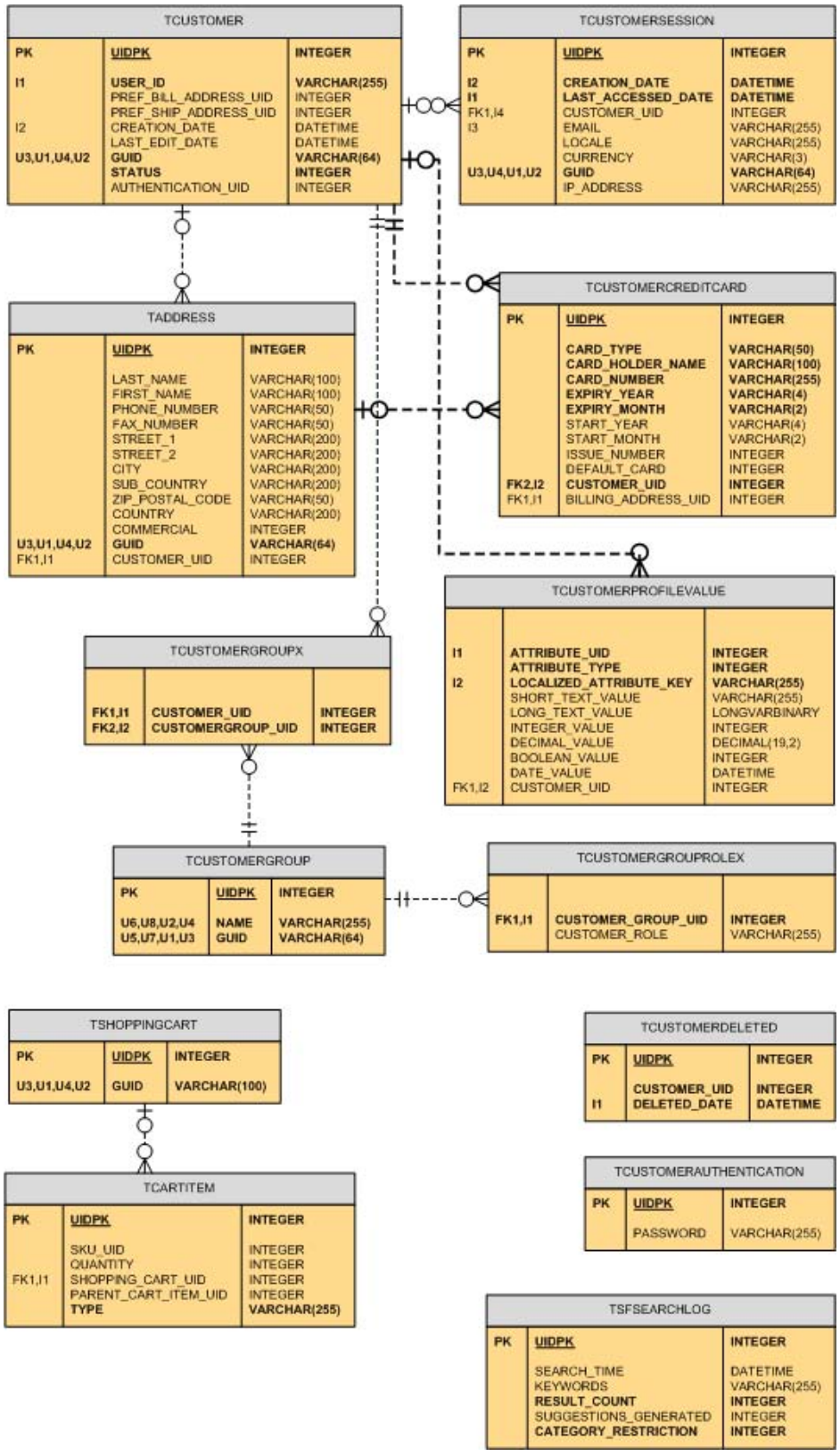
## Product


TCATEGORY
PK UIDPK INTEGER
I2 START_DATE DATETIME
I2 END_DATE DATETIME
ORDERING INTEGER
HIDDEN INTEGER
I4 CATEGORY_TYPE_UID INTEGER
FK1,I3 PARENT_CATEGORY_UID INTEGER
U1,U3,U2 CODE VARCHAR(64)
I1 LAST_MODIFIED_DATE DATETIME
TPRODUCTCATEGORY
PK UIDPK INTEGER
FK2,I2 PRODUCT_UID INTEGER
FK1,I1 CATEGORY_UID INTEGER
FEAT_PRODUCT_ORDER INTEGER
TBRAND
PK UIDPK INTEGER
U1,U3,U2 CODE VARCHAR(255)
IMAGE_URL VARCHAR(255)
TPRODUCTLDF
PK,FK1,I2 PRODUCT_UID INTEGER
PK,I1 LOCALE VARCHAR(20)
URL VARCHAR(255)
KEY_WORDS VARCHAR(255)
DESCRIPTION VARCHAR(255)
TITLE VARCHAR(255)
I1 DISPLAY_NAME VARCHAR(255)
TPRODUCT
PK UIDPK INTEGER
I4 START_DATE DATETIME
I4 END_DATE DATETIME
I3 LAST_MODIFIED_DATE DATETIME
IMAGE VARCHAR(255)
I5 PRODUCT_TYPE_UID INTEGER
FK1,I1 BRAND_UID INTEGER
DEFAULT_SKU_UID INTEGER
FK2,I2 DEFAULT_CATEGORY_UID INTEGER
U1,U3,U2 CODE VARCHAR(64)
HIDDEN INTEGER
SALES_COUNT INTEGER
MIN_QUANTITY INTEGER
TAX_CODE_UID INTEGER
TPRODUCTPRICE
PK UIDPK INTEGER
CURRENCY VARCHAR(255)
FK1,I1 PRODUCT_UID INTEGER
TPRODUCTPRICETIER
PK MIN_QUANTITY INTEGER
PK,FK1,I1 PRODUCT_PRICE_UID INTEGER
LIST_PRICE DECIMAL(19,2)
SALE_PRICE DECIMAL(19,2)
TPRODUCTATTRIBUTEVALUE
FK1,I1 ATTRIBUTE_UID INTEGER
ATTRIBUTE_TYPE INTEGER
LOCALIZED_ATTRIBUTE_KEY VARCHAR(255)
SHORT_TEXT_VALUE VARCHAR(255)
LONG_TEXT_VALUE LONGVARBINARY
INTEGER_VALUE INTEGER
DECIMAL_VALUE DECIMAL(19,2)
BOOLEAN_VALUE INTEGER
DATE_VALUE DATETIME
FK2,I2 PRODUCT_UID INTEGER
TPRODUCTASSOCIATION
PK UIDPK INTEGER
ASSOCIATION_TYPE INTEGER
FK1,I1 SOURCE_PRODUCT_UID INTEGER
FK2,I2 TARGET_PRODUCT_UID INTEGER
I3 TARGET_SKU_UID INTEGER
I4 START_DATE DATETIME
I4 END_DATE DATETIME
DEFAULT_QUANTITY INTEGER
SOURCE_PRODUCT_DEPENDENT INTEGER
ORDERING INTEGER
TATTRIBUTE
PK UIDPK INTEGER
U1,U3,U2 ATTRIBUTE_KEY VARCHAR(255)
LOCALE_DEPENDANT INTEGER
ATTRIBUTE_TYPE INTEGER
NAME VARCHAR(255)
SEARCH_INDEX INTEGER
REQUIRED INTEGER
VALUE_LOOKUP_ENABLED INTEGER
ATTRIBUTE_USAGE INTEGER
MULTI_VALUE_ENABLED INTEGER
GUIDED_NAVIGATION INTEGER
SYSTEM INTEGER
TTOPSELLERPRODUCTS
FK1,U3,I1,U2 TOP_SELLER_UID INTEGER
U3,U2 PRODUCT_UID INTEGER
SALES_COUNT INTEGER
TTOPSELLER
PK UIDPK INTEGER
U1,U3,U2 CATEGORY_UID INTEGER
TPRODUCTDELETED
PK UIDPK INTEGER
PRODUCT_UID INTEGER
I1 DELETED_DATE DATETIME

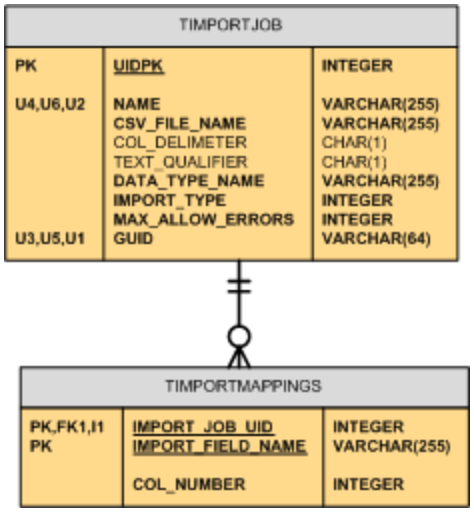
## Product SKU


TPRODUCT
PK UIDPK INTEGER
I2 START_DATE DATETIME
I2 END_DATE DATETIME
I1 LAST_MODIFIED_DATE DATETIME
IMAGE VARCHAR(255)
I3 PRODUCT_TYPE_UID INTEGER
DEFAULT_SKU_UID INTEGER
U1,U3,U2 CODE VARCHAR(64)
HIDDEN INTEGER
SALES_COUNT INTEGER
MIN_QUANTITY INTEGER
TAX_CODE_UID INTEGER
TPRODUCTSKUPRICE
PK UIDPK INTEGER
CURRENCY VARCHAR(255)
FK1,I1 PRODUCT_SKU_UID INTEGER
TSKUPRICETIER
LIST_PRICE DECIMAL(19,2)
SALE_PRICE DECIMAL(19,2)
MIN_QUANTITY INTEGER
FK1,I1 SKU_PRICE_UID INTEGER
TPRODUCTSKUOPTIONVALUE
FK1,I1 PRODUCT_SKU_UID INTEGER
OPTION_KEY VARCHAR(100)
FK2 OPTION_VALUE_UID INTEGER
TPRODUCTSKU
PK UIDPK INTEGER
I2 START_DATE DATETIME
I2 END_DATE DATETIME
U1,U3,U2 SKUCODE VARCHAR(255)
IMAGE VARCHAR(255)
FK1 INVENTORY_UID INTEGER
FK2,I1 PRODUCT_UID INTEGER
WEIGHT DECIMAL(19,2)
HEIGHT DECIMAL(19,2)
WIDTH DECIMAL(19,2)
LENGTH DECIMAL(19,2)
SHIPPABLE INTEGER
DIGITAL_ASSET_UID INTEGER
TSKUOPTIONVALUE
PK UIDPK INTEGER
U3,U1,U2 OPTION_VALUE_KEY VARCHAR(255)
ORDERING INTEGER
FK1,I1 SKU_OPTION_UID INTEGER
IMAGE VARCHAR(255)
TSKUOPTION
PK UIDPK INTEGER
U3,U1,U2 OPTION_KEY VARCHAR(100)
TINVENTORY
PK UIDPK INTEGER
QUANTITY_ON_HAND INTEGER
INFINITE_QUANTITY_ON_HAND INTEGER
RESERVED_QUANTITY INTEGER
REORDER_MINIMUM INTEGER
VISIBLE_WHEN_OUT_OF_STOCK INTEGER
RESTOCK_DATE DATETIME
TPRODUCTSKUATTRIBUTEVALUE
FK1,I1 ATTRIBUTE_UID INTEGER
ATTRIBUTE_TYPE INTEGER
LOCALIZED_ATTRIBUTE_KEY VARCHAR(255)
SHORT_TEXT_VALUE VARCHAR(255)
LONG_TEXT_VALUE LONGVARBINARY
INTEGER_VALUE INTEGER
DECIMAL_VALUE DECIMAL(19,2)
BOOLEAN_VALUE INTEGER
DATE_VALUE DATETIME
FK2,I2 PRODUCT_SKU_UID INTEGER
TATTRIBUTE
PK UIDPK INTEGER
U1,U3,U2 ATTRIBUTE_KEY VARCHAR(255)
LOCALE_DEPENDANT INTEGER
ATTRIBUTE_TYPE INTEGER
NAME VARCHAR(255)
SEARCH_INDEX INTEGER
REQUIRED INTEGER
VALUE_LOOKUP_ENABLED INTEGER
ATTRIBUTE_USAGE INTEGER
MULTI_VALUE_ENABLED INTEGER
GUIDED_NAVIGATION INTEGER
SYSTEM INTEGER

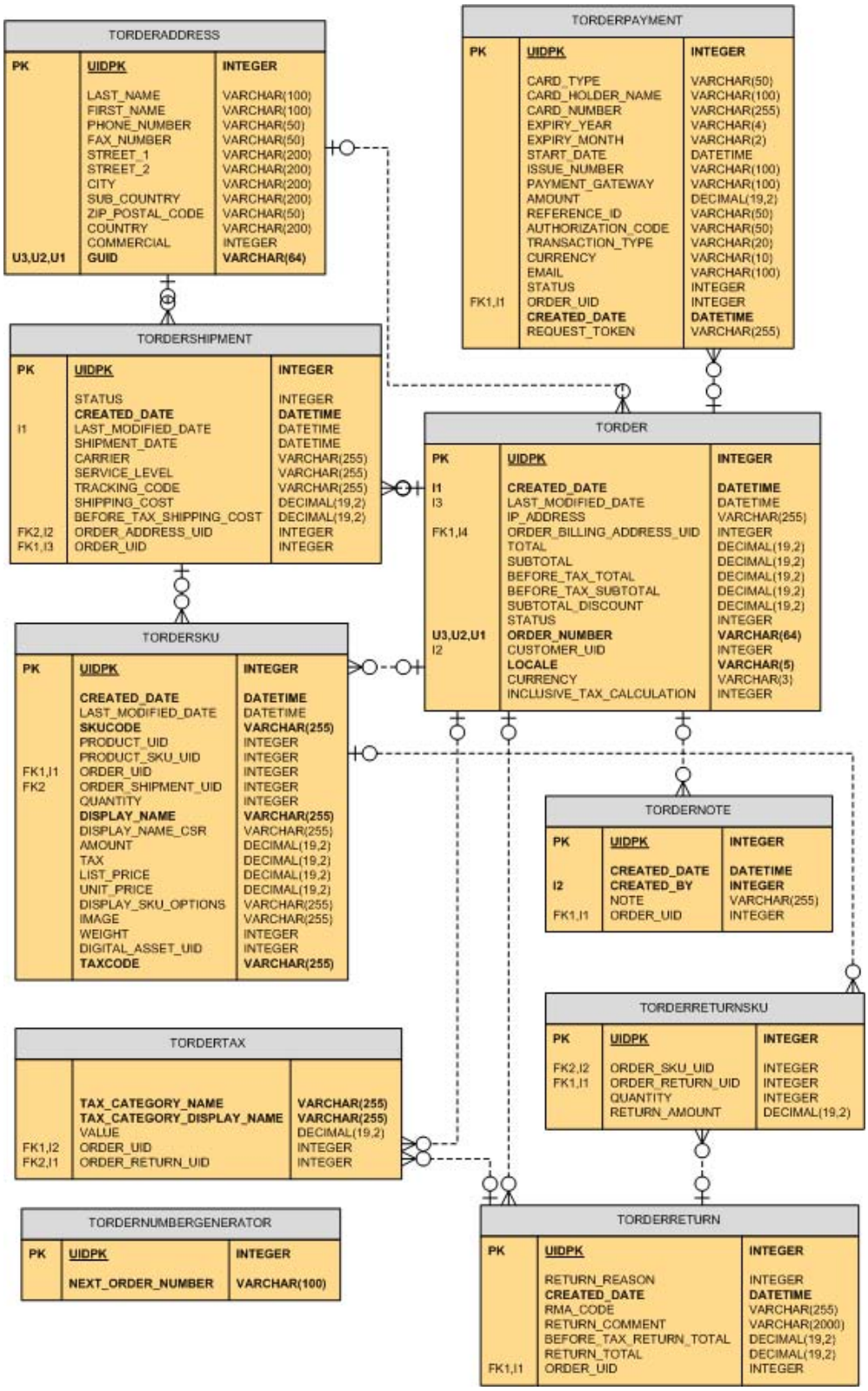
# Customer


TCUSTOMER
PK UIDPK INTEGER
I1 USER_ID VARCHAR(255)
PREF_BILL_ADDRESS_UID INTEGER
PREF_SHIP_ADDRESS_UID INTEGER
I2 CREATION_DATE DATETIME
LAST_EDIT_DATE DATETIME
U3,U1,U4,U2 GUID VARCHAR(64)
STATUS INTEGER
AUTHENTICATION_UID INTEGER
TCUSTOMERSESSION
PK UIDPK INTEGER
I2 CREATION_DATE DATETIME
I1 LAST_ACCESSED_DATE DATETIME
FK1,I4 CUSTOMER_UID INTEGER
I3 EMAIL VARCHAR(255)
LOCALE VARCHAR(255)
CURRENCY VARCHAR(3)
U3,U4,U1,U2 GUID VARCHAR(64)
IP_ADDRESS VARCHAR(255)
TADDRESS
PK UIDPK INTEGER
LAST_NAME VARCHAR(100)
FIRST_NAME VARCHAR(100)
PHONE_NUMBER VARCHAR(50)
FAX_NUMBER VARCHAR(50)
STREET_1 VARCHAR(200)
STREET_2 VARCHAR(200)
CITY VARCHAR(200)
SUB_COUNTRY VARCHAR(200)
ZIP_POSTAL_CODE VARCHAR(50)
COUNTRY VARCHAR(200)
COMMERCIAL INTEGER
U3,U1,U4,U2 GUID VARCHAR(64)
FK1,I1 CUSTOMER_UID INTEGER
TCUSTOMERCREDITCARD
PK UIDPK INTEGER
CARD_TYPE VARCHAR(50)
CARD_HOLDER_NAME VARCHAR(100)
CARD_NUMBER VARCHAR(255)
EXPIRY_YEAR VARCHAR(4)
EXPIRY_MONTH VARCHAR(2)
START_YEAR VARCHAR(4)
START_MONTH VARCHAR(2)
ISSUE_NUMBER INTEGER
DEFAULT_CARD INTEGER
FK2,I2 CUSTOMER_UID INTEGER
FK1,I1 BILLING_ADDRESS_UID INTEGER
TCUSTOMERPROFILEVALUE
I1 ATTRIBUTE_UID INTEGER
ATTRIBUTE_TYPE INTEGER
I2 LOCALIZED_ATTRIBUTE_KEY VARCHAR(255)
SHORT_TEXT_VALUE VARCHAR(255)
LONG_TEXT_VALUE LONGVARBINARY
INTEGER_VALUE INTEGER
DECIMAL_VALUE DECIMAL(19,2)
BOOLEAN_VALUE INTEGER
DATE_VALUE DATETIME
FK1,I2 CUSTOMER_UID INTEGER
TCUSTOMERGROUPX
FK1,I1 CUSTOMER_UID INTEGER
FK2,I2 CUSTOMERGROUP_UID INTEGER
TCUSTOMERGROUP
PK UIDPK INTEGER
U6,U8,U2,U4 NAME VARCHAR(255)
U5,U7,U1,U3 GUID VARCHAR(64)
TCUSTOMERGROUPROLEX
FK1,I1 CUSTOMER_GROUP_UID INTEGER
CUSTOMER_ROLE VARCHAR(255)
TSHOPPINGCART
PK UIDPK INTEGER
U3,U1,U4,U2 GUID VARCHAR(100)
TCARTITEM
PK UIDPK INTEGER
SKU_UID INTEGER
QUANTITY INTEGER
FK1,I1 SHOPPING_CART_UID INTEGER
PARENT_CART_ITEM_UID INTEGER
TYPE VARCHAR(255)
TCUSTOMERDELETED
PK UIDPK INTEGER
CUSTOMER_UID INTEGER
I1 DELETED_DATE DATETIME
TCUSTOMERAUTHENTICATION
PK UIDPK INTEGER
PASSWORD VARCHAR(255)
TSFSEARCHLOG
PK UIDPK INTEGER
SEARCH_TIME DATETIME
KEYWORDS VARCHAR(255)
RESULT_COUNT INTEGER
SUGGESTIONS_GENERATED INTEGER
CATEGORY_RESTRICTION INTEGER

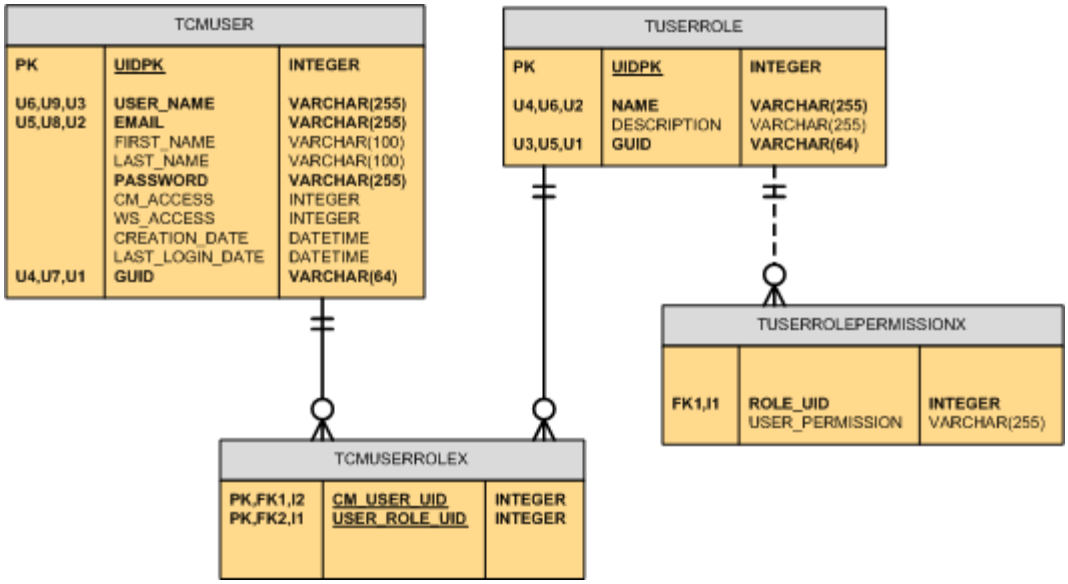
## Import


TIMPORTJOB
PK
UIDPK
INTEGER
U4,U6,U2
NAME
VARCHAR(255)
CSV_FILE_NAME
VARCHAR(255)
COL_DELIMETER
CHAR(1)
TEXT_QUALIFIER
CHAR(1)
DATA_TYPE_NAME
VARCHAR(255)
IMPORT_TYPE
INTEGER
MAX_ALLOW_ERRORS
INTEGER
U3,U5,U1
GUID
VARCHAR(64)
TIMPORTMAPPINGS
PK,FK1,I1
IMPORT_JOB_UID
INTEGER
PK
IMPORT_FIELD_NAME
VARCHAR(255)
COL_NUMBER
INTEGER

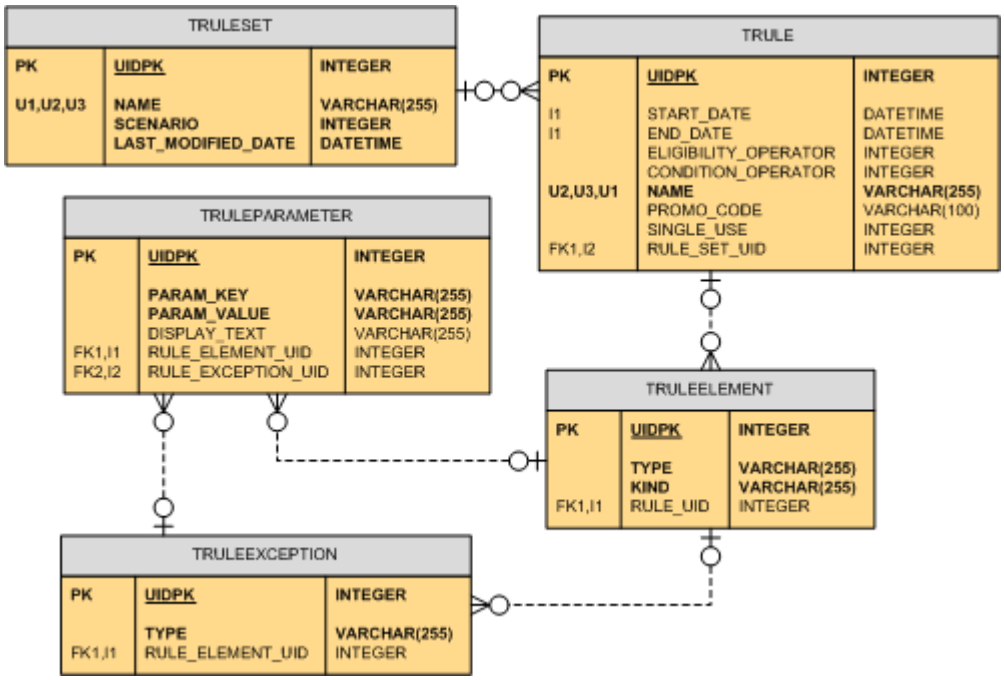
# Orders


TORDERADDRESS
PK UIDPK INTEGER
LAST_NAME VARCHAR(100)
FIRST_NAME VARCHAR(100)
PHONE_NUMBER VARCHAR(50)
FAX_NUMBER VARCHAR(50)
STREET_1 VARCHAR(200)
STREET_2 VARCHAR(200)
CITY VARCHAR(200)
SUB_COUNTRY VARCHAR(200)
ZIP_POSTAL_CODE VARCHAR(50)
COUNTRY VARCHAR(200)
COMMERCIAL INTEGER
U3,U2,U1 GUID VARCHAR(64)
TORDERPAYMENT
PK UIDPK INTEGER
CARD_TYPE VARCHAR(50)
CARD_HOLDER_NAME VARCHAR(100)
CARD_NUMBER VARCHAR(255)
EXPIRY_YEAR VARCHAR(4)
EXPIRY_MONTH VARCHAR(2)
START_DATE DATETIME
ISSUE_NUMBER VARCHAR(100)
PAYMENT_GATEWAY VARCHAR(100)
AMOUNT DECIMAL(19,2)
REFERENCE_ID VARCHAR(50)
AUTHORIZATION_CODE VARCHAR(50)
TRANSACTION_TYPE VARCHAR(20)
CURRENCY VARCHAR(10)
EMAIL VARCHAR(100)
STATUS INTEGER
FK1,I1 ORDER_UID INTEGER
CREATED_DATE DATETIME
REQUEST_TOKEN VARCHAR(255)
TORDERSHIPMENT
PK UIDPK INTEGER
STATUS INTEGER
CREATED_DATE DATETIME
I1 LAST_MODIFIED_DATE DATETIME
SHIPMENT_DATE DATETIME
CARRIER VARCHAR(255)
SERVICE_LEVEL VARCHAR(255)
TRACKING_CODE VARCHAR(255)
SHIPPING_COST DECIMAL(19,2)
BEFORE_TAX_SHIPPING_COST DECIMAL(19,2)
FK2,I2 ORDER_ADDRESS_UID INTEGER
FK1,I3 ORDER_UID INTEGER
TORDER
PK UIDPK INTEGER
I1 CREATED_DATE DATETIME
I3 LAST_MODIFIED_DATE DATETIME
IP_ADDRESS VARCHAR(255)
FK1,I4 ORDER_BILLING_ADDRESS_UID INTEGER
TOTAL DECIMAL(19,2)
SUBTOTAL DECIMAL(19,2)
BEFORE_TAX_TOTAL DECIMAL(19,2)
BEFORE_TAX_SUBTOTAL DECIMAL(19,2)
SUBTOTAL_DISCOUNT DECIMAL(19,2)
STATUS INTEGER
U3,U2,U1 ORDER_NUMBER VARCHAR(64)
I2 CUSTOMER_UID INTEGER
LOCALE VARCHAR(5)
CURRENCY VARCHAR(3)
INCLUSIVE_TAX_CALCULATION INTEGER
TORDERSKU
PK UIDPK INTEGER
CREATED_DATE DATETIME
LAST_MODIFIED_DATE DATETIME
SKUCODE VARCHAR(255)
PRODUCT_UID INTEGER
PRODUCT_SKU_UID INTEGER
FK1,I1 ORDER_UID INTEGER
FK2 ORDER_SHIPMENT_UID INTEGER
QUANTITY INTEGER
DISPLAY_NAME VARCHAR(255)
DISPLAY_NAME_CSR VARCHAR(255)
AMOUNT DECIMAL(19,2)
TAX DECIMAL(19,2)
LIST_PRICE DECIMAL(19,2)
UNIT_PRICE DECIMAL(19,2)
DISPLAY_SKU_OPTIONS VARCHAR(255)
IMAGE VARCHAR(255)
WEIGHT INTEGER
DIGITAL_ASSET_UID INTEGER
TAXCODE VARCHAR(255)
TORDERNOTE
PK UIDPK INTEGER
CREATED_DATE DATETIME
I2 CREATED_BY INTEGER
NOTE VARCHAR(255)
FK1,I1 ORDER_UID INTEGER
TORDERRETURNSKU
PK UIDPK INTEGER
FK2,I2 ORDER_SKU_UID INTEGER
FK1,I1 ORDER_RETURN_UID INTEGER
QUANTITY INTEGER
RETURN_AMOUNT DECIMAL(19,2)
TORDERTAX
TAX_CATEGORY_NAME VARCHAR(255)
TAX_CATEGORY_DISPLAY_NAME VARCHAR(255)
VALUE DECIMAL(19,2)
FK1,I2 ORDER_UID INTEGER
FK2,I1 ORDER_RETURN_UID INTEGER
TORDERNUMBERGENERATOR
PK UIDPK INTEGER
NEXT_ORDER_NUMBER VARCHAR(100)
TORDERRETURN
PK UIDPK INTEGER
RETURN_REASON INTEGER
CREATED_DATE DATETIME
RMA_CODE VARCHAR(255)
RETURN_COMMENT VARCHAR(2000)
BEFORE_TAX_RETURN_TOTAL DECIMAL(19,2)
RETURN_TOTAL DECIMAL(19,2)
FK1,I1 ORDER_UID INTEGER

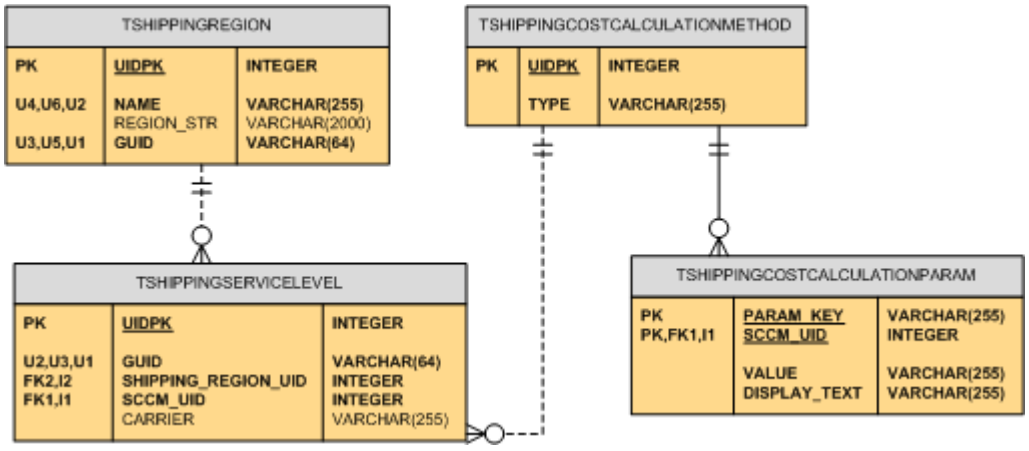
## Roles


TCMUSER
PK
U6,U9,U3
U5,U8,U2
U4,U7,U1
UIDPK
USER_NAME
EMAIL
FIRST_NAME
LAST_NAME
PASSWORD
CM_ACCESS
WS_ACCESS
CREATION_DATE
LAST_LOGIN_DATE
GUID
INTEGER
VARCHAR(255)
VARCHAR(255)
VARCHAR(100)
VARCHAR(100)
VARCHAR(255)
INTEGER
INTEGER
DATETIME
DATETIME
VARCHAR(64)
TUSERROLE
PK
U4,U6,U2
U3,U5,U1
UIDPK
NAME
DESCRIPTION
GUID
INTEGER
VARCHAR(255)
VARCHAR(255)
VARCHAR(64)
TUSERROLEPERMISSIONX
FK1,I1
ROLE_UID
USER_PERMISSION
INTEGER
VARCHAR(255)
TCMUSERROLEX
PK,FK1,I2
PK,FK2,I1
CM_USER_UID
USER_ROLE_UID
INTEGER
INTEGER

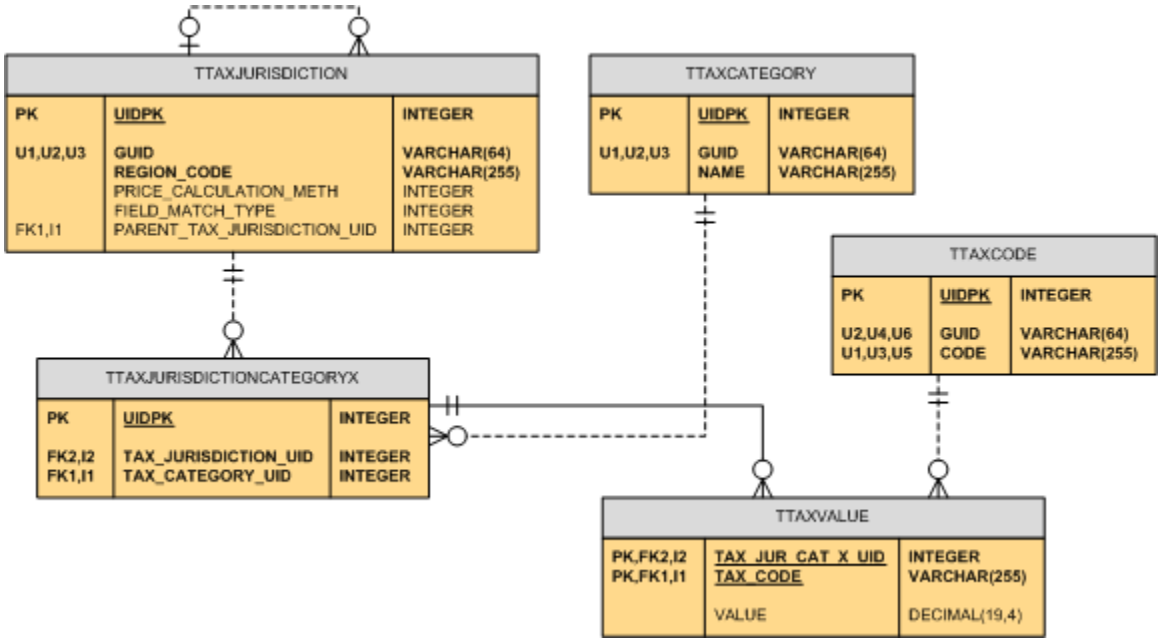
## Rules


TRULESET
PK
U1,U2,U3
UIDPK
NAME
SCENARIO
LAST_MODIFIED_DATE
INTEGER
VARCHAR(255)
INTEGER
DATETIME
TRULE
PK
I1
I1
U2,U3,U1
FK1,I2
UIDPK
START_DATE
END_DATE
ELIGIBILITY_OPERATOR
CONDITION_OPERATOR
NAME
PROMO_CODE
SINGLE_USE
RULE_SET_UID
INTEGER
DATETIME
DATETIME
INTEGER
INTEGER
VARCHAR(255)
VARCHAR(100)
INTEGER
INTEGER
TRULEPARAMETER
PK
FK1,I1
FK2,I2
UIDPK
PARAM_KEY
PARAM_VALUE
DISPLAY_TEXT
RULE_ELEMENT_UID
RULE_EXCEPTION_UID
INTEGER
VARCHAR(255)
VARCHAR(255)
VARCHAR(255)
INTEGER
INTEGER
TRULEELEMENT
PK
FK1,I1
UIDPK
TYPE
KIND
RULE_UID
INTEGER
VARCHAR(255)
VARCHAR(255)
INTEGER
TRULEEXCEPTION
PK
FK1,I1
UIDPK
TYPE
RULE_ELEMENT_UID
INTEGER
VARCHAR(255)
INTEGER

## Shipping




## Taxes



## *Database Tables*

| Table Name | OM Class | Description |
|---|---|---|
| TDIGITALASSETS | Tdigitalassets | This table represents a reference to digital good files stored on the file system. |
| TTAXCATEGORY | Ttaxcategory | This table represents a category of tax, i.e. GST (Canada). |
| TTAXCODE | Ttaxcode | This table represents a TaxCode. For example goods and books. |
| TTAXJURISDICTION | Ttaxjurisdiction | This table represents a TaxJurisdiction, a geographical area i.e. Canada. |
| TTAXJURISDICTIONCATEGORYX | Ttaxjurisdictioncategoryx | This table represents the TaxJurisdiction and TaxCategory association, include the taxValues for each configured TaxCode. |
| TTAXVALUE | Ttaxvalue | This table represents a TaxValue for a given TaxJurisdiction and TaxCategory. |
| TATTRIBUTE | Tattribute | This table represents a customized property of an object such as Category, Product or Product Sku. |
| TCUSTOMER | Tcustomer | This table represents a Customer account. |
| TCUSTOMERAUTHENTICATION | Tcustomerauthentication | This table represents a Customer authentication. |
| TCUSTOMERPROFILEVALUE | Tcustomerprofilevalue | This table represents the value of a customer profile. |
| TCUSTOMERDELETED | Tcustomerdeleted | This table represents the audit of a deleted Customer. |
| TADDRESS | Taddress | This table represents a Customer Address. |
| TCATEGORYTYPE | Tcategorytype | This table represents the type of a Category, which determines the set of attributes that it has. An example of a category type |

| Table Name | OM Class | Description |
|---|---|---|
| | | would be Electronics. |
| TCATEGORY | Tcategory | This table represents a collection of related Products. |
| TCATEGORYDELETED | Tcategorydeleted | This table represents a deleted category. |
| TCATEGORYATTRIBUTEVALUE | Tcategoryattributevalue | This table represents the Attribute Values for a given Category. |
| TCATEGORYLDF | Tcategoryldf | This table contains locale-dependent information about a Category. |
| TCATEGORYTYPEATTRIBUTE | Tcategorytypeattribute | This table represents a mapping of Category Attributes to a CategoryType |
| TCMUSER | Tcmuser | This table represents a person with an account in the system for accessing the Commerce Manager or web services. |
| TUSERROLE | Tuserrole | This table represents a user's role. |
| TCMUSERROLEX | Tcmuserrolex | This table represents the UserRoles assigned to a Customer |
| TCUSTOMERGROUP | Tcustomergroup | This table represents a customer group. |
| TCUSTOMERGROUPROLEX | Tcustomergrouprolex | This table represents the Role of a CustomerGroup. |
| TCUSTOMERGROUPX | Tcustomergroupx | This table represents the Customers assigned to a CustomerGroup. |
| TCUSTOMERSESSION | Tcustomersession | This table represents information about customers who may not be logged in. |
| TCUSTOMERCREDITCARD | Tcustomercreditcard | This table represents a credit card stored by a Storefront customer. |
| TIMPORTJOB | Timportjob | This table represents an import job. |

| Table Name | OM Class | Description |
|---|---|---|
| TIMPORTMAPPINGS | Timportmappings | This table represents the column to field mapping for an import job. |
| TORDERADDRESS | Torderaddress | This table represents a copy of a customer Address created with an Order. |
| TORDER | Torder | This table represents a customer Order. |
| TORDERNUMBERGENERATOR | Tordernumbergenerator | This table represents the next available order number. |
| TORDERNOTE | Tordernote | This table represents a note made on an order by a CSR. |
| TORDERPAYMENT | Torderpayment | This table represents customer payments made against an Order. |
| TORDERSHIPMENT | Tordershipment | This table represents a customer's order shipment. |
| TORDERRETURN | Torderreturn | This table represents a customer's order return. |
| TORDERTAX | Tordertax | This table represents the tax paid on a customer's Order. |
| TPRODUCTTYPE | Tproducttype | This table represents the type of a Product, which determines the set of attributes that it has. An example of a product type would be shoes. |
| TBRAND | Tbrand | This table represents product manufacturer/brand information. |
| TPRODUCT | Tproduct | This table represents a merchandise product. A product must have at least 1 ProductSku associated in order to be sold. |
| TORDERSKU | Tordersku | This table represents an order for a quantity of SKUs. |
| TORDERRETURNSKU | Torderreturnsku | This table represents the return of a quantity of SKUs |

| Table Name | OM Class | Description |
|---|---|---|
| | | for an order. |
| TPRODUCTATTRIBUTEVALUE | Tproductattributevalue | This table represents the value of a Product Attribute. |
| TPRODUCTCATEGORY | Tproductcategory | This table represents the association between a Category and its contained Products. |
| TPRODUCTLDF | Tproductldf | This table contains locale-dependent information about a Product. |
| TPRODUCTPRICE | Tproductprice | This table contains a set of ProductPriceTiers for each configured currency. |
| TPRODUCTPRICETIER | Tproductpricetier | This table contains a set of Product Prices Tiers. |
| TINVENTORY | Tinventory | This table represents the Inventory information for a ProductSku. |
| TINVENTORYAUDIT | Tinventoryaudit | This table contains an audit of inventory adjustment events. |
| TPRODUCTSKU | Tproductsku | This table represents a variation of a merchandise product. |
| TPRODUCTASSOCIATION | Tproductassociation | This table represents ProductAssociations between two products. Examples of product associations are Cross-Sells, Up-Sells, Accessories. |
| TPRODUCTSKUATTRIBUTEVALUE | Tproductskuattributevalue | This table represents the Attribute Values for a given Product. |
| TPRODUCTSKUPRICE | Tproductskuprice | This table contains a set of SkuPriceTiers for each configured currency |
| TSKUPRICETIER | Tskupricetier | This table contains a set of Sku Prices Tiers. |
| TPRODUCTTYPEATTRIBUTE | Tproducttypeattribute | This table represents a mapping of Product Attributes to a ProductType. |

| Table Name | OM Class | Description |
|---|---|---|
| TPRODUCTTYPESKUATTRIBUTE | Tproducttypeskuattribute | This table represents a mapping of ProductSKU Attributes to a ProductType. |
| TRULESET | Truleset | This table represents a set of promotion rules. |
| TRULE | Trule | This table represents a Rule that can be applied by the Rules Engine. |
| TRULEELEMENT | Truleelement | This table represents the component of a Rule. For example a condition or an action. |
| TRULEEXCEPTION | Truleexception | This table represents an exception of either a Rule action or condition. |
| TRULEPARAMETER | Truleparameter | This table represents a parameter of a rule condition, such as the category that a product must belong to to qualify for a promotion. |
| TSHOPPINGCART | Tshoppingcart | This table represents the state of a customer's shopping cart. |
| TCARTITEM | Tcartitem | This table represents a quantity of SKUs in a shopping cart, saved cart, wish list, etc. |
| TUSERROLEPERMISSIONX | Tuserrolepermissionx | This table represents the level of permission assigned to a given UserRole. |
| TSKUOPTION | Tskuoption | This table represents a SKU option that can be configured. For example size or color. |
| TSKUOPTIONVALUE | Tskuoptionvalue | This table represents an available option value for a SKU option. Example option values include red, green, small, large, etc. |
| TPRODUCTTYPESKUOPTION | Tproducttypeskuoption | This table represents the mapping of SKU options to a ProductType. |

| Table Name | OM Class | Description |
|---|---|---|
| TPRODUCTSKUOPTIONVALUE | Tproductskuoptionvalue | This table represents the mapping of a ProductSKU to a SKUOptionValue. |
| TSHIPPINGREGION | Tshippingregion | This table represents a region that will be associated with one or more shipping services. |
| TSHIPPINGCOSTCALCULATIONMETHOD | Tshippingcostcalculationmethod | This table represents a method to be used for shipping cost calculation; for example Fixed Price. |
| TSHIPPINGCOSTCALCULATIONPARAM | Tshippingcostcalculationparam | This table represents a parameter of a shipping cost calculation method, such as the dollar value of the fix base shipping cost. |
| TSHIPPINGSERVICELEVEL | Tshippingservicelevel | This table represents a ShippingOption, for example Next Day associated with a ShippingRegion. |
| TPRODUCTDELETED | Tproductdeleted | This table represents an audit of deleted products. |
| TLOCALIZEDPROPERTIES | Tlocalizedproperties | This table represents a list of localized properties. For example Brand names. |
| TDIGITALASSETAUDIT | Tdigitalassetaudit | This table represents the audit of a DigitalAsset download attempt. |
| TAPPLIEDRULE | Tappliedrule | This table represents a rule that has been applied to an order. |
| TTOPSELLER | Ttopseller | This table represents a category of top selling products. |
| TTOPSELLERPRODUCTS | Ttopsellerproducts | This table represents a rank of top selling products in the store. |
| TSFSEARCHLOG | Tsfsearchlog | This table represents a log of searches performed on the storefront. |

## TDIGITALASSETS

This table represents a reference to digital good files stored on the file system.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| FILE_NAME | VARCHAR | (255) | | FileName | | | X | |
| EXPIRY_DAYS | INTEGER | | | ExpiryDays | | | | |
| MAX_DOWNLOAD_TIMES | INTEGER | | | MaxDownloadTimes | | | | |

## TTAXCATEGORY

This table represents a category of tax, i.e. GST (Canada).

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |

## TTAXCODE

This table represents a TaxCode. For example goods and books.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| CODE | VARCHAR | (255) | | Code | | | X | |

## TTAXJURISDICTION

This table represents a TaxJurisdiction, a geographical area i.e. Canada.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| GUID | VARCHAR | (64) | | Guid | | | X |
| REGION_CODE | VARCHAR | (255) | | RegionCode | | | X |
| PRICE_CALCULATION_METH | INTEGER | | 0 | PriceCalculationMeth | | | |
| FIELD_MATCH_TYPE | INTEGER | | | FieldMatchType | | | |
| PARENT_TAX_JURISDICTION_UID | BIGINT | | | ParentTaxJurisdictionUid | | X | |

## TTAXJURISDICTIONCATEGORYX

This table represents the TaxJurisdiction and TaxCategory association, include the taxValues for each configured TaxCode.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| TAX_JURISDICTION_UID | BIGINT | | | TaxJurisdictionUid | | X | X | |
| TAX_CATEGORY_UID | BIGINT | | | TaxCategoryUid | | X | X | |

## TTAXVALUE

This table represents a TaxValue for a given TaxJurisdiction and TaxCategory.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| TAX_JUR_CAT_X_UID | BIGINT | | | TaxJurCatXUid | X | X | X | |
| TAX_CODE | VARCHAR | (255) | | TaxCode | X | X | X | |
| VALUE | DECIMAL | (19,4) | | Value | | | | |

## TATTRIBUTE

This table reprents a customized property of an object such as Category, Product or Product Sku.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| ATTRIBUTE_KEY | VARCHAR | (255) | | AttributeKey | | | X | |
| LOCALE_DEPENDANT | INTEGER | | 0 | LocaleDependant | | | X | |
| ATTRIBUTE_TYPE | INTEGER | | | AttributeType | | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| SEARCH_INDEX | INTEGER | | 0 | SearchIndex | | | | |
| REQUIRED | INTEGER | | 0 | Required | | | | |
| VALUE_LOOKUP_ENABLED | INTEGER | | 0 | ValueLookupEnabled | | | | |
| MULTI_VALUE_ENABLED | INTEGER | | 0 | MultiValueEnabled | | | | |
| GUIDED_NAVIGATION | INTEGER | | 0 | GuidedNavigation | | | | |
| ATTRIBUTE_USAGE | INTEGER | | | AttributeUsage | | | X | |
| SYSTEM | INTEGER | | 0 | System | | | | |

## TCUSTOMER

This table respresents a Customer account.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| USER_ID | VARCHAR | (255) | | UserId | | | X | |
| PREF_BILL_ADDRESS_UID | BIGINT | | | PrefBillAddressUid | | | | |
| PREF_SHIP_ADDRESS_UID | BIGINT | | | PrefShipAddressUid | | | | |
| CREATION_DATE | DATE | | | CreationDate | | | | |
| LAST_EDIT_DATE | TIMESTAMP | | | LastEditDate | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| STATUS | INTEGER | | | Status | | | X | |
| AUTHENTICATION_UID | BIGINT | | | AuthenticationUid | | | | |

## TCUSTOMERAUTHENTICATION

This table respresents a Customer authentication.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| PASSWORD | VARCHAR | (255) | | Password | | | | |

## TCUSTOMERPROFILEVALUE

This table represents the value of a customer profile.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X |
| ATTRIBUTE_TYPE | INTEGER | | | AttributeType | | | X |
| LOCALIZED_ATTRIBUTE_KEY | VARCHAR | (255) | | LocalizedAttributeKey | | | X |
| SHORT_TEXT_VALUE | VARCHAR | (255) | | ShortTextValue | | | |
| LONG_TEXT_VALUE | LONGVARCHAR | (65535) | | LongTextValue | | | |
| INTEGER_VALUE | INTEGER | | | IntegerValue | | | |
| DECIMAL_VALUE | DECIMAL | (19,2) | | DecimalValue | | | |
| BOOLEAN_VALUE | INTEGER | | 0 | BooleanValue | | | |
| DATE_VALUE | DATE | | | DateValue | | | |
| CUSTOMER_UID | BIGINT | | | CustomerUid | | X | |

## TCUSTOMERDELETED

This table represents the audit of a deleted Customer.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CUSTOMER_UID | BIGINT | | | CustomerUid | | | X | |
| DELETED_DATE | DATE | | | DeletedDate | | | X | |

## TADDRESS

This table represents a Customer Address.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| LAST_NAME | VARCHAR | (100) | | LastName | | | | |
| FIRST_NAME | VARCHAR | (100) | | FirstName | | | | |
| PHONE_NUMBER | VARCHAR | (50) | | PhoneNumber | | | | |
| FAX_NUMBER | VARCHAR | (50) | | FaxNumber | | | | |
| STREET_1 | VARCHAR | (200) | | Street1 | | | | |
| STREET_2 | VARCHAR | (200) | | Street2 | | | | |
| CITY | VARCHAR | (200) | | City | | | | |
| SUB_COUNTRY | VARCHAR | (200) | | SubCountry | | | | |
| ZIP_POSTAL_CODE | VARCHAR | (50) | | ZipPostalCode | | | | |
| COUNTRY | VARCHAR | (200) | | Country | | | | |
| COMMERCIAL | INTEGER | | 0 | Commercial | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| CUSTOMER_UID | BIGINT | | | CustomerUid | | X | | |

## TCATEGORYTYPE

This table represents the type of a Category, which determines the set of attributes that it has. An example of a category type would be Electronics.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| TEMPLATE | VARCHAR | (255) | | Template | | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TCATEGORY

This table represents a collection of related Products.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| START_DATE | DATE | | | StartDate | | | X | |
| END_DATE | DATE | | | EndDate | | | | |
| LAST_MODIFIED_DATE | TIMESTAMP | | | LastModifiedDate | | | X | |
| ORDERING | INTEGER | | | Ordering | | | | |
| HIDDEN | INTEGER | | 0 | Hidden | | | | |
| CATEGORY_TYPE_UID | BIGINT | | | CategoryTypeUid | | X | X | |
| PARENT_CATEGORY_UID | BIGINT | | | ParentCategoryUid | | X | | |
| CODE | VARCHAR | (64) | | Code | | | X | |

## TCATEGORYDELETED

This table represents a deleted category.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CATEGORY_UID | BIGINT | | | CategoryUid | | | X | |
| DELETED_DATE | DATE | | | DeletedDate | | | X | |

## TCATEGORYATTRIBUTEVALUE

This table represents the Attribute Values for a given Category.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X |
| ATTRIBUTE_TYPE | INTEGER | | | AttributeType | | | X |
| LOCALIZED_ATTRIBUTE_KEY | VARCHAR | (255) | | LocalizedAttributeKey | | | X |
| SHORT_TEXT_VALUE | VARCHAR | (255) | | ShortTextValue | | | |
| LONG_TEXT_VALUE | LONGVARCHAR | (65535) | | LongTextValue | | | |
| INTEGER_VALUE | INTEGER | | | IntegerValue | | | |
| DECIMAL_VALUE | DECIMAL | (19,2) | | DecimalValue | | | |
| BOOLEAN_VALUE | INTEGER | | 0 | BooleanValue | | | |
| DATE_VALUE | DATE | | | DateValue | | | |
| CATEGORY_UID | BIGINT | | | CategoryUid | | X | |

## TCATEGORYLDF

This table contains locale-dependent information about a Category.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| CATEGORY_UID | BIGINT | | | CategoryUid | X | X | X | |
| URL | VARCHAR | (255) | | Url | | | | |
| KEY_WORDS | VARCHAR | (255) | | KeyWords | | | | |
| DESCRIPTION | VARCHAR | (255) | | Description | | | | |
| TITLE | VARCHAR | (255) | | Title | | | | |
| DISPLAY_NAME | VARCHAR | (255) | | DisplayName | | | | |
| LOCALE | VARCHAR | (20) | | Locale | X | | X | |

## TCATEGORYTYPEATTRIBUTE

This table represents a mapping of Category Attributes to a CategoryType

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| ORDERING | INTEGER | | | Ordering | | | | |
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X | |
| CATEGORY_TYPE_UID | BIGINT | | | CategoryTypeUid | | X | X | |

## TCMUSER

This table represents a person with an account in the system for accessing the Commerce Manager or web services.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| USER_NAME | VARCHAR | (255) | | UserName | | | X | |
| EMAIL | VARCHAR | (255) | | Email | | | X | |
| FIRST_NAME | VARCHAR | (100) | | FirstName | | | | |
| LAST_NAME | VARCHAR | (100) | | LastName | | | | |
| PASSWORD | VARCHAR | (255) | | Password | | | X | |
| CM_ACCESS | INTEGER | | 0 | CmAccess | | | | |
| WS_ACCESS | INTEGER | | 0 | WsAccess | | | | |
| CREATION_DATE | DATE | | | CreationDate | | | | |
| LAST_LOGIN_DATE | DATE | | | LastLoginDate | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TUSERROLE

This table represents a user''s role.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| DESCRIPTION | VARCHAR | (255) | | Description | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TCMUSERROLEX

This table represents the UserRoles assigned to a Customer

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| CM_USER_UID | BIGINT | | | CmUserUid | X | X | X | |
| USER_ROLE_UID | BIGINT | | | UserRoleUid | X | X | X | |

## TCUSTOMERGROUP

This table represents a customer group.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TCUSTOMERGROUPROLEX

This table represents the Role of a CustomerGroup.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| CUSTOMER_GROUP_UID | BIGINT | | | CustomerGroupUid | | X | X | |
| CUSTOMER_ROLE | VARCHAR | (255) | | CustomerRole | | | | |

## TCUSTOMERGROUPX

This table represents the Customers assigned to a CustomerGroup.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| CUSTOMER_UID | BIGINT | | | CustomerUid | | X | X | |
| CUSTOMERGROUP_UID | BIGINT | | | CustomergroupUid | | X | X | |

## TCUSTOMERSESSION

This table represents information about customers who may not be logged in.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CREATION_DATE | DATE | | | CreationDate | | | X | |
| LAST_ACCESSED_DATE | TIMESTAMP | | | LastAccessedDate | | | X | |
| CUSTOMER_UID | BIGINT | | | CustomerUid | | X | | |
| EMAIL | VARCHAR | (255) | | Email | | | | |
| LOCALE | VARCHAR | (255) | | Locale | | | | |
| CURRENCY | VARCHAR | (3) | | Currency | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| IP_ADDRESS | VARCHAR | (255) | | IpAddress | | | | |

## TCUSTOMERCREDITCARD

This table represents a credit card stored by a Storefront customer.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CARD_TYPE | VARCHAR | (50) | | CardType | | | X | |
| CARD_HOLDER_NAME | VARCHAR | (100) | | CardHolderName | | | X | |
| CARD_NUMBER | VARCHAR | (255) | | CardNumber | | | X | |
| EXPIRY_YEAR | VARCHAR | (4) | | ExpiryYear | | | X | |
| EXPIRY_MONTH | VARCHAR | (2) | | ExpiryMonth | | | X | |
| START_YEAR | VARCHAR | (4) | | StartYear | | | | |
| START_MONTH | VARCHAR | (2) | | StartMonth | | | | |
| ISSUE_NUMBER | INTEGER | | | IssueNumber | | | | |
| DEFAULT_CARD | INTEGER | | | DefaultCard | | | | |
| CUSTOMER_UID | BIGINT | | | CustomerUid | | X | X | |
| BILLING_ADDRESS_UID | BIGINT | | | BillingAddressUid | | X | | |

## TIMPORTJOB

This table represents an import job.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| CSV_FILE_NAME | VARCHAR | (255) | | CsvFileName | | | X | |
| COL_DELIMETER | CHAR | (1) | | ColDelimeter | | | | |
| TEXT_QUALIFIER | CHAR | (1) | | TextQualifier | | | | |
| DATA_TYPE_NAME | VARCHAR | (255) | | DataTypeName | | | X | |
| IMPORT_TYPE | INTEGER | | | ImportType | | | X | |
| MAX_ALLOW_ERRORS | INTEGER | | | MaxAllowErrors | | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TIMPORTMAPPINGS

This table represents the column to field mapping for an import job.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| IMPORT_JOB_UID | BIGINT | | | ImportJobUid | X | X | X | |
| COL_NUMBER | INTEGER | | | ColNumber | | | X | |
| IMPORT_FIELD_NAME | VARCHAR | (255) | | ImportFieldName | X | | X | |

## TORDERADDRESS

This table represents a copy of a customer Address created with an Order.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| LAST_NAME | VARCHAR | (100) | | LastName | | | | |
| FIRST_NAME | VARCHAR | (100) | | FirstName | | | | |
| PHONE_NUMBER | VARCHAR | (50) | | PhoneNumber | | | | |
| FAX_NUMBER | VARCHAR | (50) | | FaxNumber | | | | |
| STREET_1 | VARCHAR | (200) | | Street1 | | | | |
| STREET_2 | VARCHAR | (200) | | Street2 | | | | |
| CITY | VARCHAR | (200) | | City | | | | |
| SUB_COUNTRY | VARCHAR | (200) | | SubCountry | | | | |
| ZIP_POSTAL_CODE | VARCHAR | (50) | | ZipPostalCode | | | | |
| COUNTRY | VARCHAR | (200) | | Country | | | | |
| COMMERCIAL | INTEGER | | 0 | Commercial | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TORDER

This table represents a customer Order.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| LAST_MODIFIED_DATE | TIMESTAMP | | | LastModifiedDate | | | |
| CREATED_DATE | DATE | | | CreatedDate | | | X |
| IP_ADDRESS | VARCHAR | (255) | | IpAddress | | | |
| ORDER_BILLING_ADDRESS_UID | BIGINT | | | OrderBillingAddressUid | | X | |
| TOTAL | DECIMAL | (19,2) | | Total | | | |
| SUBTOTAL | DECIMAL | (19,2) | | Subtotal | | | |
| BEFORE_TAX_TOTAL | DECIMAL | (19,2) | | BeforeTaxTotal | | | |
| BEFORE_TAX_SUBTOTAL | DECIMAL | (19,2) | | BeforeTaxSubtotal | | | |
| SUBTOTAL_DISCOUNT | DECIMAL | (19,2) | | SubtotalDiscount | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| STATUS | INTEGER | | 0 | Status | | | |
| ORDER_NUMBER | VARCHAR | (64) | | OrderNumber | | | X |
| CUSTOMER_UID | BIGINT | | | CustomerUid | | X | |
| LOCALE | VARCHAR | (5) | | Locale | | | X |
| CURRENCY | VARCHAR | (3) | | Currency | | | |
| INCLUSIVE_TAX_CALCULATION | INTEGER | | 0 | InclusiveTaxCalculation | | | |

## TORDERNUMBERGENERATOR

This table represents the next available order number.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | 1 | Uidpk | X | | X | |
| NEXT_ORDER_NUMBER | VARCHAR | (100) | | NextOrderNumber | | | X | |

## TORDERNOTE

This table represents a note made on an order by a CSR.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CREATED_DATE | DATE | | | CreatedDate | | | X | |
| CREATED_BY | BIGINT | | | CreatedBy | | X | X | |
| NOTE | VARCHAR | (255) | | Note | | | | |
| ORDER_UID | BIGINT | | | OrderUid | | X | | |

## TORDERPAYMENT

This table represents customer payments made against an Order.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CREATED_DATE | TIMESTAMP | | | CreatedDate | | | X | |
| CARD_TYPE | VARCHAR | (50) | | CardType | | | | |
| CARD_HOLDER_NAME | VARCHAR | (100) | | CardHolderName | | | | |
| CARD_NUMBER | VARCHAR | (255) | | CardNumber | | | | |
| EXPIRY_YEAR | VARCHAR | (4) | | ExpiryYear | | | | |
| EXPIRY_MONTH | VARCHAR | (2) | | ExpiryMonth | | | | |
| START_DATE | DATE | | | StartDate | | | | |
| ISSUE_NUMBER | VARCHAR | (100) | | IssueNumber | | | | |
| PAYMENT_GATEWAY | VARCHAR | (100) | | PaymentGateway | | | | |
| AMOUNT | DECIMAL | (19,2) | | Amount | | | | |
| REFERENCE_ID | VARCHAR | (50) | | ReferenceId | | | | |
| REQUEST_TOKEN | VARCHAR | (255) | | RequestToken | | | | |
| AUTHORIZATION_CODE | VARCHAR | (50) | | AuthorizationCode | | | | |
| TRANSACTION_TYPE | VARCHAR | (20) | | TransactionType | | | | |
| CURRENCY | VARCHAR | (10) | | Currency | | | | |
| EMAIL | VARCHAR | (100) | | Email | | | | |
| STATUS | INTEGER | | 0 | Status | | | | |
| ORDER_UID | BIGINT | | | OrderUid | | X | | |

## TORDERSHIPMENT

This table represents a customer"s order shipment.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| STATUS | INTEGER | | 1 | Status | | | |
| LAST_MODIFIED_DATE | TIMESTAMP | | | LastModifiedDate | | | |
| CREATED_DATE | DATE | | | CreatedDate | | | X |
| SHIPMENT_DATE | DATE | | | ShipmentDate | | | |
| CARRIER | VARCHAR | (255) | | Carrier | | | |
| SERVICE_LEVEL | VARCHAR | (255) | | ServiceLevel | | | |
| TRACKING_CODE | VARCHAR | (255) | | TrackingCode | | | |
| SHIPPING_COST | DECIMAL | (19,2) | | ShippingCost | | | |
| BEFORE_TAX_SHIPPING_COST | DECIMAL | (19,2) | | BeforeTaxShippingCost | | | |
| ORDER_ADDRESS_UID | BIGINT | | | OrderAddressUid | | X | |
| ORDER_UID | BIGINT | | | OrderUid | | X | |

## TORDERRETURN

This table represents a customer"s order return.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| RETURN_REASON | INTEGER | | 1 | ReturnReason | | | |
| CREATED_DATE | DATE | | | CreatedDate | | | X |
| RMA_CODE | VARCHAR | (255) | | RmaCode | | | |
| RETURN_COMMENT | VARCHAR | (2000) | | ReturnComment | | | |
| BEFORE_TAX_RETURN_TOTAL | DECIMAL | (19,2) | | BeforeTaxReturnTotal | | | |
| RETURN_TOTAL | DECIMAL | (19,2) | | ReturnTotal | | | |
| ORDER_UID | BIGINT | | | OrderUid | | X | |

## TORDERTAX

This table represents the tax paid on a customers Order.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| TAX_CATEGORY_NAME | VARCHAR | (255) | | TaxCategoryName | | | X |
| TAX_CATEGORY_DISPLAY_NAME | VARCHAR | (255) | | TaxCategoryDisplayName | | | X |
| VALUE | DECIMAL | (19,2) | | Value | | | |
| ORDER_UID | BIGINT | | | OrderUid | | X | |
| ORDER_RETURN_UID | BIGINT | | | OrderReturnUid | | X | |

## TPRODUCTTYPE

This table represents the type of a Product, which determines the set of attributes that it has. An example of a product type would be shoes.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| WITH_MULTIPLE_SKUS | INTEGER | | 0 | WithMultipleSkus | | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| TEMPLATE | VARCHAR | (255) | | Template | | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| TAX_CODE_UID | BIGINT | | | TaxCodeUid | | X | X | |

## TBRAND

This table represents product manufacturer/brand information.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CODE | VARCHAR | (255) | | Code | | | X | |
| IMAGE_URL | VARCHAR | (255) | | ImageUrl | | | | |

## TPRODUCT

This table represents represents a merchandise product. A product must have at least 1 ProductSku associated in order to be sold.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| LAST_MODIFIED_DATE | TIMESTAMP | | | LastModifiedDate | | | X | |
| START_DATE | DATE | | | StartDate | | | X | |
| END_DATE | DATE | | | EndDate | | | | |
| IMAGE | VARCHAR | (255) | | Image | | | | |
| PRODUCT_TYPE_UID | BIGINT | | | ProductTypeUid | | X | X | |
| BRAND_UID | BIGINT | | | BrandUid | | X | | |
| DEFAULT_SKU_UID | BIGINT | | | DefaultSkuUid | | | | |
| DEFAULT_CATEGORY_UID | BIGINT | | | DefaultCategoryUid | | X | | |
| CODE | VARCHAR | (64) | | Code | | | X | |
| MIN_QUANTITY | INTEGER | | 1 | MinQuantity | | | X | |
| HIDDEN | INTEGER | | 0 | Hidden | | | | |
| SALES_COUNT | INTEGER | | 0 | SalesCount | | | | |
| TAX_CODE_UID | BIGINT | | | TaxCodeUid | | X | | |

## TORDERSKU

This table represents an order for a quantity of SKUs.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| LAST_MODIFIED_DATE | TIMESTAMP | | | LastModifiedDate | | | | |
| CREATED_DATE | TIMESTAMP | | | CreatedDate | | | X | |
| SKUCODE | VARCHAR | (255) | | Skucode | | | X | |
| TAXCODE | VARCHAR | (255) | | Taxcode | | | X | |
| PRODUCT_UID | BIGINT | | | ProductUid | | | | |
| PRODUCT_SKU_UID | BIGINT | | | ProductSkuUid | | | | |
| ORDER_UID | BIGINT | | | OrderUid | | X | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ORDER_SHIPMENT_UID | BIGINT | | | OrderShipmentUid | | X | | |
| QUANTITY | INTEGER | | | Quantity | | | | |
| DISPLAY_NAME | VARCHAR | (255) | | DisplayName | | | X | |
| DISPLAY_NAME_CSR | VARCHAR | (255) | | DisplayNameCsr | | | | |
| AMOUNT | DECIMAL | (19,2) | | Amount | | | | |
| TAX | DECIMAL | (19,2) | | Tax | | | | |
| LIST_PRICE | DECIMAL | (19,2) | | ListPrice | | | | |
| UNIT_PRICE | DECIMAL | (19,2) | | UnitPrice | | | | |
| DISPLAY_SKU_OPTIONS | VARCHAR | (255) | | DisplaySkuOptions | | | | |
| IMAGE | VARCHAR | (255) | | Image | | | | |
| WEIGHT | INTEGER | | 0 | Weight | | | | |
| DIGITAL_ASSET_UID | BIGINT | | | DigitalAssetUid | | X | | |

## TORDERRETURNSKU

This table represents the return of a quantity of SKUs for an order.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| ORDER_SKU_UID | BIGINT | | | OrderSkuUid | | X | | |
| ORDER_RETURN_UID | BIGINT | | | OrderReturnUid | | X | | |
| QUANTITY | INTEGER | | | Quantity | | | | |
| RETURN_AMOUNT | DECIMAL | (19,2) | | ReturnAmount | | | | |

## TPRODUCTATTRIBUTEVALUE

This table represents the value of a Product Attribute.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X |
| ATTRIBUTE_TYPE | INTEGER | | | AttributeType | | | X |
| LOCALIZED_ATTRIBUTE_KEY | VARCHAR | (255) | | LocalizedAttributeKey | | | X |
| SHORT_TEXT_VALUE | VARCHAR | (255) | | ShortTextValue | | | |
| LONG_TEXT_VALUE | LONGVARCHAR | (65535) | | LongTextValue | | | |
| INTEGER_VALUE | INTEGER | | | IntegerValue | | | |
| DECIMAL_VALUE | DECIMAL | (19,2) | | DecimalValue | | | |
| BOOLEAN_VALUE | INTEGER | | 0 | BooleanValue | | | |
| DATE_VALUE | DATE | | | DateValue | | | |
| PRODUCT_UID | BIGINT | | | ProductUid | | X | |

## TPRODUCTCATEGORY

This table represents the association between a Category and it''s contained Products.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| PRODUCT_UID | BIGINT | | | ProductUid | | X | X | |
| CATEGORY_UID | BIGINT | | | CategoryUid | | X | X | |
| FEAT_PRODUCT_ORDER | INTEGER | | 0 | FeatProductOrder | | | | |

## TPRODUCTLDF

This table contains locale-dependent information about a Product.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| PRODUCT_UID | BIGINT | | | ProductUid | X | X | X | |
| URL | VARCHAR | (255) | | Url | | | | |
| KEY_WORDS | VARCHAR | (255) | | KeyWords | | | | |
| DESCRIPTION | VARCHAR | (255) | | Description | | | | |
| TITLE | VARCHAR | (255) | | Title | | | | |
| DISPLAY_NAME | VARCHAR | (255) | | DisplayName | | | | |
| LOCALE | VARCHAR | (20) | | Locale | X | | X | |

## TPRODUCTPRICE

This table contains a set of ProductPriceTiers for each configured currency.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CURRENCY | VARCHAR | (255) | | Currency | | | | |
| PRODUCT_UID | BIGINT | | | ProductUid | | X | | |

## TPRODUCTPRICETIER

This table contains a set of Product Prices Tiers.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| LIST_PRICE | DECIMAL | (19,2) | | ListPrice | | | X | |
| SALE_PRICE | DECIMAL | (19,2) | | SalePrice | | | | |
| MIN_QUANTITY | INTEGER | | | MinQuantity | X | | X | |
| PRODUCT_PRICE_UID | BIGINT | | | ProductPriceUid | X | X | X | |

## TINVENTORY

This table represents the Inventory information for a ProductSku.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| QUANTITY_ON_HAND | INTEGER | | | QuantityOnHand | | | |
| INFINITE_QUANTITY_ON_HAND | INTEGER | | 0 | InfiniteQuantityOnHand | | | |
| RESERVED_QUANTITY | INTEGER | | | ReservedQuantity | | | |
| REORDER_MINIMUM | INTEGER | | 0 | ReorderMinimum | | | |
| VISIBLE_WHEN_OUT_OF_STOCK | INTEGER | | 0 | VisibleWhenOutOfStock | | | |
| RESTOCK_DATE | DATE | | | RestockDate | | | |

## TINVENTORYAUDIT

This table contains an audit of inventory adjustment events.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| INVENTORY_UID | BIGINT | | | InventoryUid | | | X | |
| QUANTITY | INTEGER | | | Quantity | | | | |
| LOG_COMMENT | LONGVARCHAR | (65535) | | LogComment | | | | |
| CMUSER_UID | BIGINT | | | CmuserUid | | | X | |
| LOG_DATE | TIMESTAMP | | | LogDate | | | | |

## TPRODUCTSKU

This table represents a variation of a merchandise product.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| START_DATE | DATE | | | StartDate | | | X | |
| END_DATE | DATE | | | EndDate | | | | |
| SKUCODE | VARCHAR | (255) | | Skucode | | | X | |
| IMAGE | VARCHAR | (255) | | Image | | | | |
| INVENTORY_UID | BIGINT | | | InventoryUid | | X | | |
| PRODUCT_UID | BIGINT | | | ProductUid | | X | | |
| SHIPPABLE | INTEGER | | 1 | Shippable | | | | |
| WEIGHT | DECIMAL | (19,2) | 0 | Weight | | | | |
| HEIGHT | DECIMAL | (19,2) | 0 | Height | | | | |
| WIDTH | DECIMAL | (19,2) | 0 | Width | | | | |
| LENGTH | DECIMAL | (19,2) | 0 | Length | | | | |
| DIGITAL_ASSET_UID | BIGINT | | | DigitalAssetUid | | X | | |

## TPRODUCTASSOCIATION

This table represents ProductAssociations between two products. Example of product associations are Cross-Sells, Up-Sells, Accessories.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| ASSOCIATION_TYPE | INTEGER | | | AssociationType | | | X |
| SOURCE_PRODUCT_UID | BIGINT | | | SourceProductUid | | X | X |
| TARGET_PRODUCT_UID | BIGINT | | | TargetProductUid | | X | X |
| TARGET_SKU_UID | BIGINT | | | TargetSkuUid | | X | |
| START_DATE | DATE | | | StartDate | | | X |
| END_DATE | DATE | | | EndDate | | | |
| DEFAULT_QUANTITY | INTEGER | | 1 | DefaultQuantity | | | X |
| SOURCE_PRODUCT_DEPENDENT | INTEGER | | 0 | SourceProductDependent | | | |
| ORDERING | INTEGER | | 0 | Ordering | | | |

## TPRODUCTSKUATTRIBUTEVALUE

This table represents the Attribute Values for a given Product.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X |
| ATTRIBUTE_TYPE | INTEGER | | | AttributeType | | | X |
| LOCALIZED_ATTRIBUTE_KEY | VARCHAR | (255) | | LocalizedAttributeKey | | | X |
| SHORT_TEXT_VALUE | VARCHAR | (255) | | ShortTextValue | | | |
| LONG_TEXT_VALUE | LONGVARCHAR | (65535) | | LongTextValue | | | |
| INTEGER_VALUE | INTEGER | | | IntegerValue | | | |
| DECIMAL_VALUE | DECIMAL | (19,2) | | DecimalValue | | | |
| BOOLEAN_VALUE | INTEGER | | 0 | BooleanValue | | | |
| DATE_VALUE | DATE | | | DateValue | | | |
| PRODUCT_SKU_UID | BIGINT | | | ProductSkuUid | | X | |

## TPRODUCTSKUPRICE

This table contains a set of SkuPriceTiers for each configured currency.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CURRENCY | VARCHAR | (255) | | Currency | | | | |
| PRODUCT_SKU_UID | BIGINT | | | ProductSkuUid | | X | | |

## TSKUPRICETIER

This table contains a set of Sku Prices Tiers.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| LIST_PRICE | DECIMAL | (19,2) | | ListPrice | | | | |
| SALE_PRICE | DECIMAL | (19,2) | | SalePrice | | | | |
| MIN_QUANTITY | INTEGER | | | MinQuantity | | | | |
| SKU_PRICE_UID | BIGINT | | | SkuPriceUid | | X | | |

## TPRODUCTTYPEATTRIBUTE

This table represents a mapping of Product Attributes to a ProductType.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| ORDERING | INTEGER | | | Ordering | | | | |
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X | |
| PRODUCT_TYPE_UID | BIGINT | | | ProductTypeUid | | X | X | |

## TPRODUCTTYPESKUATTRIBUTE

This table represents a mapping of ProductSKU Attributes to a ProductType.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| ORDERING | INTEGER | | | Ordering | | | | |
| ATTRIBUTE_UID | BIGINT | | | AttributeUid | | X | X | |
| PRODUCT_TYPE_UID | BIGINT | | | ProductTypeUid | | X | X | |

## TRULESET

This table represents a set of promotion rules.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LAST_MODIFIED_DATE | TIMESTAMP | | | LastModifiedDate | | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| SCENARIO | INTEGER | | | Scenario | | | X | |

## TRULE

This table represents a Rule that can be applied by the Rules Engine.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| START_DATE | DATE | | | StartDate | | | | |
| END_DATE | DATE | | | EndDate | | | | |
| ELIGIBILITY_OPERATOR | INTEGER | | 0 | EligibilityOperator | | | | |
| CONDITION_OPERATOR | INTEGER | | 0 | ConditionOperator | | | | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| PROMO_CODE | VARCHAR | (100) | | PromoCode | | | | |
| SINGLE_USE | INTEGER | | 0 | SingleUse | | | | |
| RULE_SET_UID | BIGINT | | | RuleSetUid | | X | | |

## TRULEELEMENT

This table represents the component of a Rule. For example a condition or an action.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| TYPE | VARCHAR | (255) | | Type | | | X | |
| KIND | VARCHAR | (255) | | Kind | | | X | |
| RULE_UID | BIGINT | | | RuleUid | | X | | |

## TRULEEXCEPTION

This table represents an exception of either a Rule action or condition.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| TYPE | VARCHAR | (255) | | Type | | | X | |
| RULE_ELEMENT_UID | BIGINT | | | RuleElementUid | | X | | |

## TRULEPARAMETER

This table represents a parameter of a rule condition, such as the category that a product must belong to to qualify for a promotion.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| PARAM_KEY | VARCHAR | (255) | | ParamKey | | | X | |
| PARAM_VALUE | VARCHAR | (255) | | ParamValue | | | X | |
| DISPLAY_TEXT | VARCHAR | (255) | | DisplayText | | | | |
| RULE_ELEMENT_UID | BIGINT | | | RuleElementUid | | X | | |
| RULE_EXCEPTION_UID | BIGINT | | | RuleExceptionUid | | X | | |

## TSHOPPINGCART

This table represents the state of a customers shopping cart.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| GUID | VARCHAR | (100) | | Guid | | | X | |

## TCARTITEM

This table represents a quantity of SKUs in a shopping cart, saved cart, wish list, etc.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| TYPE | VARCHAR | (255) | | Type | | | X | |
| SKU_UID | BIGINT | | | SkuUid | | X | | |
| QUANTITY | INTEGER | | | Quantity | | | | |
| SHOPPING_CART_UID | BIGINT | | | ShoppingCartUid | | X | | |
| PARENT_CART_ITEM_UID | BIGINT | | | ParentCartItemUid | | | | |

## TUSERROLEPERMISSIONX

This table represents the level of permission assigned to a given UserRole.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| ROLE_UID | BIGINT | | | RoleUid | | X | X | |
| USER_PERMISSION | VARCHAR | (255) | | UserPermission | | | | |

## TSKUOPTION

This table represents a SKU option that can be configured. For example size or color.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| OPTION_KEY | VARCHAR | (100) | | OptionKey | | | X | |

## TSKUOPTIONVALUE

This table represents an available option value for a SKU option. Example option values include red, green, small, large, etc.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| OPTION_VALUE_KEY | VARCHAR | (255) | | OptionValueKey | | | X | |
| ORDERING | INTEGER | | | Ordering | | | | |
| SKU_OPTION_UID | BIGINT | | | SkuOptionUid | | X | X | |
| IMAGE | VARCHAR | (255) | | Image | | | | |

## TPRODUCTTYPESKUOPTION

This table represents the mapping of SKU options to a ProductType.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| PRODUCT_TYPE_UID | BIGINT | | | ProductTypeUid | | X | X | |
| SKU_OPTION_UID | BIGINT | | | SkuOptionUid | | X | X | |
| ORDERING | BIGINT | | 0 | Ordering | | | | |

## TPRODUCTSKUOPTIONVALUE

This table represents the mapping of a ProductSKU to a SKUOptionValue.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| PRODUCT_SKU_UID | BIGINT | | | ProductSkuUid | | X | X | |
| OPTION_KEY | VARCHAR | (100) | | OptionKey | | | X | |
| OPTION_VALUE_UID | BIGINT | | | OptionValueUid | | X | X | |

## TSHIPPINGREGION

This table represents a region that will be associated with one or more shipping services.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| NAME | VARCHAR | (255) | | Name | | | X | |
| REGION_STR | VARCHAR | (2000) | | RegionStr | | | | |
| GUID | VARCHAR | (64) | | Guid | | | X | |

## TSHIPPINGCOSTCALCULATIONMETHOD

This table represents a method to be used for shipping cost calculation; for example Fixed Price.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| TYPE | VARCHAR | (255) | | Type | | | X | |

## TSHIPPINGCOSTCALCULATIONPARAM

This table represents a parameter of a shipping cost calculation method, such as the dollar value of the fix base shipping cost.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| PARAM_KEY | VARCHAR | (255) | | ParamKey | X | | X | |
| VALUE | VARCHAR | (255) | | Value | | | X | |
| DISPLAY_TEXT | VARCHAR | (255) | | DisplayText | | | X | |
| SCCM_UID | BIGINT | | | SccmUid | X | X | X | |

## TSHIPPINGSERVICELEVEL

This table represents a ShippingOption, for example Next Day assoicated with a ShippingRegion.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| GUID | VARCHAR | (64) | | Guid | | | X | |
| SHIPPING_REGION_UID | BIGINT | | | ShippingRegionUid | | X | X | |
| SCCM_UID | BIGINT | | | SccmUid | | X | X | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CARRIER | VARCHAR | (255) | | Carrier | | | | |

## TPRODUCTDELETED

This table represents an audit of deleted products.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| PRODUCT_UID | BIGINT | | | ProductUid | | | X | |
| DELETED_DATE | DATE | | | DeletedDate | | | X | |

## TLOCALIZEDPROPERTIES

This table represents a list of localized properties. For example Brand names.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| OBJECT_UID | BIGINT | | | ObjectUid | X | | X | |
| LOCALIZED_PROPERTY_KEY | VARCHAR | (255) | | LocalizedPropertyKey | X | | X | |
| VALUE | VARCHAR | (255) | | Value | | | X | |

## TDIGITALASSETAUDIT

This table represents the audit of a DigitalAsset download attempt.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| ORDERSKU_UID | BIGINT | | | OrderskuUid | | | X | |
| DIGITALASSET_UID | BIGINT | | | DigitalassetUid | | | X | |
| DOWNLOAD_TIME | DATE | | | DownloadTime | | | X | |
| IP_ADDRESS | VARCHAR | (255) | | IpAddress | | | | |

## TAPPLIEDRULE

This table represents a rule that has been applied to an order.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| ORDER_UID | BIGINT | | | OrderUid | | | X | |
| RULE_UID | BIGINT | | | RuleUid | | | X | |
| RULE_NAME | VARCHAR | (255) | | RuleName | | | X | |
| RULE_CODE | LONGVARCHAR | (65535) | | RuleCode | | | X | |

## TTOPSELLER

This table represents a category of top selling products.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X | |
| CATEGORY_UID | BIGINT | | | CategoryUid | | | X | |

## TTOPSELLERPRODUCTS

This table represents a rank of top selling products in the store.

| Name | Type | Size | Default | JavaName | PK | FK | not null | Description |
|---|---|---|---|---|---|---|---|---|
| TOP_SELLER_UID | BIGINT | | | TopSellerUid | | X | X | |
| PRODUCT_UID | BIGINT | | | ProductUid | | | X | |
| SALES_COUNT | INTEGER | | | SalesCount | | | X | |

## TSFSEARCHLOG

This table represents a log of searches performed on the storefront.

| Name | Type | Size | Default | JavaName | PK | FK | not null |
|---|---|---|---|---|---|---|---|
| UIDPK | BIGINT | | | Uidpk | X | | X |
| SEARCH_TIME | TIMESTAMP | | | SearchTime | | | |
| KEYWORDS | VARCHAR | (255) | | Keywords | | | |
| RESULT_COUNT | INTEGER | | | ResultCount | | | X |
| SUGGESTIONS_GENERATED | INTEGER | | 0 | SuggestionsGenerated | | | |
| CATEGORY_RESTRICTION | BIGINT | | 0 | CategoryRestriction | | | X |

# *Miscellaneous*

## Database Compatibility Issues

### Oracle

- Query Length Limitatin
  Since Oracle only allow 1000 expressions in one query, we have to separate big query over 1000 expressions like "select ... from ... where ... in (..., ...)" to multiple queries. The following code from ProductServiceImpl.findByUids() is the recommended way to do it.

```
final List queries = this.getUtility().composeQueries(Criteria.PRODUCT_BY_UIDS,
Criteria.PLACE_HOLDER_FOR_LIST, productUids);
    final List result = getPersistenceEngine().retrieve(queries);
```

- Ampersand(&) cannot be used in sql scripts.

```
INSERT INTO TBRAND (UIDPK,CODE,GUID)  VALUES (1110,'D&H', 'D&H');
```

  must be changed to :

```
INSERT INTO TBRAND (UIDPK,CODE,GUID)  VALUES (1110,concat('D',
concat(CHR(38), 'H')), concat('D', concat(CHR(38), 'H')));
```

- Literal date value like '2006-11-11 11:11:11' cannot be used in sql scripts.
  It must be changed to :

```
to_date('2005-06-10','YYYY-MM-DD')
```

  That means if you have literal date value in a torque file, the generated sql script for Oracle won't work.
  You'll have to manually change it.

# API Reference

The complete API reference is available here:
http://edocs.bea.com/alcs/docs60/javadoc.

# AquaLogic Commerce Services Web Services

Web Services is a module included in AquaLogic Commerce Services which allows you to integrate your AquaLogic Commerce Services store with external systems.

Web Services integrate Apache Axis (an open source Web Services engine for Java) along with Dozer and XDoclet to expose services from the AquaLogic Commerce Services service layer as Web Services to external systems. Once you define which services to include in the Web Services API (by editing configuration files) Web Services will build your Web Services API for you.

## *What are Web Services?*

Web Services is a term used to define a set of technologies that exposes business functionality over the Web as a set of declarative interfaces. These interfaces allow applications to discover and integrate with other applications over the web using standard Internet protocols (i.e. HTTP) and data format (i.e. XML).

A web service:

- Is a programmable application, accessible as a component via standard Web protocols.
- Uses standard Web protocols like HTTP, XML and SOAP.
- Works through existing proxies and firewalls.
- Can take advantage of HTTP authentication.
- Supports encryption for free with SSL.
- Supports easy incorporation with existing XML messaging solutions.
- Takes advantage of XML messaging schemas and easy transition from XML RPC solutions.
- Combines the best aspects of component-based development and the Web.
- Is available to a variety of clients (platform independent).

Web Services is being considered the next "big thing" in software development. It represents an important evolutionary step in building distributed applications. Typical application areas are business-to-business integration, business process integration and management, content management, and design collaboration. Most business will eventually become both suppliers and consumers of Web Services. Web Services take what HTML and TCP/IP started, and add the element of XML to enable task-focused services that come together dynamically over the Web.

ALCS Web Services

ElasticPath Web Services is one of the methods that remote clients can use to access the ElasticPath core application. The Web Services API is a well defined, SOAP-based layer that is intended to provide integration functionality for ElasticPath clients. It is not intended to be a public interface.

The ALCS Web Services product consists of essentially four parts:

1. A public, use-case-based API for performing operations
2. An implementation of the public API that integrates with the Elastic Path core library
3. A remoting layer implemented using the Java JAX-WS and JAXB standards (specifically the JAX-WS Reference Implementation from SUN). This is the layer that gives the Web Services web application its web services capabilities.
4. A pre-generated Web Services client library in Java. Any web services-capable language is in theory capable of being a client to the Web Services server, but a Java client library is supplied with the product.

The design goals of the API layer are to make it stable, consistent, complete, robust, and useable:

**Stability -** the API is designed to be stable even when the underlying application changes. To achieve this, a separate layer of services and data transfer objects (DTO) is created that should be able to adapt to changes to the underlying services and domain objects it proxies.

**Consistency** - the API is designed to be consistent from the client's perspective. It should provide an intuitive interface with consistent error handling, parameter validation, list handling, etc.

**Completeness** - the API should be complete as far as providing enough functionality for clients to reasonably achieve goals for common integration use cases. Obviously, this is a moving target and the API will undergo some revision, however it should at every release provide a decent base amount of functionality

**Robustness** - the API is designed to be robust in a couple of senses: it should withstand a reasonably amount of stress from external client calls and it should be able to withstand changes to the underlying application without breaking external clients

**Usability** - the API is designed to be usable by clients and developers. It uses JAX-WS as the underlying SOAP engine and WSDL for client library generation.

## *Service design*

Web services calls incur a certain amount of overhead due to the fact that they are made over HTTP and require JAX-WS translations (marshalling and unmarshalling XML into Java objects). So, in order to provide reasonable performance a key decision is to approach service design based on specific use cases and to try to model as few service calls per use case as possible, ideally requiring only single call per use case.

Services are composed of related operations in a similar way to the service design of the underlying application. Each service exposes a WSDL.

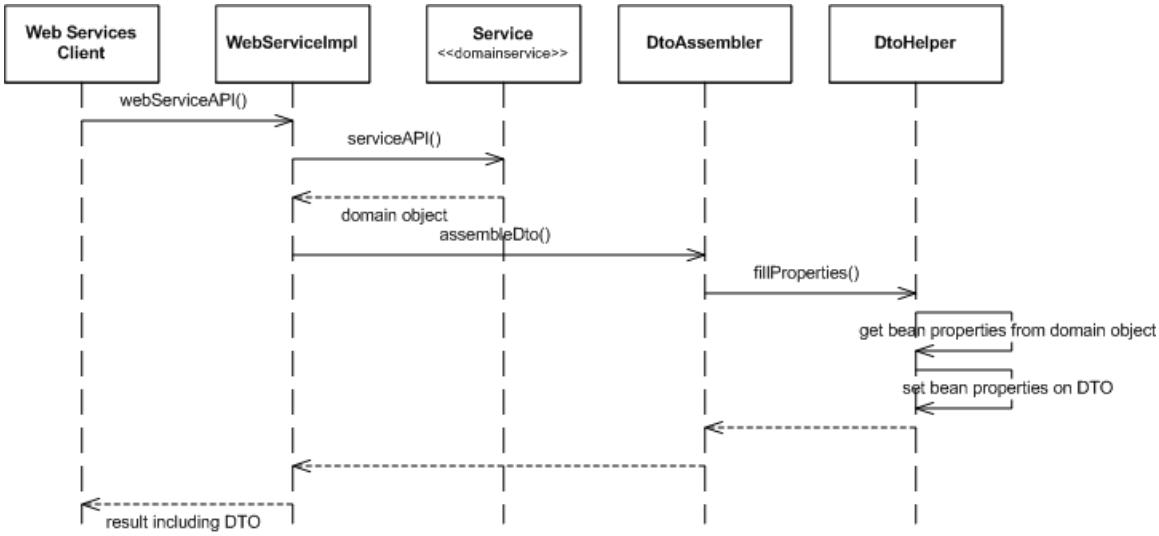
## *Request/response objects*

Each API method accepts a "request" object and (in most cases) returns a "response" object. The request object encapsulates all the fields necessary to instruct the API on which function to perform and how much data to return. The response object is either a plain Result (containing status and any messages) or a more complex object made up of API-specific information plus Result, plus (optionally) Paging information.

## *DTO translation*

To maintain a stable API it's necessary to decouple the domain objects from the web services. We created a layer of data transfer objects, which are modelled on the domain objects but typically contain a much smaller subset of fields and methods. "Assemblers" translate domain objects to and from DTOs.

There are 2 approaches to DTO translation, manually copying fields from the domain object to the DTO or automatic population using a reflection-based method that compares fields in the domain object and the DTO and reads and writes dynamically from one to the other. Following is a sequence diagram indicating how the automatic assembler populates a DTO from a domain object:



With both the manual and automatic assembers, if the underlying domain object is refactored (particularly if an exported field is modified), the assembler will likely need to be updated to handle the change on the DTO.

**Updating the manual assembler** - if a domain object's field is renamed from say, cartItems to allCartItems and its' accessor method is renamed from getCartItems() to getAllCartItems(), then the assembler would need to be refactored to read cart items with the getAllCartItems() method.

**Updating the automatic assembler**- if the same change is made to the domain object, then the reflection assembler would need to be modified to pass a Map of the source object's (domain object) field name to the target object's (DTO) field name, i.e. it would contain a single pair: allCartItems à cartItems. This will allow the assembler to resolve the correct matching fields. See DtoHelper.fillFields().

**Note:** since the automatic assembler uses reflection, it is possible to miss changes to the underlying domain objects at compile-time. Thus, it's crucial to have 100% unit test coverage of the automatic assembler translations.

## *Aspect-Oriented Programming (AOP)*

There are 2 places that aspects are used in the web services design: error handling and parameter validation. The ServiceInterceptor applies before method advice for Validation (see ValidationAdvisor) and catches Exception to provide error handling advice. The next 2 sections describe the aspect implementation for error handling and validation.

## *Error handling*

The web services API provides a common way for reporting errors in the Result object, which is returned by each API call. Result has a Status (SUCCESS or FAILURE) and a list of Messages. Each Message has a MessageStatus (ERROR, WARNING, INFO) and and information string. Services are designed to handle known exception cases by constructing a FAILURE result with some kind of useful message and to handle unknown exception cases by allowing runtime exceptions to propogate to an AOP interceptor.

### Runtime Exceptions

Runtime exceptions are caught by an AOP interceptor that logs an error on the ElasticPath side with a unique ID and creates a Result with an error message that contains the same unique ID. Clients can retrieve the failed Result and log the message containing the unique ID so that system administrators will later be able to correlate the client's logs with the ElasticPath logs to troubleshoot errors.

**Note:** this implementation requires that no web service code catch and swallow Exception or Throwable unless strictly necessary to the API - unexpected errors must be allowed to bubble up to the AOP layer.

## *Parameter validation*

There are 2 approaches to parameter validation of DTOs, manually applying validation rules in the DTO or service object or automatically applying validation with an AOP aspect. The AOP validation uses annotations on the fields in the request object, verifying whether the required fields are present, paging fields have valid offset/max values, etc. For a DTO to be validated with AOP it must implement the marker interface, Validatable.

```
public class DeleteSkuRequest implements Validatable {


   @XmlElement(name = "SkuCode", required = true)
   private String skuCode;


   ...
}
```

## *Paging*

There is a standard for sending and receiving Paging information in API requests that return lists of objects. Clients can send a Paging object, specifying the offset and max results (per page) and the web services should return an appropriately paged list along with a PagingResponse object that contains the total number of results as well as the offset and max results specified by the client.

## *Unit tests*

ElasticPath web services has unit tests of the client-generated proxy code, server-side service implementations and DTO assemblers.

- Client tests - exercise the service API against a running webserver/database and may need some minimal amount of manual data configuration in order to pass. Usually it should be enough to have imported the base snapitup store catalog, however some tests require user activity, such as ShoppingCart test for getCart().
- Server-side tests - exercise the service API replacing real service and domain object calls with Mock objects
- DTO assember tests - also server-side tests using Mock objects, however as mentioned above it is critically important to have 100% test coverage of DTO assembers that use the automatic reflection-based method of translating from domain object to DTO and vice versa

## *Web Services Security*

AquaLogic Commerce Services Connect currently uses SSL (HTTPS) in conjunction with HTTP Basic Authorization to provide security for web services. Acegi is used (as in the Storefront and Commerce Manager) to provide authentication. If your EP Connect application will be exposed to the internet, please ensure that HTTPS is enabled since Basic information essentially sent unencrypted over the network.

Currently, Web Services users must be configured as a Web Services User (to allow WS access) through the Commerce Manager. Out of the box, users with the super user role can access all web services (i.e., product, customer, inventory, order).

## *Using Web Services*

### Where are the WSDLs?

All the web services have links to their WSDLs that can be accessed at the following URL:

http://localhost:7001/webservices/shoppingcartwebservice?wsdl

for AquaLogic Commerce Services Web Services deployed locally on port 7001 with a root context of *webservices*.

If you access directly a Web Service URL:

http://localhost:7001/webservices/shoppingcartwebservice

an HTTP Basic user/pass prompt should appear. Enter the username and password of a Web Services User with the SUPERUSER user role (i.e., the admin user by default). If the correct username and password are entered a page displaying the web service and links to WSDLs will appear. The AdminService and Version web services are enabled by default by Axis. Click on the WSDL link(s) to download and view the WSDL. With the downloaded WSDL one can construct web service client code to call AquaLogic Commerce Services Web Services.

### Client Testing

Download the WSDL for the web service you'd like to test and use your favorite language/web services toolkit to create a web services client; i.e., Java's Axis framework can generate web services client code and skeleton JUnit test cases based on WSDLs.

## *Available Web Services and Methods*

Please refer to Appendix A.

# Exception Handling

This document reviews exception handling practices in Java as well as how they are handled in the Web tier, AJAX, and the Web Services component of AquaLogic Commerce Services.

## *Exception handing practices*

The following exception handling tips and practices should be observed.

- Never throw java.lang.RuntimeException or java.lang.Exception. Use a more specific exception that indicates the nature of the problem.
- Never catch an exception just to log it and then ignore it or rethrow the same exception to a higher application layer. In the AquaLogic Commerce Services applications, all exceptions will be caught and logged in the web layer.
- You may catch a checked exception, wrap it with a generic exception (see list of generic exceptions below), and throw it to a higher layer.
- If you create a new exception in a method, remember to put it in the method declaration header. This is especially important for runtime exceptions.
- Throw exceptions appropriate to the abstraction. In other words, exceptions thrown by a method should be defined at an abstraction level consistent with what the method does, not necessarily with the low-level details of how it is implemented.
- Fail early; Fail loudly - If you have an error let it be thrown and don't hide it. It is better to throw it early than find out later.
- Be sure to throw the proper type of exception, e.g. do not throw an EpDomainException in service layer code.
- Make the exception message well versed since these messages may be exposed to end-user in the Commerce Manager or through a web service.

The following code snippet shows examples of preferred and discouraged error handling practices.

```
//The error handling approach in this method is discouraged.
//Objects of the wrong type are ignored, potentially hiding
//a serious error.
public int compare(Object o1, Object o2) {
    int result = 0;
    String time1 = "";
    String time2 = "";
    if (o1 instanceof String && o2 instanceof String) {
        final String time1 = (String)o1;
        final String time2 = (String)o2;
```

```
        result = Timestamp.valueOf(time1).compareTo(Timestamp.valueOf(time2));
    }
    return result;
}

//This method is preferred. In this example, the code is less complex
//and passing an object of the wrong type will throw an exception,
//potentially uncovering a serious bug.
public int compare(Object o1, Object o2) {
    final String time1 = (String)o1;
    final String time2 = (String)o2;
    return Timestamp.valueOf(time1).compareTo(Timestamp.valueOf(time2));
}
```

### *Generic AquaLogic Commerce Services exceptions by layer*

When a new exception type is not warranted, throw an existing exception corresponding to the application layer. The following exception types are available.

- EpPersistenceException
- EpServiceException
- EpDomainException
- EpSfWebException

### *Uncaught exceptions in the Web layer*

A list of ordered HandlerExceptionResolver objects can be configured in the web layer with Spring MVC. Make sure the most generic one, namely EpSystemExceptionHandler.java, has the lowest order number. This handler will catch all uncaught exceptions in the web layer and redirect the user to the general error page where the exception is displayed.

# Search and Indexing

This section explains how to interface with the Lucene search engine which underlies all search capabilities in AquaLogic Commerce Services.

It also introduces Solr, a freely distributable searching scheme based on Lucene, which handles all forms of search in AquaLogic Commerce Services as of release 6.0, including search in both the storefront and the Commerce Manager client.

Finally, this section discusses using the Lucene index viewer, Luke, to view the AquaLogic Commerce Services Solr indexes.

## *Lucene Search Engine and Indexing*

### How to use Lucene search

1. Create and populate a SearchCriteria bean. There are 4 types of search criteria: ProductSearchCriteria, CategorySearchCriteria, CustomerSearchCriteria, OrderSearchCriteria
2. Inject an indexSearchService and call the following method of the service.

```
/**
 * Searches the index with the given search criteria.
 *
 * @param searchCriteria the search criteria
 * @return a list of object uids which match the given search criteria
 */
    List search(SearchCriteria searchCriteria);
```

3. Load objects based on your needs

The search will return the UIDs of the objects that can be used to load them. However, you many only need to load the top 100 products of the returned 5000 product UID list.

You can use the following method in ProductService to load products by their UIDs.

```
/**
 * Returns a list of <code>Product</code> based on the given uids.
 * The returned products will be populated based on the given load tuner. If a
 * given product Uid is not found, it won't be included in the return list.
 *
 * @param productUids a collection of product uids
 * @param loadTuner the load tuner
 * @return a list of <code>Product</code>s
 */
List findByUids(Collection productUids, ProductLoadTuner loadTuner);
```

### Search fields and index fields

| Object | Search Fields | Index Fields | Locale | Comment |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| product(SF) | keyword | product display name; brand display name; sku code; attribute value which are defined indexable | all locales | If you give multiple keywords, like: keyword1 keyword2. The search result will products whose index fields contain BOTH keyword1 and keyword2. |
| product(SF) | active only | start date & end date | n/a | product is active if : start date <= now <= end date |
| category(CM) | keyword | category display name, code | system default locale | |
| category(CM) | active only | start date & end date | n/a | category is active if : start date <= now <= end date |
| category(CM) | inactive only | start date & end date | n/a | category is inactive if : now <= start date or now >= end date |
| product(CM) | keyword | product display name, code, sku code | system default locale | |
| product(CM) | active only | start date & end date | n/a | product is active if : start date <= now <= end date |
| product(CM) | inactive only | start date & end date | n/a | product is active if : now <= start date or now >= end date |
| product(CM) | brand code | brand code | n/a | |
| product(CM) | category uid | all ancestor categories uids | n/a | |
| customer(CM) | first name | customer first name | n/a | |
| customer(CM) | last name | customer last name | n/a | |
| customer(CM) | customer number | customer number(uidPk) | n/a | |
| customer(CM) | email | customer email | n/a | |
| customer(CM) | phone number | phone number | n/a | |
| order(CM) | order number | order number | n/a | |
| order(CM) | order status | order status code | n/a | |

| | | | | |
|---|---|---|---|---|
| order(CM) | sku code | all sku codes in an order | n/a | |
| order(CM) | order from date | order create date | n/a | where start date <= create date |
| order(CM) | order to date | order create date | n/a | where create date <= to date |
| order(CM) | order shipment zipcode | all order shipments zipcode | n/a | |
| order(CM) | order shipment status | all order shipments status code | n/a | |
| order(CM) | first name | customer first name | n/a | |
| order(CM) | last name | customer last name | n/a | |
| order(CM) | customer number | customer number(uidPk) | n/a | |
| order(CM) | email | customer email | n/a | |
| order(CM) | phone number | customer phone number | n/a | |

For an object search, if multiple search fields are given, it's always an AND relationship among all of them.

## Index builder

The index builder constructs the index that Lucene uses to execute searches. The index builder is invoked by a Quartz scheduled job configured in quartz.xml. The index builder uses properties from buildIndex.properties, which is described in the following section. There are several index builders that build indices for different entities and their index files are named according to the pattern xBuildIndex.properties where x is the name of the entity being indexed such as a customer or product.

## Key classes and files

The following files and Java classes are also related to the Lucene search index support.

**buildIndex.properties**

- Contains the following two properties.

  - Rebuild - set to true to create a new index. This provides a way to create a new index when the server is already started. The default value is false and the value is also set to false after a successful index build/update.
  - LastBuildDate - the date when the last build was executed successfully.

**IndexBuildService**

- This service class handles the building of the Lucene index as well as updating and deleting documents from the index.
- provides two key methods.
  - void buildIndex(final boolean rebuild) - If rebuild is true, creates a new index, if rebuild is false, updates the existing index.
  - void buildIndex() - Checks the following conditions to determine whether to create a new index or update an existing index.
    - if the Rebuild property in buildIndex.properties is set to true, calls buildIndex(true)
    - if LastBuildDate property in buildIndex.properties is null or empty, calls buildIndex(true)
    - else calls buildIndex(false)

**PropertiesDao**

- This DAO reads and saves properties files.
- getPropertiesFile(buildIndex) - Retrieves buildIndex.properties.
- storePropertiesFile(buildIndexProperties, buildIndex) - saves the Rebuild and LastBuildDate properties to buildIndex.properties

**ProductService**

- This service class handles product retrieval.
- list() - Returns list of all products, this is used to create a new index.
- findByModifiedDate(lastBuildDate) - Returns a list of products whose modified date is after the last build date, this is used to update index.
- findByDeletedDate(lastBuildDate) - Returns a list of deleted products whose deleted date is after the last build date, this is used to update index.

**IndexDao**

- This DAO creates new indices and updates/deletes documents from existing indices.
- createIndex(documentList, locale) - Create a new index with the passed in list of documents.

- updateDocumentInIndex(documentList, product uid key, locale) - Update a document in an existing index.
- deleteDocumentInIndex(termsList, locale) - Delete a document in an existing index.

## Related Code

The following packages contain code related to the Lucene search.

- com.bea.alcs.domain.search.*
- com.bea.alcs.service.index.*
- com.bea.alcs.persistence.IndexWriter
- com.bea.alcs.persistence.Searcher

## *Search and Indexing with Solr*

Solr, based on Lucene (http://lucene.apache.org/), a freely distributable searching scheme, handles all forms of search in AquaLogic Commerce Services, including search in both the storefront and the Commerce Manager client.

Prior to release 6.0, each AquaLogic Commerce Services storefront and Commerce Manager created their own indexes which they searched. With Solr, a single server builds these indexes and searches them, centralizing indexing and searching efforts and allowing for more scalable search functionality.

Solr also provides a variety of request handlers that allow you to provide different search behaviors with the same server. For example you can both update and perform many different types of searches all on the same server because it provides these functionalities as different request handlers.

There also exists a schema for the index which allows you to define different field types for documents. Having field types also allows one to have different analysis for each type of field. One of the things that Solr provides out of the box is multiple analysis's for each type of field. Better, yet they also distinguish between indexing and querying.

Solr configuration happens within an XML file, which provides for easy tuning of the search server.

## *Solr Search Server*

Solr runs on a self-contained search server. Searching has been broken down into five categories: customer searches, order searches, category searches, product searches, and promotion searches. Each type of search is index separately, similar to the previous implementation. Each of these has been separated into a Solr core. In order to query two or more of these categories, sequential calls must be made to the Solr server.

## Solr Cores

Solr cores were an essential part of the move to Solr because different indexes required different cores. In our implementation, cores are essentially created as different servlet filters. Unfortunately, this isn't the direction that Solr is actually going and may require some change in the future; they are aiming for cores to be configured via XML.

In AquaLogic Commerce Services, the class that handles the distribution of SolrServer's in the system is DefaultSolrManager. This class has the capability to produce both embedded servers and HTTP servers, but due to a file system constraint, only HTTP servers are used.

You can think of the servers that DefaultSolrManager gives out as managed cores. Unmanaged servers that are passed to this class could cause problems in some of its methods.

## Configuration

There are two types of configuration that one needs to be aware of for the current implementation: Solr configuration and AquaLogic Commerce Services search configuration.

Solr configuration is primarily in contained under the /conf directory within Solr's home directory (under WEB-INF/solrHome in the Commerce Server directory). Configuration is broken up into schema and configuration files.

Schema files define which fields that are stored and indexed as well as definitions of field types themselves. Config files contain physical search and update handlers as well as default boost values for fields (there are defaults if not specified there). For more information on Solr configuration, check out the Solr wiki.

At the time of writing, Solr cores must be explicitly specified within the servlet config file. Discussion around the Solr cores issue may lead to creating these cores dynamically, but this is not the case for the time being.

AquaLogic Commerce Services configuration is self-contained within a separated file named search-config.xml. This file houses all configuration details for searching and is for the moment and is not required. Defaults are used if the configuration file is missing/misnamed or doesn't not contain all the appropriate sections. This file houses details such as number of return results and accuracy of spelling suggestions for each type of search. More importantly, this file houses boost values for each field of each of the different types of searches. Having these sections allows one to put more emphasis on a product name and less on its attributes for example.

The AquaLogic Commerce Services configuration file is housed within two locations, the Store Front and the Commerce server. Commerce clients will all receive the file via the Commerce server the same way it receives other configuration properties. Access to the search configuration is available through the elastic path instance via the getSearchConfig(String) method.

## Solr Java Client

Solr Java client is a feature in development for version 1.3 of Solr. It is used within AquaLogic Commerce Services to hide the details of parsing results returned from Solr, as well as providing a common interface for updating it.

## SolrManager

SolrManager is the central class involved in distributing and managing Solr servers. It distributes server instances to both search classes and indexing classes. This class also wraps common operations to prevent simple mistakes, which should be used rather than managing these operations yourself. There are two types of servers that this class distributes, an embedded server and an HTTP server. Both inherit from the same interface making operations cross server compliant.

## Searching

All searching classes receive HTTP servers from SolrManager. Each search category has the ability to build spelling indexes, but are only able to use those indexes if the search criteria inherits from SpellSuggestionSearchCriteriafor simplicity of spelling suggestions. Spelling indexes also aren't kept up to date with the indexes, they have to be manually rebuilt before using them. This is handled automatically by quartz jobs after new items have been indexed.

## *Indexers*

The indexers support two types of updates: incremental updates and full rebuilds. Generally this will always be incremental builds. Incremental builds run on a quartz job every so often. To know which objects/UIDs to update, objects tend to keep a last modified date to keep track of changes. The algorithm for checking if an object has changed is quite primitive as the only state it keeps is the last time the indexes were built. So setting an objects last modified date in the future will make that object continuously be reindexed. The indexes keep track of the last time that they were built in a property file which is read/written every time the quartz job triggers. For instance, the product index stores the last time it was built within a property file called productBuildIndex.properties. In AquaLogic Commerce Services, these files are located under WEB-INF/conf/resources. There are similar property files for each type of index.

The property files also contain a rebuild flag in order to rebuild the entire index. Rebuilding the index will clear the index for you. Either setting the rebuild flag or clearing the date (or both) will trigger a full rebuild.

Each time the indexer is run, it goes through these basic lifecycle events:

1. Check the DB for modifications
2. If some, get the UIDs of the objects modified
3. Create a new updated document for the UID for each added or modified object

4. Send the updated document to Solr
5. Send the command to delete removed objects
6. Notify build listeners and update property files

A modification event can either be from these last modified dates explained above or from a notification. On top of that, these can either be additions, modifications or deletions. Deletions always happen last so that if an object is both marked for update and deletion, it will be correctly deleted. This can also be seen as a limitation of the system as you if you wanted to "undo" the process, you can't do it immediately, and you would have to wait until after the item has been deleted.

One thing the indexers make use of extensively is OpenJPA's fetch groups. These do may testing harder as current tests won't show these types of breakages. Fetch groups allows one to finely tune a result so that only the required information is retrieved. These are necessary for the general case, but the LDD needs them to function properly.

## *Asynchronous Index Notification*

The index sub-system has an asynchronous notification system that can be utilized to notify the indexes of important milestones. The initial design was to allow the database to act as a message store for the indexes. At this time, the database is used as a queue, so once notifications are processed, they are not able to be accessed again.

Currently there are three types of notifications:

- General commands
- Specific/Container UID matchers
- Index Queries

General commands are fairly obvious and affect all objects; they should not require a specific UID or object type to act upon. These, in essence, are special container UID matchers. Generally, this should only be deletion of all indexes and rebuilding all indexes.

Specific/Container UID matchers are where things get complicated. These types of notifications require a UID and object type to process. In the simplest case, the object type is the type the index is working with (or a SINGLE_UNIT entity type). The indexers can also extend this functionality to process a specific entity type (a container or perhaps related UID). For example, the product index can handle the affected UID of a store. This would be a container UID matcher where the indexer knows that this store UID really means all products within the store.

Index queries are the second main way to send notifications. These were created because it is sometimes slow to use direct DB queries for large container UID matchers. These notifications use the index service that we've built to find the desired UIDs. All that's required for this type is search criteria that we use for every other type of search. One thing to note about this type of notification is that because its asynchronous, it may not have the same results that you think it should. For instance, the index could have

been updated after you construct the search criteria and your criteria might not cover this new/updated object.

## *AquaLogic Commerce Services Search Components*

This section explains the different search components involve in a search. Storefront searches and Commerce Manger search are fairly different in regards to what happens under the hood, but they still retrieve results in a unified way. Every type of search has the following general format:

1. Create a search criteria for your search (any of the search criteria's can be used)
2. Use the IndexSearchService to perform the search
    1. Sends your search criteria to the SolrIndexSearcherImpl
    2. Passes required information (search configuration and criteria) to the SolrQueryFactoryImpl to construct the SolrQuery
    3. SolrIndexSearcherImpl then processes the results by reading the UIDs of the results and parses any facet information present in the result set
    4. SolrIndexSearcherImpl then sends the result back to the client
3. Get the results

The SolrQueryFactoryImpl distributes on what type of search you are doing based on the search criteria. Most searches will distribute to the appropriate Query Composer for the construction of the Lucene Query which is used as the Solr Query. This also adds some additional Solr information such as filter queries, number of results, sorting and such. The one special case is the KeywordSearchCriteria. This criteria has no query composer therefore requires the query work to be done elsewhere. This criteria creates a query that sends results to a Solr disjunction max request handler which does the work of constructing the query based on the fields that you give it.

In general, all searches use Solrs ability to page results and perform additional searches as new pages are requested. This allows us to get the objects and forget about them rather than keeping and maintaining a list of objects for each page, especially as the number of results increase.

The Commerce Manager uses a search request job for all searches that it performs. Generally you will be using the request job in one view and a listener listens for results to be returned in another. The request job is passed along with the listener update which should be used in the resulting view. A single reference to the request job should be maintained as the request job handles the changing of search criteria's and maintaining of the result set so that a search can be performed again on a different page.

Storefront searches (including browsing) go through a service which provides a bean given to the templates. Browsing and searching are performed in a very similar fashion with page 0 indicating you want all results (unpaginated). Browsing has to handle a special case where we are at a root level category and searching has to perform fuzzy and spelling suggestions. Both of the searches first perform a search for featured

products and then a spefic search for the items in that section (a category for browsing and a keyworded search for search).

One thing to note about any of the *search* components (not indexing): they all have locale fallback. For instance, if you search for an item in en_US, you will also be searching for the data in en as well as en_US. This same concept can be seen in search configuration for boost parameters. Boosts can specify a locale of en but can be overridden by values in en_US, or in other words, a search in en_US would fall back to en and finally to no locale if it didn't have any boost values set for en_US or en.

### Solr Searching

Solr searching is where the bulk of operations go down. This component is essentially the classes SolrIndexSearcherImpl, SolrQueryFactoryImpl and SolrFacetAdapter. The SolrIndexSearcherImpl distributes the work of building the query to SolrQueryFactoryImpl. It also processes the sends the query off to Solr using the core that should be used for the search criteria given and parses any facet information within the result set.

SolrQueryFactoryImpl is where the construction of the Solr query happens. This construction is generally request handler specific (we have separate construction mechanisms for each of our spell checker request handler, store dismax search and general searches). This class also distributes some of the filter/facet work to SolrFacetAdapter.

## Indexing/Searching Utilities

There exists a utility class for performing general functions relating to Solr. For example the IndexUtility class can perform methods to transform an Attribute object into a solr field or provide the ability to sort a list of objects with their respective UID list retrieved from Solr.

## Solr Request handlers

Nothing specific to AquaLogic Commerce Services happens in the request handlers, but some have been extended in order to operate in our system. For instance, the disjunction max request handler was updated so that it would accept fuzzy searches as well as regular searches. The spell checker is another, it was extended so that all locales weren't stored in the same 'index'.

### Search Configuration

All search configurations is defined in a file called search-config.xml which lives in each WEB-INF directory. Parsing of these configuration files is done by leeching off the AquaLogic Commerce Services setup in ElasticPathDaoXmlFileImpl. The original parsing design was to replace the XML parsers in ElasticPathDaoXmlFileImpl to distribute to spring injected Xml Parsers which would allow one to easily add new configurations to AquaLogic Commerce Services via XML. Unfortunately there were some minor hiccups moving some of the existing code to these XML parsers so only the search config is done via these spring injected parsers.

The XML parsers are in charge of setting up SearchConfig objects which are where all search configuration settings are kept. The configuration design specifies defaults that can be fallen back on for all configuration settings. This includes getting the search configuration itself. Attempting to get a search configuration that doesn't exist will retrieve a default search config object that has all defaults.

On the Storefront, search configuration is reloaded with a quartz job so that changes can be reflected immediately. The Commerce Client gets a new configuration every time it starts up so that the Commerce Manager server doesn't need to be restarted.

### Search Criteria

Most search criteria are pretty self explanatory, but there are two which are special: LuceneRawSearchCriteria and FilteredSearchCriteria.

LuceneRawSearchCriteria allow you to harness the full power of the Lucene API very quickly. They are also used in the indexers to reparse query notifications.

FilteredSearchCriteria allow you to not only nest search criteria (even nest a filtered search criteria within itself), but also use a search criteria to filter out results from the original search criteria and AND/OR the results of multiple search criteria. To see the filtered search criteria in action, check out CatalogPromoQueryComposerHelper.

### Query Composers

The query composer is the bridge between the AquaLogic Commerce Services search API and the Lucene API. There are many helper methods which allow for easy extension of simple fields, but this doesn't cover all the bases.

All search criteria have a match all flag which overrides any of the parameters in the search criteria.

## *Debugging Search*

The easiest way to debug a search is to send a search directly to the search server yourself through a browser. If JDK logging is set to info or above, Solr will output any query it receives to the log. This query can be taken directly from the log and send to Solr. If you are trying to debug field data, your best bet it to enable storage of all fields and reindex. If you are in need of the fields and don't might seeing internal data, you can use faceting to reveal these fields for you by using a facet.field query. This isn't always helpful though as it doesn't tell you which document the data came from.

If you are sending manual queries to the server, *be sure the server is started and you are sending them to the correct core!*

One quirk about the query parser is that the query

```
field:one query
```

will actually represent the query

```
field:one defaultField:query
```

where defaultField is the default field. To do this type of search, one must enclose the query in quotes:

```
field:"one query"
```

## Query Parser Syntax

The general query parser syntax is:

```
<field_name>:<query>
```

The query shouldn't contain any special characters. If special characters are required, they should be escaped using backslashes (). Queries aren't exactly boolean, but can be translated to booleans:

```
field1:here && field2:there || field3:back ==> +field1:here +field2:there field3:back
```

Lucene has the concept of *must*, *must not* and *could* which can be used to construct a boolean query. Combined with brackets, any type of query can be accomplished.

There exists a special query which will match all documents:

```
*:*
```

Keep in mind that *:here will blow up and field:* is a different query entirely.

For a more advanced explanation of the Lucene query parser syntax, see http://lucene.apache.org/java/docs/queryparsersyntax.html. Keep in mind, the only modification that EP has is mixed range queries, i.e. the syntax field: {0 TO *] and field:[0 TO *} are valid.

## Lucene Locks

Lucene locks are bound to happen once in a while during development. The reason that they pop up is because the server was unexpectedly stopped while the indexers were running a job.

To remedy the problem, you have to figure out the index that is complaining; generally this can be seen easily enough in the console. If not, you will have to go into each index directory and search for a file called lucene-*.lck, where the *

part is dependent on the machine you are working on. Deleting this file will fix the problem.

The presence of this file should indicate to you that you should reindex, but because the indexers only disregard items after it knows everything has been built, it should pick up these objects again and re-index them.

### Reindexing

In the case where you need to manually reindex, just remember that indexes store the last time they were built in a property file and this file is written and read each time the quartz job triggers. All that needs be done is change the last build date or set the rebuild flag. Doing both will have the same effect. Wiping the last build date also has the same effect.

> In most environments, the above will work, but, in some cases, you may be required to stop your server before you change the property files.

One thing to note is that doing only this (changing the date) merely overwrites the current indexes. Wiping the date or setting the rebuild flag will clean the indexes automatically. If you wish to start from a clean state, you will have to stop your server, delete the index data directory WEB-INF/solrHome/data, change the property files and start the server up again. The index directories will automatically repopulate themselves.

Sometimes it is advantageous to do a clean on the indexes as a build won't delete old objects.

### Matching Everything

Generally you will have at least 1 criterion to search upon and won't require the match all functionality, but in the case that you do, be aware that this overrides all other queries flags and parameters. Be sure to check for this as it may be the cause of problems if you are using it and forget to clear it.

## *Tutorials*

This next section walks through a few tutorial circumstances for index changes.

### Add a new field

Adding a new field should be easy. There are 4 things we need to do to add a new field:

- Update Solr index schema
- Update the search criteria (optional - generally this *is* required)
- Update the search criteria query composer
- Update the indexers to index the field (optionally - generally this *is* required)

Let's add the field Product SKU UID to the product search. First we need to extend the search criteria to store this Product SKU UID.

### ProductSearchCriteria.java

```
private Long productSkuUid;


public Long getProductSkuUid() {
   return productSkuUid;
}


public void setProductSkuUid(final Long productSkuUid) {
   this.productSkuUid = productSkuUid;
}
```

Why use a Long object? This is generally up to the developer, but I prefer it because if you are using it, you know you are using it. If you wanted to clear it, you know how to clear it. This technique is especially useful if you wanted a tri-state boolean flag for example.

Now we need to update the Solr index schema. There are many field types that we could use, but let's use a SortableLong.

### product.schema.xml

```
<schema ...>
   <types>
      <fieldType name="slong" ...>
   </types>


   ...
   <fields>
      ...
      <field name="productSkuUid" type="slong"
         indexed="true" stored="false"/>
      ...
   </fields>
</schema>
```

Now lets update the product query composer:

### ProductQueryComposerImpl.java

```
public Query composeQueryInternal(...) {
```

```
    ...
    hasSomeCriteria |= addWholeFieldToQuery("productSkuUid",
        searchCriteria.getProductSkuUid(), booleanQuery, Occur.MUST, true);
    ...
}

public Query composeFuzzyQueryInternal(...) {
    ...
    hasSomeCriteria |= addWholeFuzzyFieldToQuery("productSkuUid",
        searchCriteria.getProductSkuUid(), booleanQuery, Occur.MUST, true);
    ...
}
```

Make sure you don't forget to add the field to the composeFuzzyQueryInternal() method as well. You should be familiar with the addWholeFieldToQuery() method; it is a very helpful helper method in the parent class. It doesn't come without limitations though, so don't worry about creating a customized routine like others that are in the class. Now let's update the indexer:

### ProductIndexBuildService.java

```
public SolrInputDocument createDocument(...) {
    ...
    final Set<Long> skuUids = new HashSet<Long>();
    for (ProductSku sku : product.getProductSkus()) {
        skuUids.add(sku.getUidPk());
    }
    addFieldToDocument(docment, "productSkuUid", skuUids);
    ...
}
```

Another helper method! And it takes a collection; how convenient! As in the query composer don't be afraid to mock up a routine to do the same thing as the helper method for advanced functionality. In the indexer, you will likely never need more advanced functionality than that provided by the helper methods.

That's it! Now all that's left is to reindex and you'll be able to use your new field.

## Add a new filter or storefront facet

The idea of a filter was initially a Storefront concept which allowed one to filter results. This has been extended to be included in searches as well. All search criteria's allow one to add a collection of filters to the search criteria. Of course not all filters logically

make sense with all types of search criteria, but this gives one an alternative to the FilteredSearchCriteria. The FilteredSearchCriteria approach is probably easier as it doesn't require a lot of modifications, but it's not always reasonable, i.e. you may not want to pollute your search criteria with irrelevant data.

The FilteredSearchCriteria method is to add all the data that you require to the search criteria itself and then update the query composer to reflect the new data. Now all one must do is use the filtered search criteria with another search criteria as a filter with the data you want to filter on. Just remember that using this method may not make logical sense with the data you're trying to filter on as explained in the FilteredSearchCriteria section.

The other is to actually use a Filter. The first thing that one must do is implement the Filter interface. Now we must determine if we need to show this filter on the Storefront (there are filters that exists solely for the purpose of being a filter and *not* being displayed in the Storefront). Regardless, we need to add the logic to handle the filter. The convention has been made to add this logic to the SolrFacetAdapter. The method that we need to modify is constructFilterQuery(Filter) to return a Lucene query for the given filter. This is where you have a lot of power since you have access to the Lucene API again. Generally these queries should be fairly simple, but there is nothing stopping you to take full advantage of the Lucene API. Unfortunately there are no helper methods to help construct the query as in the query composers.

This is where you are complete if you don't want to show the filter as a Storefront facet (faceting is currently only implemented for product searches). If you wanted to add faceting capability to the filter, you need to modify SolrFacetAdapter#addFacets(SolrQuery, SearchCriteria). Currently there is only the ability to include or exclude all facets and not select which ones you want. Faceting is really a Solr term which literally means counting up the terms. There are two types of facets that you can do. The easier, although not useful in all cases is the facet field. All you need to do is specify a field and viola, the field is faceted on all stored values. The second type is a facet query which allows you to specify a query to use to facet on.

After we've specified what we want to facet on, we need these facets in the Storefront. This is where SolrIndexSearcherImpl comes along. This is the class that parses this facet information into separate fields in the SolrIndexSearchResult object for storage. Unfortunately, parsing facet information will increasely get more complex to separate out as you add more facets. It is already slight complicated trying to distinguish between the facets that we have.

Once SolrIndexSearcherImpl parses the facet information and places it in SolrIndexSearchResult (this is just where all the other facet information is), we can get access to it from anywhere we do a search. To add these new filters to the side menu that you see when you're browsing, you'll need to alter AbstractCatalogViewServiceImpl to pass the filter options to the CatalogViewRequest so that we have access to them in the templates. Now all that we need to do is alter the sidemenu.vm template file to use new filter options.

There are a few complexities that have only been grazed, but I leave it to the reader to resolve them.

## Add sorting

Now we will add a simple sort for products. Addition of different sorts can happen in this fashion, but will quickly lead to unmanageable code when there are numerous sort types. Let's add a sort field for UID. First we need to add a sort type to the SortBy enum:

### SortBy.java

```
public enum SortBy {
   ...
   UID("uid");
}
```

Now all that's left is to add a case to sort for that field in SolrQueryFactoryImpl. Remember that sorting is simply an additional field on the SolrQuery. There is already a method that applies the sort field to Solr, so let's just add to it:

### SolrQueryFactoryImpl.java

```
private void addSorting(...) {
   ...
   switch (...) {
   ...
   case UID:
      return "objectUid";
      break;
   ...
   }
}
```

That's it! Unfortunately sorting fields is almost completely dependent on how you index data. For instance, sorting on a field that has multiple values or tokenizes the field may produce invalid results. Tokenized fields/Multi-valued fields may sort based on the lowest/highest of any of the tokens/values.

## Adding a new type of search

The first thing that you should do is actually set up another core for the index. If you remember, each index has a separate core so that we can separate the schema a bit for each index.

The first thing that you need to do is set up another servlet filter for the additional core:

### web.xml

```
<filter>
```

```
    <filter-name>Order Sku Index</filter-name>
    <filter-class>org.apache.solr.servlet.SolrDispatchFilter</filter-class>
    <init-param>
      <param-name>core</param-name>
      <param-value>customer</param-value>
    </init-param>
    <init-param>
      <param-name>config</param-name>
      <param-value>orderSku.config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>schema</param-name>
      <param-value>orderSku.schema.xml</param-value>
    </init-param>
    <init-param>
      <param-name>path-prefix</param-name>
      <param-value>/orderSku</param-value>
    </init-param>
</filter>

...

<filter-mapping>
    <filter-name>Order Sku Index</filter-name>
    <url-pattern>/orderSku/*</url-pattern>
</filter-mapping>
```

To finalize the new core you have to add a config and schema files. Since these files are quite large, I will only mention that its best to copy and paste from an existing file. Using the existing naming convention, we would have <core name>.schema.xml or <core name>.config.xml. This process shouldn't be too difficult, but may require a little bit of knowledge of the Solr config and schema files. Here is a list of things that should change after the copy and paste:

- The name attribute of the document element of both the config and schema files (this should be the core name)
- Under <core name>.schema.xml the field list should change under the xpath /schema/fields** Change field names for explicit fields that are defined
    - Change dynamic fields for multi-fields that should be defined
    - The /schema/uniqueKey element

    - Any copy field under /schema/copyField
- Under <core name>.config.xml,
    - Remove any custom request handlers, generally you'll want to leave the update request handler and the standard request handler
        - You can optionally just mark these request handlers startup as *lazy*
    - Change the gettable files under /config/admin/gettableFiles to the config and schema we've just created

Once all that is done, you've created your Solr core and are ready to start using it. You can even upload some custom data yourself and query it directly now. Now let's set up the rest of the API to do the querying.

The first thing that you should do is create an indexer. These are generally quite painless as the bulk of configuration of the indexer happens in the abstract class. For most of the simple methods, you can just follow what other index files do (i.e. getting the name of the index, finding the modified/deleted UIDs, getting the index type, etc.).

One thing that you should remember to do is to define another property file in com.bea.alcs.core, as these are currently copied over in the build process and *not* created if they are not there and *do* throw exceptions if they are missing. Just follow the naming convention <core name>BuildIndex.properties and place it in com.bea.alcs.core/WEB-INF/conf/resources and the build process will automatically handle the copying. A custom name will require special modification of the build.xml file to copy the file over. The only thing other that you really need to do is to define the createDocument() method.

### OrderSkuIndexBuildService.java

```
protected SolrInputDocument createDocument(final long uid) {
   final OrderSku sku = orderSku
   if (sku == null) {
      return null;
   }
   final SolrInputDocument document = new SolrInputDocument();
   addFieldToDocument(document, SolrIndexConstants.OBJECT_UID,
String.valueOf(orderSku.getUidPk()));
   addFieldToDocument(docuemnt, "orderSkuCode", orderSku.getCode());


   // More fields to add here
   return document;
}
```

The one thing to note about this process is that the fields that you add should be the fields that you added to the schema. Make sure that the string value of the fields that you are adding match those types that are defined in the schema. The next thing one must

do to complete the indexer is to add a quartz job for it since all indexes run off a different quartz job. I assume that you know how to do that.

Next we need to create search criteria and a query composer. The search criteria in this case should contain merely getters and setters. Also due to the lack of complexity in our indexers (due to the lack of fields we are trying to index), our query composer will be just as simple.

### OrderSkuSearchCriteria.java

```
public class OrderSkuSearchCriteria extends AbstractSearchCriteriaImpl {
   private skuCode;

   public String getSkuCode() {
      return skuCode;
   }

   public setSkuCode(final String skuCode) {
      this.skuCode = skuCode;
   }
}
```

### OrderSkuQueryComposerImpl.java

```
public class OrderSkuQueryComposerImpl extends AbstractQueryComposer {
   protected boolean isValidSearchCriteria(final SearchCriteria searchCriteria) {
      return searchCriteria instanceof OrderSkuSearchCriteria;
   }

   public Query composerQueryInternal(SearchCriteria criteria, SearhConfig config) {
      final OrderSkuSearchCriteria searchCriteria = (OrderSkuSearchCriteria) criteria;
      final BooleanQuery booleanQuery = new BooleanQuery();
      boolean hasSomeCriteria = false;

      hasSomeCriteria |= addWholeFieldToQuery("orderSkuCode",
         searchCriteria.getSkuCode(), null, config, booleanQuery, Occur.MUST, true);

      if (!hasSomeCriteria) {
         throw new EpEmptySearchCriteriaException();
      }
   }
```

```
public Query composeFuzzyQueryInternal(...) {
    ...

    hasSomeCriteria |= addWholeFuzzyFieldToQuery("orderSkuCode",
        searchCriteria.getSkuCode(), null, config, booleanQuery, Occur.MUST, true);

    ...
}
```

Now that we have the search criteria's and query composers, we just need to add search criteria to the PrototypeBeanFactory as normal and the query composer to the QueryComposerFactory (do this in spring).

One more thing that you might want to do is update DefaultSolrManager so that the new index has a separate search configuration. This means that you'll be able to update search-config.xml with configuration and have it work.

That's it! All you need to do now is use the existing API to give our OrderSkuSearchCriteria to the IndexSearchService and we will get the correct results. Be sure to index something first before you do any testing or you won't get any results!

## *Inspecting Solr Indexes with Luke*

This section describes how to inspect the Solr indexes using the Luke tool. As Solr is built on Lucene, this means we can use the Lucene index viewer, Luke, to view the AquaLogic Commerce Services Solr indexes. This can be useful to check on the contents of an index, making sure a field is indexed as expected for example.

## *Step by Step*

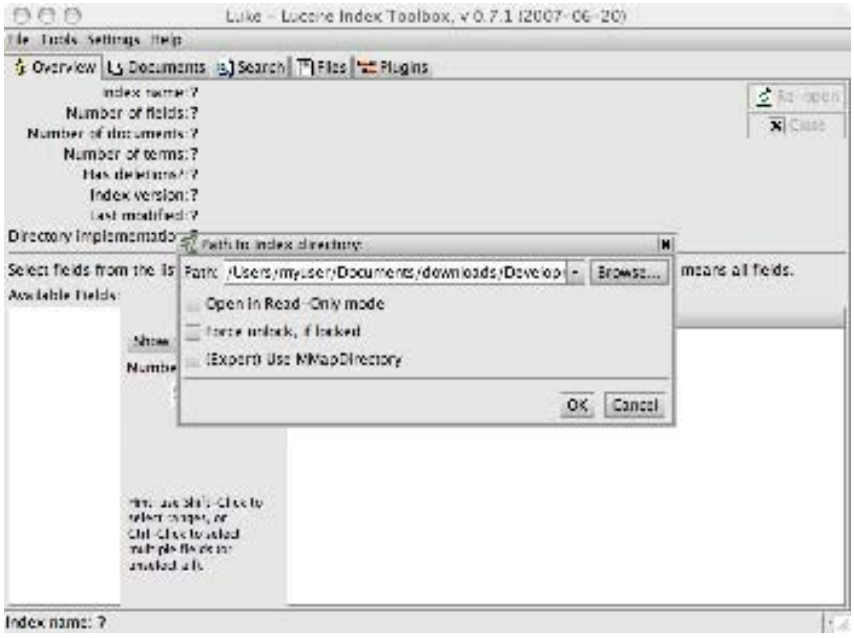
1. Download luke.jar from http://www.getopt.org/luke/
2. Run with the lucene-core jar file in EP_LIBS, this is needed so that Luke knows the format of the index files AquaLogic Commerce Services Commerce has written them in.

   ```
   java -classpath $LIB/lucence/core/lucene-core-2.3.574260-EP-dev.jar:luke-0.7.1.jar
   org.getopt.luke.Luke
   ```

   The ordering of the .jar files in the above line is important. lucene-core should come before luke.

3. You should now be presented with Luke's main window:




4. Navigate and select the directory that contains the Lucene index. The index directories will normally be below your search server's WEB-INF/solrHome/data/ with a directory for each type of information AquaLogic Commerce Services indexes.

# Internationalization and Localization

This section describes various mechanisms in AquaLogic Commerce Services that support multiple languages and currencies.

The AquaLogic Commerce Services Storefront can display content in multiple languages and currencies.

The AquaLogic Commerce Services Commerce Manager’s user interface operates in English only, but content can also be displayed in multiple languages and currencies.

## *Languages*

This document describes several issues and implementation mechanisms for multi-language support.

### Multi-language support mechanisms

There are four mechanisms used to support multiple languages in the Storefront. Each mechanism has characteristics that make it most appropriate for specific situations.

### Properties files

Properties files are used as the source of language-specific text. The properties files contain simple key value pairs where the key is a description of the String and the value is the corresponding text in a particular language. Multiple properties files with file names that indicate the language are used to support multiple languages. The language-independent keys in the properties files are used to retrieve the corresponding text in Velocity templates using a call to a Velocity macro as shown below.

```
#springMessage("productTemplate.itemsAvailable")
```

The Velocity macro is part of Spring's integration with Velocity, and it will look up the value of the key for the currently selected Locale. Properties files are generally created for each Velocity template and stored in the same location. The properties file has the same name as the template but has the extension ".properties" instead of ".vm". For Strings that are frequently used across multiple pages, the properties are stored in "globals.properties".

This multi-language mechanism is suitable only for static page content coded into Velocity templates.

### Localized Properties

The Localized Properties mechanism is a database-only solution that allows dynamic content such as a store's brands to be displayed in multiple languages. To use this mechanism, a domain object with display values in multiple languages has a reference to a LocalizedProperties object containing the language values retrieved by key and locale. All localized property data for all objects is stored as rows in the TLocalizedProperties table. It is not necessary to change the database schema to add new localized properties. See "How to add Localized Properties to a domain object" below for instructions on adding Localized Property support to a new domain object.

### Locale Dependent Fields

Locale Dependent Fields is used programmatically in a similar way to Localized Properties. An object has a reference to a LocaleDependantFields (LDF) object, from which it can retrieve localized Strings. In this mechanism, the localized properties are stored in columns in the database table and the table can be joined directly with the parent entity table. This means that performance is better than Localized Properties and the field values can be accessed using methods instead of a map key. However, the disadvantage of this approach is that a schema change is required for each new kind of localized field and a new table is required to support Locale Dependent Fields for each new object that requires them. For this reason, Locale Dependent Fields is only used for performance-critical domain objects such as products and categories.

### Attributes

The attribute system in AquaLogic Commerce Services also supports multiple languages. Attributes have the advantage that they can be created and maintained by a

business user through the Commerce Manager user interface. Attributes are relatively slower than Locale Dependent Fields and are only supported for Products, Categories, and SKUs.

## How to add new properties files

Spring framework provides messaging (i18n or internationalization) functionality by using MessageSource. When an ApplicationContext gets loaded, it automatically searches for a MessageSource bean named "messageSource" defined in the context.

In the Storefront, messageSource bean is defined in /WEB-INF/conf/spring/views/velocity/velocity.xml. The MessageSource implementation used in AquaLogic Commerce Services, ResourceBundleMessageSource, supports a hierarchical structure that defines where properties files should be defined. See the anotated XML code below.

```
<bean id="messageSource"
   class="org.springframework.context.support.
        ReloadableResourceBundleMessageSource">
  <!-- Define the parent message source bean -->
  <property name="parentMessageSource"> <ref bean="globalMessageSource"/>
  </property>
  <property name="basenames">
    <list>
      <!-- Define .properties files here (omitt the extension) -->
      <value>/WEB-INF/templates/velocity/account/create</value>
      <value>/WEB-INF/templates/velocity/address/create</value>
      . . .
    </list>
  </property>
</bean>

<bean id="globalMessageSource"
   class="org.springframework.context.support.
         ReloadableResourceBundleMessageSource">
  <property name="basenames">
  <list>
    <!-- Global .properties files are defined here -->
    <value>/WEB-INF/templates/velocity/globals</value>
    <value>org/acegisecurity/messages</value>
  </list>
```

```
  </property>
</bean>
```

With this hierarchical structure, the property definition in the files associated with messageSource bean will overwrite the definiton in the files associated with globalMessageSource, if there is any.

To avoid property key duplication among the files in the same hierarchy, property keys are prefixed with the file name. For example, the content of /WEB-INF/templates/velocity/catalog/product/productTemplate.properties will look like the following.

```
productTemplate.zoom=Zoom In
productTemplate.addToWishlist=Add to Wishlist
productTemplate.qty=Qty
productTemplate.itemno=ITEM NO
productTemplate.accessories=Optional Accessories
productTemplate.warranties=Protect this item
productTemplate.upSells=Upgrade to
...
```

## *Currencies*

AquaLogic Commerce Services supports multiple currencies, which are configured in the Commerce Manager client at the catalog and store level. Please see the Commerce Manager User Manual for information on how to configure these settings.

The currently selected currency is stored in the CustomerSession object, which is stored in the web application session.

## *Character Set Encoding*

AquaLogic Commerce Services supports multiple character set encodings. This document describes basic encoding defaults and outlines the changes that are necessary when changing the character set encoding.

### Default Encoding

The default encoding for various components of AquaLogic Commerce Services are as follows.

- Database data (UTF-8)
- Content (UTF-8)
    - html pages

    - emails
- Data files (UTF-16)
    - import data files
    - report files
- URL (UTF-8)
    - catalog browsing URLs
    - search URLs

## Changing the encoding

The following subsections describe how to change the encoding.

### How to change the encoding for database

You will need to specify the encoding when you create a database and give the same encoding in your JDBC URL.

For example, in MySql you will need to create a Database as shown below to use the Big5 encoding.

```
create database DB_NAME character set big5;
```

The corresponding JDBC URI is shown below.

```
jdbc:mysql://127.0.0.1/alcs?AutoReconnect=true&amp;
    useUnicode=true&amp;characterEncoding=big5
```

**Note:** You may encounter problems when using some Asian encodings that use a double wide character set, which makes some fields require twice as much storage space. In this case, the schema may need to be updated to accommodate the longer strings.

### How to change the encoding for content

1. Change the configuration "content.encoding" value in bea.alcs.xml.
2. Change the following settings in velocity.xml

```
<prop key="template.encoding">UTF-8</prop>
<prop key="input.encoding">UTF-8</prop>
    <prop key="output.encoding">UTF-8</prop>
```

3. Change the following settings in web.xml

```
<!-- Encoding filter -->
 <filter>
```

```
<filter-name>Encoding Filter</filter-name>
<filter-class>
   com.bea.alcs.commons.filter.impl.EncodingFilter
</filter-class>
<init-param>
  <param-name>encoding</param-name>
  <param-value>UTF-8</param-value>
</init-param>
  </filter>
```

### How to change the encoding for data files

Change the configuration "datafile.encoding" value in bea.alcs.xml.

### How to change the encoding for URLs

Changing the encoding for URLs is not recommended. However, the encoding is defined as a constant URL_ENCODING in WebConstants.java.

## *String Localization*

For an overview of string localization in AquaLogic Commerce Services, please see the Commerce Manager Localization section in the Architecture section of this document.

# Rules Engine

At the core of the promotion rule system is the JBoss Rules (formerly Drools Rules) library. JBoss Rules is a third-party rules engine that uses a fast algorithm to evaluate rule conditions and execute their actions. The input to the JBoss Rules engine is a set of objects used in the condition evaluation and action execution as well as the set of rules, which we express as text in the proprietary Drools language.

The representation of a Rule in AquaLogic Commerce Services is an object model of the components of a rule such as conditions, actions, and parameters used by various rule elements. The object model is persisted in the database directly with one table corresponding to one class in the rule object model. This allows the graph of rule objects to be easily stored, retrieved and modified, and stored again. The objects in the rule object model are responsible for generating Drools language code that is passed to JBoss Rules. The generated code is not persisted and it is not possible to re-create the object model representation of a rule given the Drools code.

The role of the Commerce Manager promotion editor is to allow the user to compose rules from rule elements and then store those rules in the database. The Storefront then retrieves the rules from the database as object graphs, requests the corresponding drools code from the rule objects, and passes the rule code to the JBoss Rules engine. JBoss Rules will then determine which rules' actions should be executed on the Java objects that are passed to it.

JBoss Rules support basic evaluation of objects' properties within the engine itself. However, more complex operations are not supported. In AquaLogic Commerce Services, nearly all conditions are evaluated in Java and the actions are also executed in Java. This allows the condition and action code to be easily debugged and unit tested. The PromotionRuleDelegate is the class that is responsible for computing conditions and executing actions as required by JBoss Rules.

# Scheduling

AquaLogic Commerce Services uses the Quartz scheduler to execute scheduled jobs such as computing product recommendations or building a Lucene index. The scheduled jobs are configured in Spring via the quartz.xml configuration file.

## *Adding a new job in quartz.xml*

To add a new scheduled job, perform the following steps.

### Define a job and set its properties

Define a job bean as shown in the code below and set the following properties.

- **targetObject** - the class that contains the logic for the scheduled job.
- **targetMethod** - the method name in the targetObject to execute.
- **concurrent** - set to false to prevent jobs from executing concurrently.

```
<bean id="newJob"
  class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
 <property name="targetObject">
  <ref bean="newJobService"/>
 </property>
 <property name="targetMethod">
  <value>executeMethod</value>
 </property>
 <property name="concurrent">
  <value>false</value>
 </property>
</bean>
```

### Define the time/trigger details and link the job

Define the trigger bean as shown below and set the following properties.

- **jobDetail** - the job to execute

- **cronExpression** - the cron expression to set how often the job should be executed

```
<bean id="newJobTrigger"
  class="org.springframework.scheduling.quartz.CronTriggerBean">
 <property name="jobDetail">
  <ref bean="newJob"/>
 </property>
 <property name="cronExpression">
  <value>0 0 0/1 * * ?</value>
 </property>
</bean>
```

### Add the trigger to the scheduler factory

Add the new trigger to the scheduler factory bean as shown below.

```
<bean id="schedulerFactory"
  class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
 <property name="triggers">
  <list>
   <!-- add the reference to the new trigger here -->
   <ref bean="newJobTrigger" />
  </list>
 </property>
</bean>
```

# Acegi – Security Framework

AquaLogic Commerce Services uses the Acegi security system to handle authentication, authorization, and HTTP/HTTPS switching. Acegi integrates with Spring to provide these services, which are configured in acegi.xml. Acegi is implemented as a chain of web application request filters that perform various security tasks. Acegi is therefore integrated with AquaLogic Commerce Services in web.xml, where the filters are declared. In case you're wondering, Acegi is pronounced "Ah-see-gee" the name has no meaning – it is just the #1, #3, #5, #7, and #9 letters of the alphabet.

For more information on acegi.xml configuration options, see acegi.xml in the deployment guide.

## *Acegi filters*

Following list of available Acegi filters are used in AquaLogic Commerce Services. The bean declaration and configuration for each filter can be found in acegi.xml.

- **channelProcessingFilter** - Determines which transport protocol to be used (HTTP or HTTPS).

The property filterInvocationDefinitionSource is used to specify which pages REQUIRES_SECURE_CHANNEL and which pages REQUIRES_INSECURE_CHANNEL as follows.

```
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
\A/billing-and-review.ep\Z=REQUIRES_SECURE_CHANNEL
\A/receipt.ep\Z=REQUIRES_SECURE_CHANNEL
\A/.password..ep.*\Z=REQUIRES_SECURE_CHANNEL
\A/sign-in.ep.*\Z=REQUIRES_SECURE_CHANNEL
\A/j_acegi_security_check.ep\Z=REQUIRES_SECURE_CHANNEL
\A/.*\Z=REQUIRES_INSECURE_CHANNEL
```

Note that the order of the entries is critical. The channelProcessingFilter will work from the top of the list down to the **first** pattern that matches the request URL. Accordingly, you should place the most specific expressions first (e.g. a/b/c/d.*), with the least specific (e.g. a/.) expressions last.

- **httpSessionContextIntegrationFilter** - Responsible for storing the SecurityContextHolder contents between invocations.
- **logoutFilter** - This filter processes logout requests, being triggered by a match with a URL configured in the filterProcessesUrl property (e.g. /sign-out.ep). This filter takes several constructor args - the first being the URL of the page to redirect to on successful logout, and the second being a list of logout handlers - an Acegi provided handler (SecurityContextLogoutHandler) performs a logout by clearing the security context and an ALCS logout handler clears the shopping cart from the session.
- **authenticationProcessingFilter** - This filter processes requests from the sign in form.
- **exceptionTranslationFilter** - Handles any AccessDeniedExceptions or AuthenticationExceptions thrown within the filter chain.
- **filterInvocationInterceptor** - Responsible for handling the security of HTTP resources. It requires an AuthenticationManager and an AccessDecisionManager.
    - AuthenticationManager - Retrieves UserDetails and validates the username and password.
    - AccessDecisionManager - Responsible for making the access control decision based on role voting.

The objectDefinitionSource property is used the configure the role based access control configuration as follows.

```
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
\A/manage-account.ep\Z=ROLE_CUSTOMER
```

```
\A/edit-account.ep\Z=ROLE_CUSTOMER
\A/create-address.ep\Z=ROLE_CUSTOMER
\A/edit-address.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/update-email.ep\Z=ROLE_CUSTOMER
\A/update-password.ep\Z=ROLE_CUSTOMER
\A/address-preference.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/order-details.ep\Z=ROLE_CUSTOMER
\A/checkout.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/shipping-address.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/checkout-address.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/delivery-options.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/billing-and-review.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/receipt.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
\A/printReceipt.ep\Z=ROLE_CUSTOMER, ROLE_ANONYMOUS_CUSTOMER
```

Note, again, that the order of the entries is critical. The filterInvocationInterceptor will work from the top of the list down to the **first** pattern that matches the request URL. Accordingly, you should place the most specific expressions first (e.g. a/b/c/d.*), with the least specifiec (e.g. a/.) expressions last.

Also note that pages in the customer self-service area cannot be accessed by anonymous customers because of the above role requirements.

## *Authentication process*

When a user signs in, Acegi performs the authentication. The AquaLogic Commerce Services sign-in HTML form sends the user's login information (a username and password) by posting to the /j_acegi_security_check.ep URL (configured in acegi.xml), which is intercepted by the filter, authenticationProcessingFilter. Acegi reads the username and password as form parameters with the pre-defined attribute names, j_username and j_password. The authenticationProcessingFilter references an instance of CustomerAuthenticationDaoImpl, which is an AquaLogic Commerce Services object wired into Acegi in acegi.xml. CustomerAuthenticationDaoImpl implements the Acegi interface, UserDetailsService, whose loadUserByUsername method is invoked by Acegi to retrieve customer information about the user who is signing in. The CustomerAuthenticationDaoImpl then retrieves and returns the Customer object corresponding to the given username (the user ID). The Customer object implements the Acegi UserDetails interface, which Acegi uses to query the password and account status for the user who is attempting to log in. If the password matches, Acegi will update the user's authentication status. Acegi uses a ThreadLocal SecurityContextHolder to store the SecurityContext between web requests. The SecurityContext contains a single getter/setter for the Authentication object that stores the user's authentication status.

EpAuthenticationProcessingFilter extends AuthenticationProcessingFilter to perform additional processing upon a successful login. The EpAuthenticationProcessingFilter

notifies the WebCustomerSession it references, which in turn retrieves the customer's CustomerSession object and stores it in the web application session.

### *Authorization process*

When a signed-in user attempts to access a page, the Acegi filterInvocationInterceptor filter will intercept the request and ensure that the user has sufficient permissions. The required permissions for a given page are specified in acegi.xml in as part of the filterInvocationInterceptor configuration. Acegi references the user's UserDetails object to call the getAuthorities() method on it and check that the user has the required authorities to access the page. An "authority" is an implementation of Acegi's GrantedAuthority interface, which has a getAuthority() method that simply returns a String. This String corresponds with the Strings in the filterInvocationInterceptor Spring configuration in acegi.xml. In AquaLogic Commerce Services, CustomerRole objects implement the GrantedAuthority interface and are wired with String authorities in the domainModel.xml Spring configuration file. However, Customer, which implements UserDetails, isn't directly associated with these roles. Customers have a collection of CustomerGroups that they belong to, and CustomerGroups have collections of CustomerRoles. Therefore the getAuthorities() method in CustomerImpl iterates over customer groups to get the collection of Roles (authorities) to return to Acegi.

## Plug-In Architecture

AquaLogic Commerce Services uses a lightweight plugin architecture based on Spring functionality. Using this architecture, you can easily create a plug-in that will work seamlessly with existing AquaLogic Commerce Services modules.

### *Spring out of the box*

You should already be familiar with how you can just change configuration in Spring bean context files. However, this alone does not make your module completely pluggable - changes still need to be made manually in the Storefront and/or Commerce Manager configuration files.

### *Auto Discovery*

Spring provides some functionality that makes it relatively easy to create pluggable components that can be auto-discovered. Auto-discovery of the code itself is easy - just add your code .jar file to the WEB-INF/lib directory of any application that requires it and the code will be automatically added to the classpath. But what about the configuration files?

The following statement in the Spring configuration file elastic-path-servlet.xml of the Storefront module will allow it to auto-discover the configuration file of any plug-in added to the classpath

```
<import resource="classpath*:META-INF/conf/plugin.xml"/>
```

Now any plug-in JAR file which includes a META-INF/conf/plugin.xml file will have that file auto-discovered!

## *Self Configuration*

The second addition required is the ability for the plug-in to self-configure - i.e. in a tight integration we will often want to change configuration parameters of existing bean definitions. Fortunately there is now a way to dynamically wire together beans without modifying the original configuration files.

### Introducing PluginBeanFactoryPostProcessor

The key piece of infrastructure which allows modules to be self configuring is a new class in the AquaLogic Commerce Services core module, com.bea.alcs.commons.util.impl.PluginBeanFactoryPostProcessor. This is an implementation of the Spring BeanFactoryPostProcessor interface, which Spring invokes after all of the configuration has been discovered and loaded into an in-memory representation, but **before the actual objects are created**.

For each property/bean you wish to override/extend, you would add a PluginBeanFactoryPostProcessor bean definition in your plugin.xml

Suppose we have the following bean definition in the original module

```
<bean id="authenticationProcessingFilter"
    class="com.bea.alcs.sfweb.filters.EpAuthenticationProcessingFilter">
  <property name="webCustomerSessionService">
   <ref bean="webCustomerSessionService"/>
  </property>
  <property name="authenticationManager">
   <ref bean="authenticationManager"/>
  </property>
  <property name="authenticationFailureUrl">
   <value>/sign-in.ep?login_failed=1</value>
  </property>
  <property name="defaultTargetUrl">
   <value>/manage-account.ep</value>
  </property>
  <property name="filterProcessesUrl">
   <value>/j_acegi_security_check.ep</value>
  </property>
</bean>
```

Here's an example of the PluginBeanFactoryPostProcessor class in use in a plug-in module's plugin.xml file to override the original bean definition

```
<!-- Override the authenticationProcessingFilter to use the WLS filter
    and add a customerService property -->
<bean class="com.bea.alcs.commons.util.impl.PluginBeanFactoryPostProcessor">
 <property name="extensionBeanName">
  <value>"authenticationProcessingFilter</value>
 </property>
 <property name="extensionClassName">

<value>com.bea.alcs.plugins.wls_authenticator.filters.WlsAuthenticationProcessingFilter</val
ue>
 </property>
 <property name="propertyName">
  <value>customerService</value>
 <property name="propertyValue">
  <ref bean="customerService"/>
 </property>
</bean>
```

The properties are as follows

| Property | Description |
|---|---|
| extensionBeanName | The name of the bean to find an override |
| extensionClassName | (optional) The class to set the overridden bean to use. This property can be omitted to leave the class as assigned in the original bean definition |
| propertyName | The property of the bean to override or add |
| propertyValue | The value for the above property |

Effectively, the above example has the same effect as manually editing the original configuration file like this

```
<bean id="authenticationProcessingFilter" class=
"com.bea.alcs.plugins.wls_authenticator.filters.WlsAuthenticationProcessingFilter"
>
 <property name="webCustomerSessionService">
  <ref bean="webCustomerSessionService"/>
```

```
</property>
<property name="authenticationManager">
  <ref bean="authenticationManager"/>
</property>
<property name="authenticationFailureUrl">
  <value>/sign-in.ep?login_failed=1</value>
</property>
<property name="defaultTargetUrl">
  <value>/manage-account.ep</value>
</property>
<property name="filterProcessesUrl">
  <value>/j_acegi_security_check.ep</value>
</property>
<property name="customerService">
  <ref bean="customerService"/>
</property>
</bean>
```

## Adding Functionality

Additional functionality can be easily added to this architecture by adding to the processing done in the PluginBeanFactoryPostProcessor.postProcessBeanFactory() method, for example you could easily add an extendList property to optionally allow a property value to add to a bean's list property rather than overwrite the list, or provide support for multiple properties in a <props> element.

## *Example*

For an example of a fully functional plug-in including self-configuration, see the WebLogic Authentication Plug-In, com.bea.alcs.plugins.wls-authenticator

## *Summary*

As you can see, using the above functionality to add auto-discovery and self-configuration means you can now create a plug-in module where all the classes and configuration are contained within a single jar file which can just be dropped in to the classpath - no additional configuration required! The plug-in module's plugin.xml file will configure any new beans and use PluginBeanFactoryPostProcessor beans to override classes/properties of beans already defined in the base modules.

# Email Capabilities

## *Functional description*

AquaLogic Commerce Services sends event-based emails that are triggered from a particular event taking place in the system. The event that triggers the email to be sent can come from a storefront, Commerce Manager Client or Server, and the Web Services application.

The following events are currently triggered to send emails:

- A new customer registers with a storefront
- Customer requests forgotten password
- Commerce Manager user requests forgotten password
- Customer password is reset (by customer or a Commerce Manager user)
- A new order is placed
- A shipment is shipped
- A customer requests that their wishlist be sent to a friend
- A gift certificate is purchased (the recipient is notified)

In addition to this list, it is possible to add new event-based emails.

All emails have a "text" version and may optionally have an HTML version. Each email event can use a different template and includes event-specific content (such as order number).

Email templates are store-specific, so a separate set of templates is needed for each store.

For bulk email marketing (i.e. sending a newsletter to thousands of customers), it is recommended that a third party system be used.

## *Technical description*

AquaLogic Commerce Services uses Velocity as the email templating system. This is the same templating technology used in the storefront application.

The standard JavaMail API is used to send event-based emails via an SMTP server.

**Note:** AquaLogic Commerce Services does not ship with an SMTP server. It is up to customers to set one up themselves.

Emails are sent as multi-part MIME messages, such that, if the email client cannot accept an HTML version, it will degrade to the text version.

Email services are included in the Commerce Manager Server application, so all applications that wish to send email at the point of a particular event trigger must make a call out to the com.bea.alcs.service.misc.EmailService interface in the Commerce Manager Server application. This service is exposed as a remote service via Spring's HTTP Invoker remoting protocol (the same protocol used to expose all Commerce Manager Server services).

# Appendix A

List of ALCS Web Services

| Service | API method(s) | Input(s) | Output |
|---|---|---|---|
| **ShoppingCartService** | getCart | userId (required), storeId (required) | list of cart items |
| **OrderService** | getOrders | userId (required), storeId (required) OR order search criteria | list of ord |
| **AssociationService** | getAssociations | productCode OR userId, storeCode, based-on (optional: order history and/or cart items), association types (optional), catalogCode | source product informatio along with list of associatio Each associatio contains type and list of targ products |
| **OrderService** | completeShipment | shipmentID (required), tracking #, capture funds (boolean), shipment date | success (boolean) with error message fails |
| **ImportService** | runImportJob | jobName (required), CSV filename (required), max allowable errors (optional) | List of rov from the import file that failed |
| **InventoryService** | getInventory | productSku code (required) | inventory counts (o hand, |

| | | | |
|---|---|---|---|
| | | | reserved, allocated available) |
| **InventoryService** | addOnHandInventory | productSku code (required), count (required) | success (boolean) with error message fails |
| **InventoryService** | subtractOnHandInventory | productSku code (required), count (required) | success (boolean) with error message fails |
| **AssociationService** | addAssociation | source product code (required), target product code (required), association type (required) | success (boolean) with error message fails |
| **AssociationService** | deleteAssociations | source product code (required), target product code (optional) | success (boolean) with error message fails |
| **CatalogService** | getProducts | flexible criteria including: product name, product code, sku code, brand, catalog, boolean: active-only (see commerce manager search screen). Additionally, need to allow the client tp specify how much product data to return. | by defaul returns product code, nar type, bra default category, price and active. Additiona client can specify a of: produc summary product price, product attributes SKU deta |

| | | | |
|---|---|---|---|
| | | | Product image, product S These optional fields are based on commerc manager product ta |
| **CatalogService** | addProduct | all product fields should be populatable | created product c |
| **CatalogService** | updateProduct | all product fields should be updatable; specify which field(s) (enum) are being updated | success (boolean) with error message fails |
| **CatalogService** | deleteProduct | product code | success (boolean) with error message fails |
| **CatalogService** | getSku | sku code | sku detail |
| **CatalogService** | addSku | all sku fields should be populatable | created s code |
| **CatalogService** | updateSku | as with updateProduct, all sku fields are updatable and user specifieds which field(s) are being updated | success (boolean) with error message fails |
| **CatalogService** | deleteSku | sku code | success (boolean) with error message fails |
| **CustomerService** | addCustomer | password, required | success |

| | | | |
|---|---|---|---|
| | | customer profile attributes & optional customer profile attributes. | (custome with error email alre exists or f formats incorrect. |
| **CustomerService** | authCustomer | email address/user ID & password | success (boolean) |
| **CustomerService** | getCustomers | flexible criteria, including customerID, email/userID, first name, last name, tel number, zip / postal code. Optional store filter. | Returns customer object, customer addresse |
| **CustomerService** | updateCustomer | as with the other update* services, most customer fields are updatable. The exception is CustomerID, which is mandatory. | success (boolean) with error message fails. |
| **OrderService** | cancelOrder | OrderID | success (boolean) with error message fails. |
| **OrderService** | holdOrder | OrderID | success (boolean) with error message fails. |
| **OrderService** | releaseHoldOrder | OrderID | success (boolean) with error message fails. |
| **CatalogService** | updatePrice | ProductCode/SkuCode, CatalogCode, PriceList | success (boolean) with error message |

| | | | |
|---|---|---|---|
| | | | fails. |
| **InventoryService** | setOnHandInventory | productSku code (required), count (required) | success (boolean) with error message fails. |
| **OrderService** | findProductCodesPurchasedByUser | order history search criteria | list of pro codes |

* indicates a high priority API that cannot be implemented now because the backend services have not been built yet.