ACTIVE COMMERCE®
TAKE BACK YOUR CUSTOMER.

CONFIGURATION GUIDE

SITECORE E-COMMERCE DONE RIGHT. ➡ ACTIVECOMMERCE.COM

Active Commerce Configuration Guide

Active Commerce Database Scaling

Taxes

Tax Calculator

Active Commerce tax calculators are managed under “Webshop Business Settings/Tax Calculation.” By default, on insert of a new Active Commerce Website, the site comes preconfigured with tax options for different tax calculation bases, including:

- North America Sales Tax with Shipping
- North America Sales Tax with Handling
- North America Sales Tax with Shipping and Handling

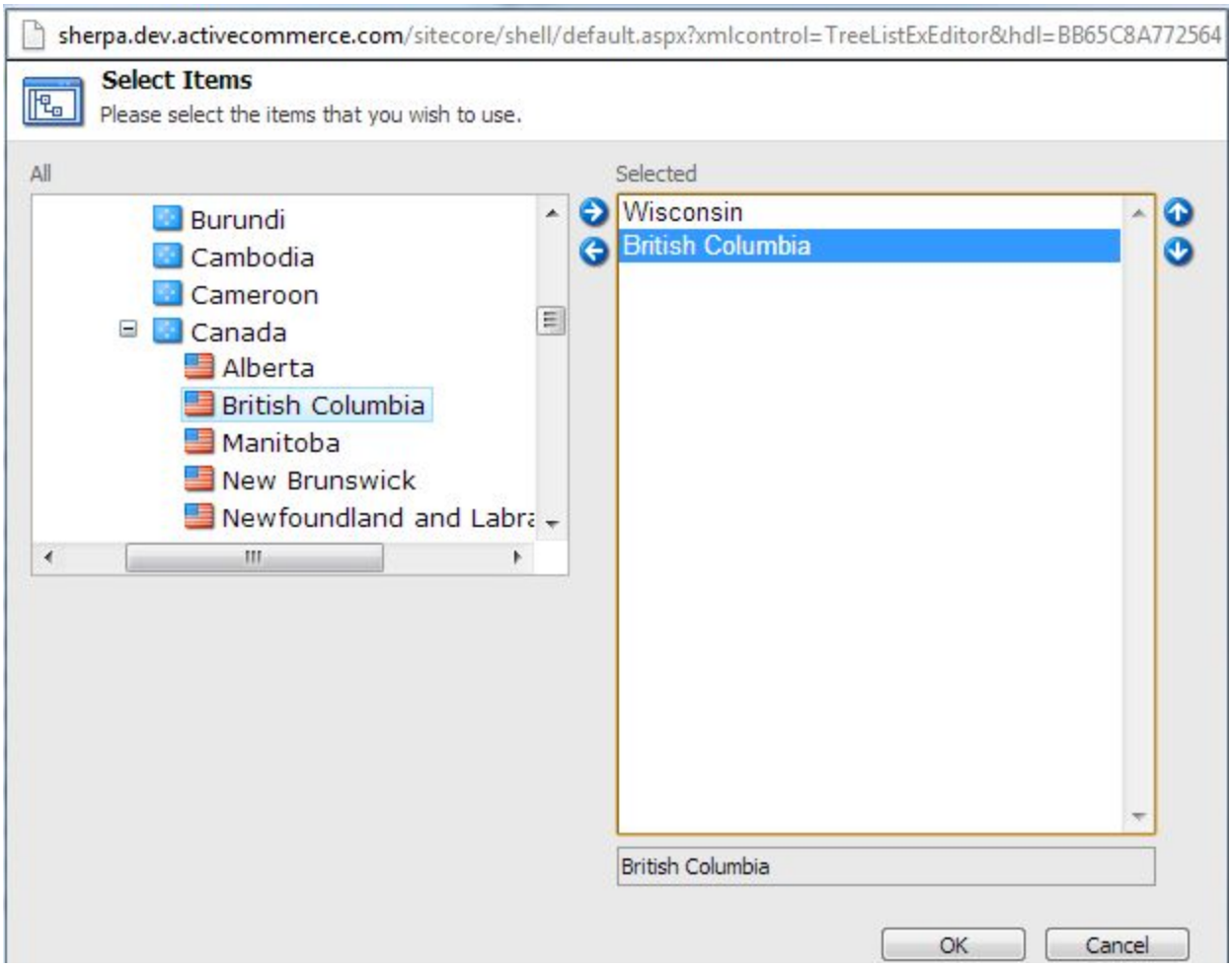
North America Tax Calculator

This calculator defines the following fields:

- **Code** – (Required) Must be unique. Used to identify the tax calculator.
- **Name** – (Required) The name of the tax calculator.
- **Title** – (Required) The title of the tax calculator.
- **Calculate Tax Based On** – (Required) Defines what the tax calculation is based on (combination of subtotal, shipping, and handling).
- **Locations** – (Optional) Defines which locations (country, region/state) this tax calculation should be used on. If no locations are selected here, this calculator is essentially inactive.
- **Tax Calculator** – (Required) Registered name of the ITaxCalculator class implementation. For North America, this should be left as “NorthAmericaShippingAddress.”
- **Rate Provider** – (Required) Registered name of the ISalesTaxRateProvider class implementation. Default is “Mock” (see *Rate Provider* section below for options).

Adding a new calculator

- In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Tax Calculation.”
- Insert item > North America Tax Calculator.
- Enter a name for the calculator and click OK.
- Enter a unique **Code**.
- Select the **Calculate Tax Based On** value.
- Select the **Locations** for which this tax calculation should be used.



- Enter the **Rate Provider** you wish to use.
- Save and publish changes. Verify that the appropriate tax calculations (per Location, Tax Based On, etc.) are now used on checkout.

AvaTax

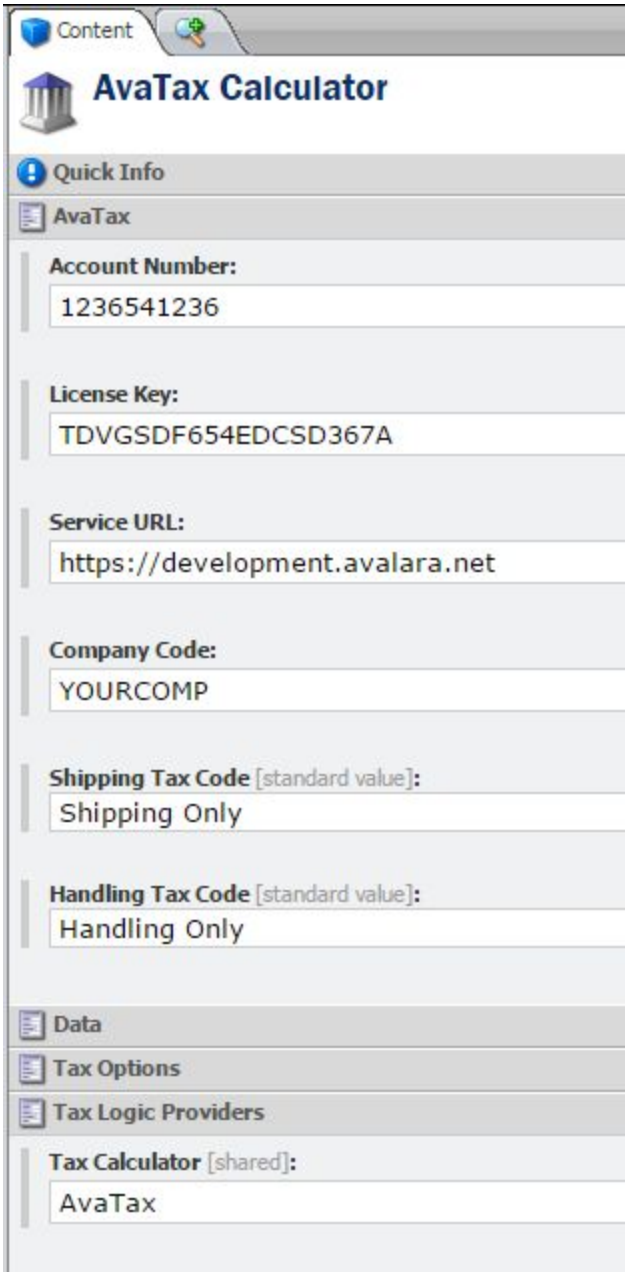
Active Commerce provides out-of-the-box integration with this third-party tax rate service. More information can be found on the Avalara website.

Account Setup

Sign up for an account on the Avalara site.

Installation

1. In Sitecore, use the Installation Wizard to install the AvaTax plugin.[1]
2. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Tax Calculation” and insert a new AvaTax Calculator item.
3. On the new AvaTax Calculator item, in the *AvaTax* section, set your AvaTax **Account Number**, **License Key**, and **Company Code**.
4. For development, your **Service URL** will be https://development.avalara.net (this is default). For production, be sure to switch this to https://avatax.avalara.net.
5. In the *Tax Logic Providers* section, ensure the **Tax Calculator** is set to “AvaTax” (this is default).
6. (Optional) In the *Tax Options* section, select **Locations** for which you’d like to limit AvaTax as the used tax calculator. If none are selected, AvaTax will be used for all locations (this is default).
7. If necessary, move the new item through workflow and publish.



Error Handling Configuration

On the AvaTax Calculator item, there is an additional *AvaTax Error Handling* section that allows configuration of AvaTax service errors. Anything above the configured **Maximum Allowed Severity Level** will result in an exception, and anything below will return the configured **Fallback Tax Rate** (e.g. 0% or a high tax rate like 12%). The default values are appropriate for most situations, but you may want to adjust depending on your business needs.

The available options have been provided per guidance from AvaTax: http://developer.avalara.com/api-documentation/avatax-15-api/designing-your-integration/errors-and-outages

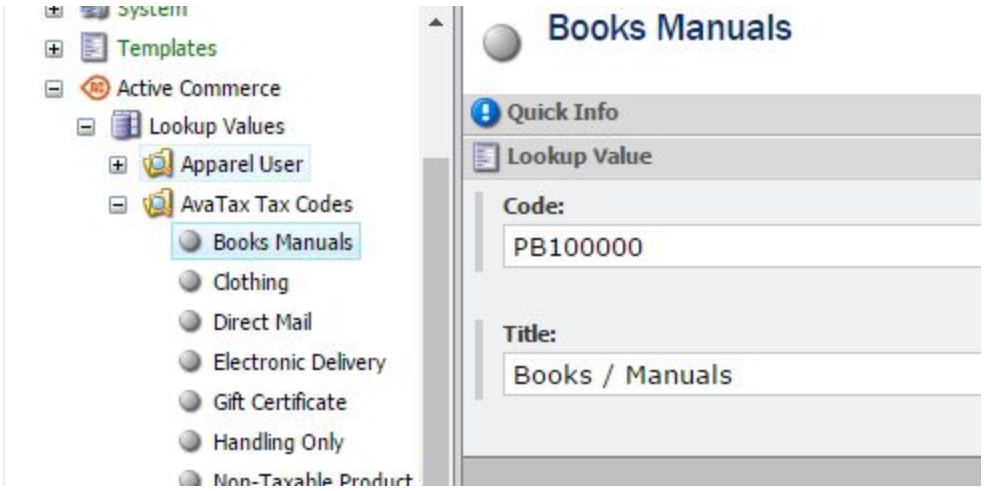
Source Address Configuration

The AvaTax Plugin requires the source address of the company to use for tax calculation. In Active Commerce, this is maintained under your site's Company Master Data record.

1. In Sitecore content tree, go to "sitecore/content/<My Site>/Webshop Business Settings/Company Master Data"
2. Fill in address information. For **Country**, make sure to use a two-letter country ISO code (e.g. "US") which corresponds to the country under "sitecore/content/<My Site>/Webshop Business Settings/Countries".
3. Save and publish all changes.

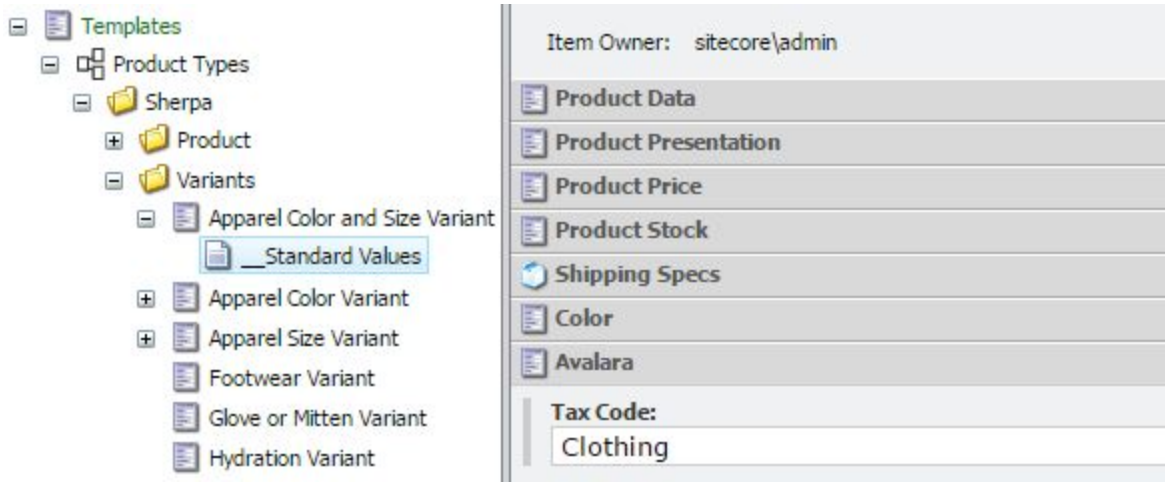
Product Configuration

The AvaTax Plugin ships with a basic list of AvaTax Tax Codes in "sitecore/Active Commerce/Lookup Values/AvaTax Tax Codes". Additional codes can be obtained from AvaTax if these don't fully meet your needs. Examples could include codes for products like Membership Dues, Concert Tickets, or Software.



Each product in Active Commerce should have a Tax Code selected in the Avalara section of the product. Consider setting a Tax Code on the Standard Values of your Product Type templates. Note, if you're product is a variant type, you will instead set the Tax Code on the Standard Values of your variant Product Type as seen below in the Active Commerce Sherpa demo store for a "Apparel Color and Size Variant" product type.

If a tax code is not provided, AvaTax will apply a default tax rate.



Troubleshooting

There is a config file for the Avalara AvaTax adapter than can be placed in your Sitecore */bin* folder to get additional debug logging. You can grab a copy of it from their GitHub site here: https://github.com/avadev/AvaTax-Calc-SOAP-csharp/blob/v14.4.1/Avalara.AvaTax.Adapter.dll.config.

The config isn't required but can do some nice logging of all AvaTax communication by adjusting the AvaLogger node. For example:

```
<AvaLogger logFilePath="../Data/logs" logLevel="DEBUG"
logMessages="true" logTransactions="true" logSoap="true"/>
```

This will result in 3 new log files in the Sitecore logs folder. Comments in the config file explain each setting. This should give you more detailed logging for the AvaTax adapter in general, and also get rid of the “Unable to load configuration file at ...” INFO-level Sitecore log entry.

NOTE: You may already see all levels of AvaTax event “logMessages” in the Sitecore log file, regardless of including this file or not. This is because the AvaTax adapter uses trace logging, and Sitecore has a trace listener included by default. See this post if you’d like to turn off these trace entries in the Sitecore log. Additionally, there is a known issue with the AvaTax adapter where the “logLevel” setting is not honored.

Rate Provider

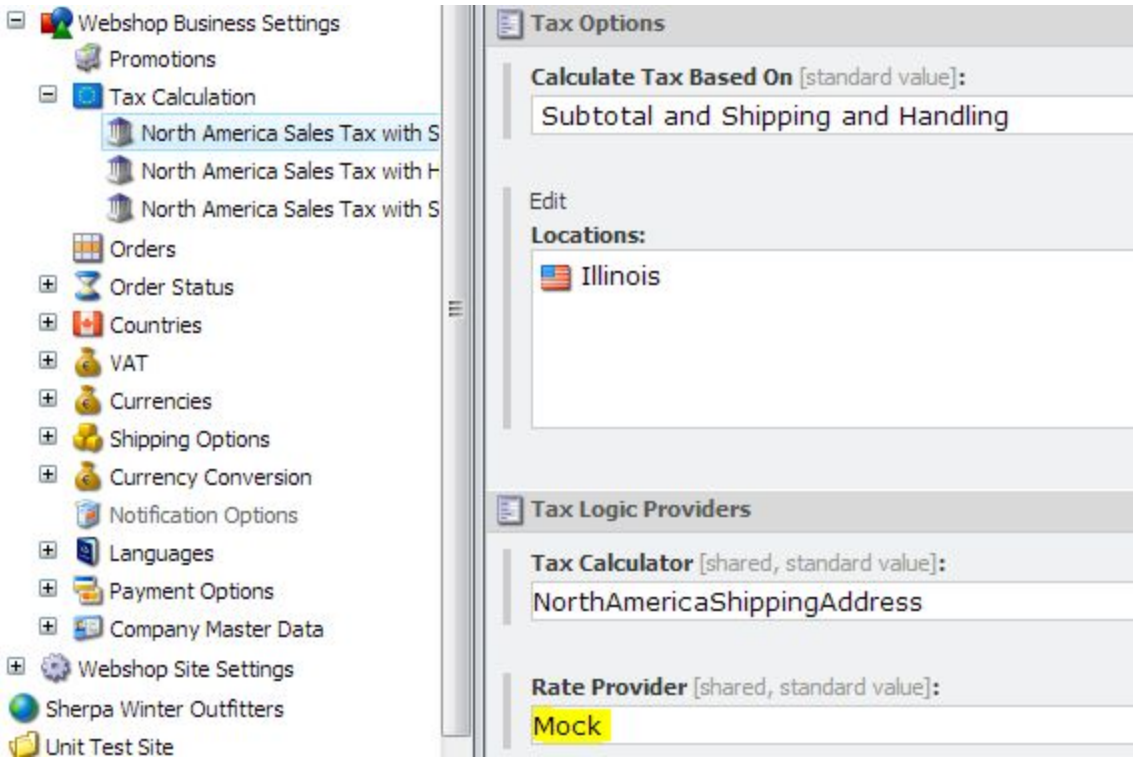
Rate providers provide sales tax rates to North America Tax Calculators. By default, a “Mock” provider is used.

Mock

This rate provider returns fixed (dummy) rates.

Configuring Tax Calculators to use Mock

1. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Tax Calculation”
2. For each item here, set the “Rate Provider” field to “Mock”



3. Save and publish all changes. Verify that the appropriate tax rates (per Location, Tax Based On, etc.) are now used on checkout.

FastTax

Active Commerce provides out-of-the-box integration with this third-party tax rate service. More information can be found on the FastTax website.

Account Setup

Sign up for an account on the FastTax site.

Installation

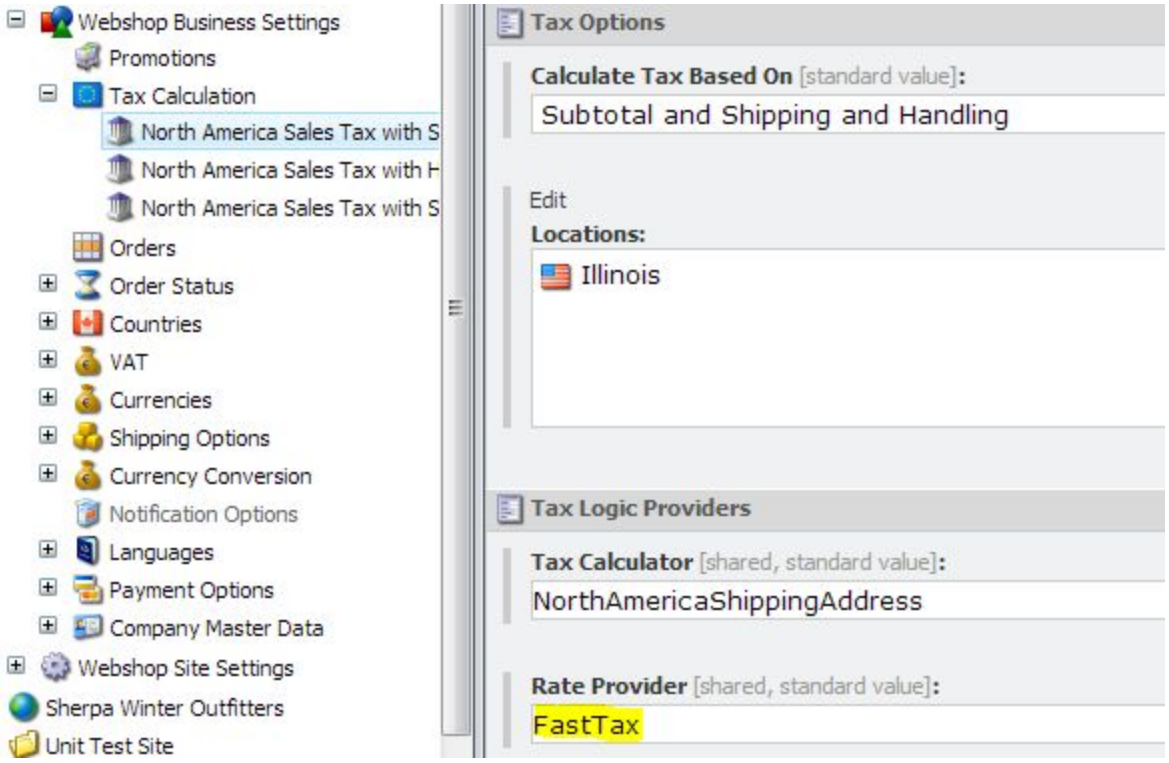
1. In Sitecore, use the Installation Wizard to install the Active Commerce FastTax plugin.[1]
2. Configure your FastTax license key in xActiveCommerce.xFastTax.config.
3. Add the FastTax service to your Web.config:

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="DOTSFastTaxSoap" closeTimeout="00:01:00"
```

```
openTimeout="00:01:00" receiveTimeout="00:10:00" sendTimeout="00:01:00"
allowCookies="false" bypassProxyOnLocal="false"
hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
useDefaultWebProxy="true">
        <readerQuotas maxDepth="32" maxStringContentLength="8192"
maxArrayLength="16384" maxBytesPerRead="4096"
maxNameTableCharCount="16384"/>
        <security mode="None">
          <transport clientCredentialType="None" proxyCredentialType="None" realm=""/>
          <message clientCredentialType="UserName" algorithmSuite="Default"/>
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
  <client>
    <endpoint address="http://trial.serviceobjects.com/ft/FastTax.asmx"
binding="basicHttpBinding" bindingConfiguration="DOTSFastTaxSoap"
contract="Service.DOTSFastTaxSoap" name="DOTSFastTaxSoap" />
  </client>
 </system.serviceModel>
```

NOTE: Use http://ws.serviceobjects.com/ft/FastTax.asmx when going live.

4. Update Tax Calculations to use FastTax
    a. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Tax Calculation.”
    b. For each item here, set the “Rate Provider” field to “FastTax.”



5. Save and publish all changes. Verify that the appropriate tax rates (per Location, Tax Based On, etc.) are now used on checkout.

Shipping

Active Commerce shipping providers are managed under “Webshop Business Settings/Shipping Options.” These are displayed on the payment page of the checkout process. By default, on insert of a new Active Commerce Website, a single fixed/default Flat Rate shipping option is included.

All providers define the following fields:

- **Code** – (Required) Must be unique. Used to identify the shipping provider.
- **Name** – (Required) Used as the “ShippingProvider” attribute on the order export.
- **Title** – (Required) Displayed during checkout. Also used as the “ShippingMethod” attribute on the order export.
- **Price** – (Optional) If not defined, 0 is used. If a rate service is configured, this value is added to the rate returned. This is then added to the Handling Fee to arrive at the final cost for the shipping option, which is displayed during checkout.
- **Handling Fee** – (Optional) If not defined, 0 is used. This is added to the Price to arrive at the final cost for the shipping option, which is displayed during checkout.
- **Display Rules** – (Optional) Define whether this shipping option should be displayed during checkout.

Providers which integrate with a rate service (all except Default/Fixed) also define the “Rate

Service Integration" fields:

- **Service Code** – Registered name of the IShippingService class implementation.
- **Service Configuration** – xml configuration specific to each service.

NOTE: Short Description, Icon, Available Notification Options, and Delivery Time fields are not used by default and can be left blank.

Default/Fixed

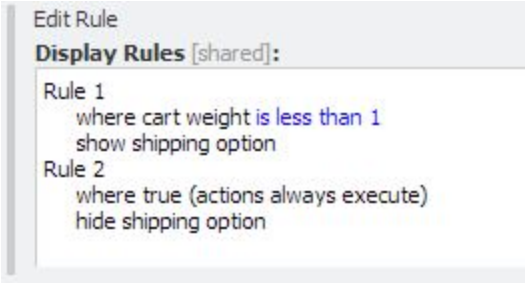
This is a simple shipping option that allows definition of fixed rates. There is no integration with a third-party service to retrieve rates. Common uses are shipping options like flat rate or low weight free.

Adding a flat rate provider

- In Sitecore content tree, go to "sitecore/Content/<My Site>/Webshop Business Settings/Shipping Options."
- Insert item > Shipping Provider.
- Enter a name for the provider (e.g., Flat Rate) and click OK.
- Enter a **Price** and, optionally, **Handling Fee.**
- Configure **Display Rules** if necessary.
- Save and publish changes. Verify that the shipping option now appears on checkout.

Adding a low weight free provider

1. In Sitecore content tree, go to "sitecore/Content/<My Site>/Webshop Business Settings/Shipping Options."
2. Insert item > Shipping Provider.
3. Enter a name for the provider (e.g., Low Weight Free) and click OK.
4. Enter a **Price** of '0'
5. Configure **Display Rules** to define your weight threshold (e.g., 1 lb in this example).



6. Save and publish changes. Verify that the shipping option now appears on checkout appropriately when total weight of products in cart is within and above threshold.

USPS

Account Setup

Sign up for an account on the USPS site.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce USPS plugin.[1]
2. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Shipping Options.”
3. Insert item > Shipping Provider.
4. Enter a name for the provider (e.g., USPS Priority) and click OK.
5. In **Service Code**, enter “USPS”
6. In **Service Configuration**, enter the following:

```
<usps>
  <userId>[Enter USPS UserId]</userId>
  <service>PRIORITY</service>
  <zipOrigination>53202</zipOrigination>
  <size>REGULAR</size>
  <container>Lg Flat Rate Box</container>
</usps>
```

7. Configure USPS settings appropriately (provided is for xml structure — values are example only).
   a. Set **userId** and **zipOrigination**.
   b. See USPS developer documentation for **service**, **size**, and **container** options.
   c. NOTE: there is no way to run in “test” mode with the RateV4 service (which this uses). Call USPS API support and let them know you’re using a tested module - and they’ll upgrade your account.
8. Enter additional **Price** and **Handling Fee** if appropriate.
9. Configure **Display Rules** if necessary.
10. Save, approve, and publish changes. Verify that the shipping option now appears on checkout.

UPS

Account Setup

Sign up for an account on the UPS site.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce UPS plugin.[1]
2. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Shipping Options.”
3. Insert item > Shipping Provider.
4. Enter a name for the provider (e.g., UPS Ground) and click OK.
5. In **Service Code**, enter “UPS”
6. In **Service Configuration**, enter the following:

```
<ups>
   <test>true</test>
   <security>
         <accessLicense>[Enter UPS License]</accessLicense>
         <username>[Enter UPS Username]</username>
         <password>[Enter UPS Password]</password>
   </security>
   <origin>
         <address>220 E Buffalo St</address>
         <city>Milwaukee</city>
         <state>WI</state>
         <zip>53202</zip>
         <country>US</country>
   </origin>
   <shipperNumber>123456</shipperNumber>
   <serviceCode>03</serviceCode>
   <packageType>02</packageType>
</ups>
```

7. Configure UPS settings appropriately (provided is for xml structure — values are example only).
    a. If **test** is set to true, the UPS test endpoint is used (https://wwwcie.ups.com/webservices/Rate). If false (or ommitted), the UPS production endpoint is used (https://onlinetools.ups.com/webservices/Rate).
    b. Set **security** and **origin**.
    c. The **shipperNumber** is optional but must be supplied in order for UPS to return negotiated rates. Ensure the shipper number is also associated with your username in your UPS account.
    d. See UPS developer documentation for **serviceCode** and **packageType** options.
8. Enter additional **Price** and **Handling Fee** if appropriate.
9. Configure **Display Rules** if necessary.
10. Save, approve, and publish changes. Verify that the shipping option now appears on checkout.

FedEx

Account Setup

Sign up for an account on the FedEx site.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce FedEx plugin.[1]
2. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Shipping Options.”

3. Insert item > Shipping Provider.
4. Enter a name for the provider (e.g., FedEx 2 Day) and click OK.
5. In **Service Code**, enter “FedEx”
6. In **Service Configuration**, enter the following:

```
<fedEx>
        <test>true</test>
        <userCredential>
                <key>[Enter FedEx Key]</key>
                <password>[Enter FedEx Password]</password>
        </userCredential>
        <clientDetail>
                <accountNumber>[Enter FedEx Account Number]</accountNumber>
                <meterNumber>[Enter FedEx Meter Number]</meterNumber>
        </clientDetail>
        <shipper>
                <!-- Fill in address below -->
                <street1>220 E Buffalo St</street1>
                <street2></street2>
                <city>Milwaukee</city>
                <state>WI</state>
                <postalCode>53202</postalCode>
                <countryCode>US</countryCode>
        </shipper>
        <serviceType>FEDEX_2_DAY</serviceType>
        <dropoffType>BUSINESS_SERVICE_CENTER</dropoffType>
        <packagingType>YOUR_PACKAGING</packagingType>
        <rateRequestType>ACCOUNT</rateRequestType>
</fedEx>
```

7. Configure FedEx settings appropriately (provided is for xml structure — values are example only).
    a. If **test** is set to true, the FedEx test endpoint is used (https://wsbeta.fedex.com:443/web-services/rate). If false (or ommitted), the FedEx production endpoint is used (https://ws.fedex.com:443/web-services).
    b. Set **userCredential**, **clientDetail**, and **shipper** values appropriately.
    c. See FedEx developer documentation for **serviceType, dropoffType,** and **packagingType** options.
    d. Set **rateRequestType** to LIST for list pricing, ACCOUNT for negotiated pricing.
8. Enter additional **Price** and **Handling Fee** if appropriate.
9. Configure **Display Rules** if necessary.
10. Save, approve, and publish changes. Verify that the shipping option now appears on checkout.

Payment

Active Commerce payment methods are managed under “Webshop Business Settings/Payment Options.” These are displayed on the payment page of the checkout process. By default, on insert of a new Active Commerce Website, a single mock payment option is included.

Mock Integrated Payment

This is a mock payment option you can use for testing/demo purposes.
NOTE: On a new Active Commerce installation, the following should already be in place, and the mock payment option ready to use.

Installation

1. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Payment Options.”
2. You should already see a “Mock Integrated Payment” option here. If not:
    a. Insert item > Payment.
    b. Enter “Mock Integrated Payment” for the name, and click OK.
    c. Change the **Code** field to “MockIntegratedPayment”
3. Mock payments must also be allowed by setting a configuration value.
    a. In “App_Config\Include\xActiveCommerce.Env.config”, ensure the “ActiveCommerce.AllowMockPayment” setting value is set to “true.”
4. Navigate to the “Credit Card Payment” component of your checkout page, and select Mock Integrated Payment as the associated Payment Option.
    a. /sitecore/content/<site>/Home/shop/checkout/Components/Credit Card Payment
5. Save, approve, and publish all changes. Verify that credit card transactions (see below for valid values) now function as a payment option on checkout.
    NOTE: You may need to clear session cookies as the old payment option could be stored in the session.

Valid credit card values

Use the following credit card information when checking out. These are values that the mock payment option will consider valid:

- **Card number**: 4111 1111 1111 1111
- **Expiration date**: any date in the future
- **Security code**: any number over 100

Authorize.NET

Account Setup

Sign up for an account on the Authorize.NET site.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce Authorize.NET plugin.[1]
2. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Payment Options.”
3. Insert item > Payment.
4. Enter “AuthorizeNET” as the name of the new payment and click OK.
5. On the new AuthorizeNET item:
    a. Ensure **Code** is set to “AuthorizeNET” (should already be set if used for item name).
    b. Set **Username** to your Authorize.NET API Login ID.
    c. Set **Password** to your Authorize.NET Transaction Key.
    d. In **Settings**, enter the following:
    ```
    <settings>
      <setting name="transactionDescription">[MySite] Order</setting>
      <setting name="authorizeOnly">false</setting>
      <setting name="testMode">true</setting>
    </settings>
    ```
6. Configure settings
    a. **transactionDescription** – Sent to Authorize.NET as transaction description. Replace [MySite] with your site name or set however you see fit.
    b. **authorizeOnly** – Set to true if you want to only perform an authorization on payments. Set to false if you’d like to perform an authorization + capture.
    c. **testMode** – Determines whether to use Authorize.NET in test mode. Set to false when going live.
7. Navigate to the “Credit Card Payment” component of your checkout page, and select AuthorizeNET as the associated Payment Option.
    a. /sitecore/content/<site>/Home/shop/checkout/Components/Credit Card Payment
8. Save, approve, and publish all changes. Verify credit card transactions now function as a payment option on checkout.
    NOTE: You may need to clear session cookies as the old payment option could be stored in the session.

CyberSource

Account Setup

Sign up for an account on the CyberSource site.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce CyberSource

plugin.[1]

2. As noted in the installation, add the CyberSource service to your Web.config:

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="ITransactionProcessor" closeTimeout="00:01:00"
openTimeout="00:01:00" receiveTimeout="00:30:00" sendTimeout="00:10:00"
allowCookies="false" bypassProxyOnLocal="false"
hostNameComparisonMode="StrongWildcard" maxBufferSize="65536"
maxBufferPoolSize="524288" maxReceivedMessageSize="65536"
messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
useDefaultWebProxy="true">
        <readerQuotas maxDepth="2147483647" maxStringContentLength="2147483647"
maxArrayLength="2147483647" maxBytesPerRead="2147483647"
maxNameTableCharCount="2147483647" />
        <security mode="TransportWithMessageCredential" />
      </binding>
    </basicHttpBinding>
  </bindings>
  <client>
    <endpoint address="https://ics2wstest.ic3.com/commerce/1.x/transactionProcessor"
binding="basicHttpBinding" bindingConfiguration="ITransactionProcessor"
contract="Service.ITransactionProcessor" name="portXML" />
</client>
</system.serviceModel>
```

NOTE: Use https://ics2ws.ic3.com/commerce/1.x/transactionProcessor when going live.

3. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Payment Options.”
4. Insert item > Payment.
5. Enter “CyberSource” as the name of the new payment and click OK.
6. On the new CyberSource item:
    a. Ensure **Code** is set to “CyberSource” (should already be set if used for item name).
    b. Set **Username** to your CyberSource Merchant ID.
    c. Set **Password** to your CyberSource Transaction Security Key.
    d. In **Settings**, enter the following:

```
<settings>
  <setting name="authorizeOnly">false</setting>
</settings>
```

7. Configure settings
    a. **authorizeOnly** – Set to true if you want to only perform an authorization on payments. Set to false if you’d like to perform an authorization + capture (also

referred to as a “sale” by CyberSource).

8. Navigate to the “Credit Card Payment” component of your checkout page, and select CyberSource as the associated Payment Option.
    a. /sitecore/content/<site>/Home/shop/checkout/Components/Credit Card Payment
9. Save, approve, and publish all changes. Verify credit card transactions now function as a payment option on checkout.
    a. *NOTE: You may need to clear session cookies as the old payment option could be stored in the session.*

Moneris

Account Setup

Sign up for an eSELECTplus account on the Moneris site.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce Moneris plugin.[1]
2. In Sitecore content tree, go to “sitecore/Content/<My Site>/Webshop Business Settings/Payment Options.”
3. Insert item > Payment.
4. Enter “Moneris” as the name of the new payment and click OK.
5. On the new Moneris item:
    a. Ensure **Code** is set to “Moneris” (should already be set if used for item name).
    b. In **Settings**, enter the following:

```
<settings>
  <setting name="Host">esplusqa.moneris.com</setting>
  <setting name="StoreId">monusqa003</setting>
  <setting name="APIToken">qatoken</setting>
  <setting name="AuthorizeOnly">false</setting>
  <setting name="TestMode">true</setting>

  <!-- eFraud Settings -->
  <setting name="AvsProtectionEnabled">false</setting>
  <setting name="AvsSuccessCodes">C,D,E,F,L,M,P,R,S,U,W,X,Y,Z</setting>
  <setting name="CvdProtectionEnabled">false</setting>
  <setting name="CvdSuccessCodes">M,Y,U</setting>
</settings>
```

6. Configure settings
    a. The above settings represent a test-mode setup using a typical Moneris test account and API token. Replace the **Host**, **StoreId**, and **APIToken** values with your real information from Moneris.
    b. **AuthorizeOnly** – Set to true if you want to only perform an authorization on payments. Set to false if you’d like to perform an authorization + capture.

c. **TestMode** – Set to true if using the Moneris QA environment. Set to false when going live.
d. **AvsProtectionEnabled** – Set to true to perform an AVS check on the billing address zip code. If the AVS response code from Moneris does not match one of the success codes (below), the payment will be considered a failure.
e. **AvsSuccessCodes** – Set to your accepted AVS codes. Please refer to the *CVD AVS Result Codes* document available at https://developer.moneris.com for more information. We provide a default set, but it's best to review and make sure these align with your business requirements.
f. **CvdProtectionEnabled** – Set to true to perform a CVD check on the billing address zip code. If the CVD response code from Moneris does not match one of the success codes (below), the payment will be considered a failure.
g. **CvdSuccessCodes** – Set to your accepted CVD codes. Please refer to the *CVD AVS Result Codes* document available at https://developer.moneris.com for more information. We provide a default set, but it's best to review and make sure these align with your business requirements.

NOTE: If testing eFraud (AVS and CVD) in the Moneris QA environment, you must use the Visa test card numbers 4242424242424242 or 4005554444444403 AND the amounts described in the Simulator eFraud Responses document available at https://developer.moneris.com. For this reason, unless specifically testing eFraud, it's recommended to disable AVS/CVD protection when in TestMode, otherwise you may see unpredictable results during checkout payment.

7. Navigate to the "Credit Card Payment" component of your checkout page, and select Moneris as the associated Payment Option.
    a. /sitecore/content/<site>/Home/shop/checkout/Components/Credit Card Payment
8. Save, approve, and publish all changes. Verify credit card transactions now function as a payment option on checkout.
    a. *NOTE: You may need to clear session cookies as the old payment option could be stored in the session.*

PayPal

Note: At this time, only "Sale" transactions are supported; it is not possible to configure "Authorization" or "Order" payment actions.

Account Setup

A PayPal Business Account is required. If you don't have one, you can sign up for a PayPal Sandbox account here: https://developer.paypal.com. This will provide you with both a test Business Account and a test Personal Account to test your transactions. When you have finished testing, then switch to a live account.

Follow the directions for creating an API signature:
https://developer.paypal.com/docs/classic/api/apiCredentials/#creating-an-api-signature

Consider turning off the “PayPal Account Optional” setting:
https://www.paypal.com/webapps/mpp/account-optional
We really don’t need to offer credit card options; those are already covered in Active Commerce.

If you’d like to store the customer’s PayPal billing address with the order, you will need to get billing address enabled for your account by contacting PayPal customer/Business Support.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce PayPal plugin.[1]
    a. Click “Yes” if prompted to override Newtonsoft.Json.dll
    b. **You will likely receive a “Could not load file or assembly 'Newtonsoft.Json’” error during install (that’s OK, the install was successful)**. This will be fixed in the next step.
2. In the Web.config, add or update the following dependentAssembly directives for Newtonsoft.Json to redirect to version 7.0 and PayPalCoreSDK to version 1.7.1. **There will likely be an existing Newtonsoft.Json directive, so be sure to remove or update it!**

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="Newtonsoft.Json" publicKeyToken="30ad4fe6b2a6aeed"
culture="neutral" />
      <bindingRedirect oldVersion="0.0.0.0-6.0.0.0" newVersion="7.0.0.0" />
    </dependentAssembly>
    <dependentAssembly>
      <assemblyIdentity name="PayPalCoreSDK" publicKeyToken="5b4afc1ccaef40fb"/>
      <bindingRedirect oldVersion="1.0.0.0-1.7.0.0" newVersion="1.7.1.0"/>
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

3. In Sitecore content tree, create a new Payment item called “PayPal” in “sitecore/Content/<My Site>/Webshop Business Settings/Payment Options.”
4. On the new PayPal item:
    a. Set **Short Description** to the markup below. This is the recommended markup provided by PayPal.

```
<img src="https://www.paypal.com/en_US/i/logo/PayPal_mark_37x23.gif"
align="left" style="margin-right:7px;"><span style="font-size:11px;
```

font-family:Arial,Verdana;">The safer, easier way to pay.

Note: Sitecore may display a validation error on the field based on this input. You can safely ignore it.

b. Set the **Redirect message** to:

<h2>Redirecting to PayPal. Please wait...</h2>

c. Set **Username** to your PayPal API username.
d. Set **Password** to your PayPal API password.
e. For test transactions, set the **Payment Provider Url** to:

https://sandbox.paypal.com/cgi-bin/webscr

For live transactions, change the **Payment Provider Url** to:

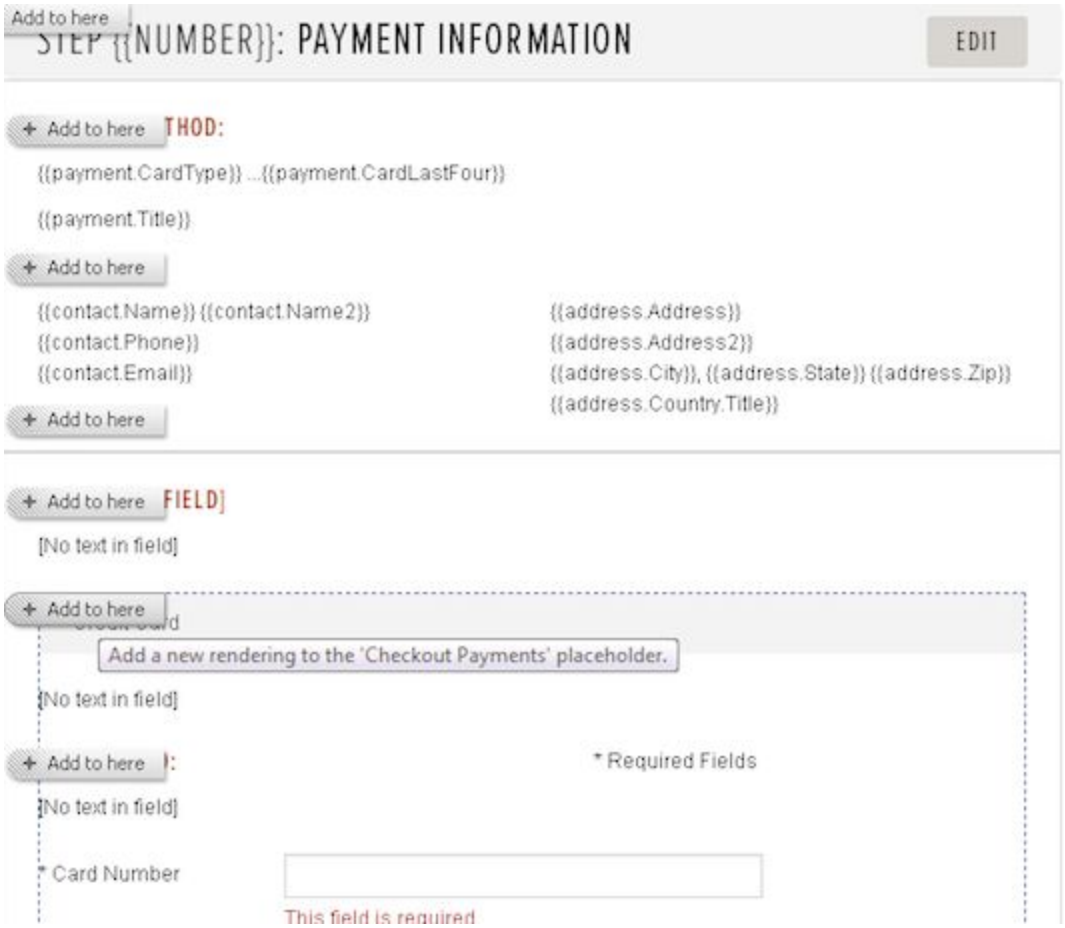
https://www.paypal.com/cgi-bin/webscr

f. In **Settings**, copy the following XML into the field and set the **APISignature** and **Environment** (*sandbox* or *live*) accordingly.

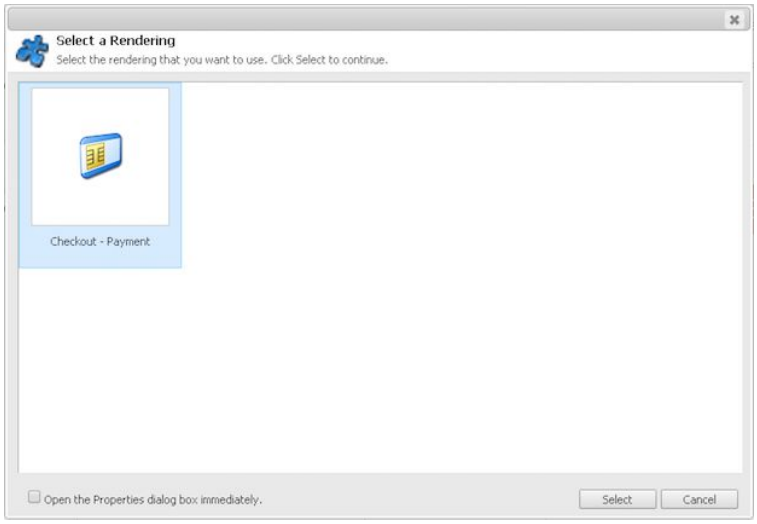
<setting id="APISignature">[APISignature]</setting>
<setting id="Environment">sandbox</setting>

When you have finished testing and are ready to go live with PayPal, change the **Environment** setting value from “sandbox” to “live”.

5. On the <site>/Home/Active Commerce/Checkout item:
   a. Open the page in the Page Editor.
   b. Find the billing/payment step, and within the *Checkout Payments* component, click “Add to here” to add a new payment option.

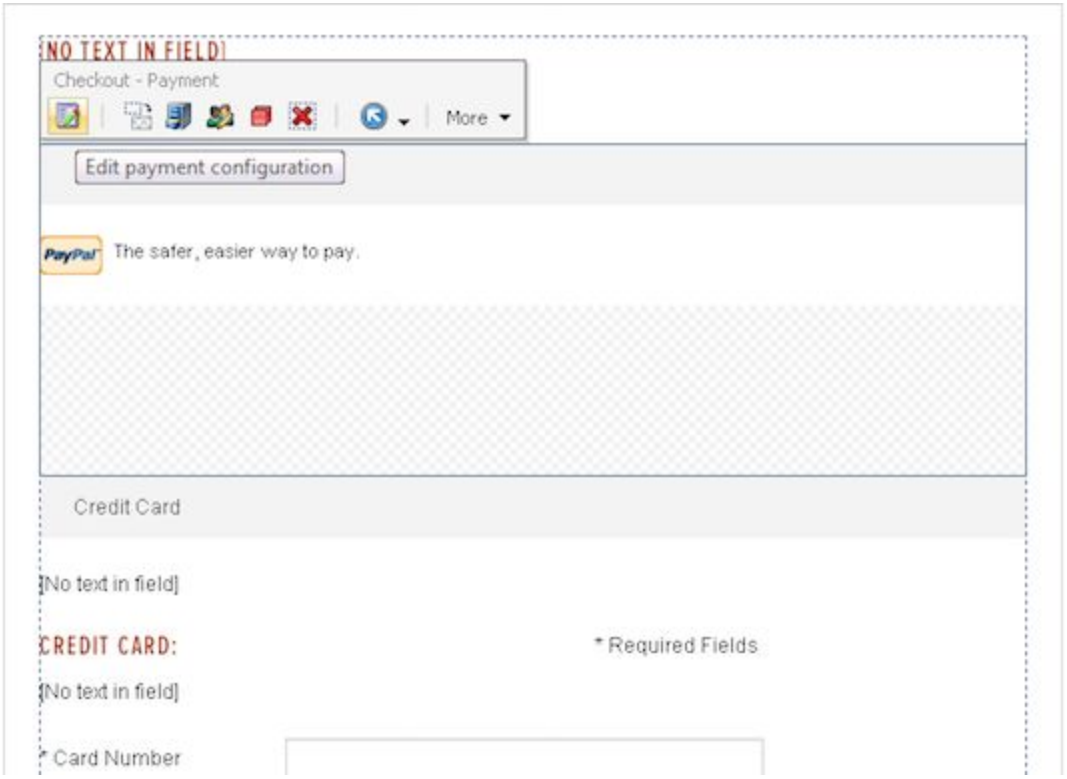


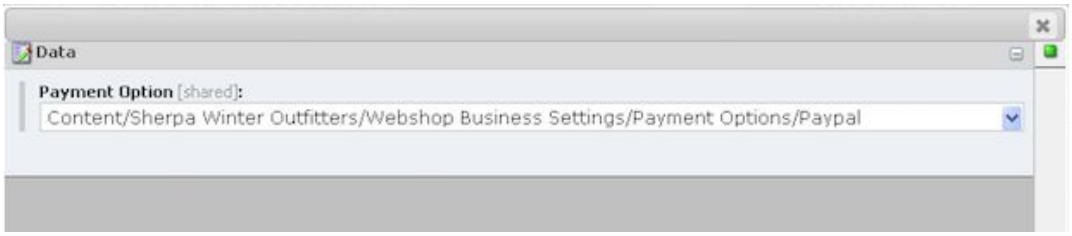
c. Select the *Checkout - Payment* rendering, and follow the instructions to create a new datasource item for the component.



d. After adding the new payment option, hover over the component and click the *Edit Payment Configuration* button. Within the resulting form, select the PayPal

payment option.





e. Click *Save* in the Page Editor ribbon.

6. Ensure all new and updated items have been approved in workflow, and publish changes. Verify that PayPal is now shown as a payment option on checkout.

Reviews

Active Commerce provides empty placeholders on the product detail page to allow simple installation of review plugins. Additional configuration is necessary if you'd like to show inline ratings on other pages (e.g. product list).

RateVoice

Also known as SocialRMS, Active Commerce provides out-of-the-box integration with this third-party reviews service. More information can be found on the RateVoice / SocialRMS website.

Account Setup

Sign up for an account on the RateVoice / SocialRMS website.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce RateVoice plugin.[1]
2. Configure your RateVoice SiteID in xActiveCommerce.xRateVoice.config.
3. Ensure reviews are now displayed on the product detail page. There should be a rating summary above the price, and a “Reviews” tab.

Additional Configuration

Please note that the RateVoice markup injected in the tab is not mobile friendly (at the time of writing). You will need to add additional styles to your skin if this is a requirement. See the Sherpa demo skin for an example (*\skins\sherpa\styles\product.mobile.less*).

Persistent Carts

Persist user shopping carts beyond the ASP.NET Session using the Active Commerce Database.

This plugin handles the following use cases:

- A customer whose session expires
- A customer who has created an account, then logs in elsewhere
- Restored products which are no longer available
- Restored products which are no longer in stock, or which have less stock available than was previously in the user's cart
- Restored coupon code which is no longer available
- Alerting the customer of a restored, or partially restored cart, including details about which products or coupon code failed
- Restoring cart when necessary, including:
    - On first request
    - When customer logs in
- Persisting cart when necessary, including:
    - When cart changes
    - When customer logs in
    - When customer creates account
- Deleting persisted cart when necessary, including:
    - When cart is emptied
    - After order is placed
- Emptying session cart when customer logs out

- Cleanup of stale persistent carts via Sitecore agent

The persistence and restoration processes have been abstracted into Sitecore pipelines to make them more extensible, as well as the restore and persist touchpoints.

Installation

1. In Sitecore, use the Installation Wizard to install the Active Commerce Persistent Carts plugin.[1]
2. Publish the new translation items added under *sitecore\commerce\translation*.
3. Go to the following address: <site host>/sitecore/admin/databaseutility.aspx
4. Click the "Update Schema" button.
5. You should see a "Success!" message when completed.
6. In SQL Server Management Studio, run the following script against your Active Commerce Database:

```
INSERT INTO [dbo].[hibernate_unique_key] ([next_hi],[entity])
VALUES (1,'ActiveCommerce.PersistentCarts.PersistentCart')
INSERT INTO [dbo].[hibernate_unique_key] ([next_hi],[entity])
VALUES (1,'ActiveCommerce.PersistentCarts.PersistentCartLine')
```

Additional Configuration

Cleanup Agent

Included with the configuration file (xActiveCommerce.xPersistentCarts.config) is an agent which cleans up stale persistent carts. By default, it runs every 4 hours and keeps carts used within the last 30 days. Depending on your traffic, you may want to adjust the interval or number of days to keep, either directly in the file, or preferably, with a Sitecore config patch.

Translation Dictionary Items

There are various translation dictionary items available for configuring cart restoration messaging. These can be found under *sitecore\commerce\translation\C*, items starting with *Cart-Restore-*.

Coveo for Sitecore Integration

Active Commerce provides a plugin to allow you to easily utilize the Coveo Enterprise Search platform as your Sitecore search provider, and as a site search option for your Active Commerce site.

Features

- Coveo index configuration for the Active Commerce product indexes
- Automatically configures Active Commerce product filters as Coveo index facets

- Automatically configures Active Commerce product index sort options as Coveo sortable fields
- Product URL resolving in Coveo Search interfaces

Compatibility

The Active Commerce plugin for Coveo has been tested with the July 2016 release of Coveo for Sitecore 4.0.

Known Issues / Limitations

- Coveo's search provider requires use of server-side pagination with LINQ (i.e., *Skip()* and *Take()*) and will default to a page size of 10. While Active Commerce search APIs support server-side pagination, the default admin and front-end UIs do not make use of this. Specifically, the Product Page tab (in the Content Editor) and the Category / product listing pages. This plugin will default the page size to the Coveo maximum, 1000, which means that you are constrained to a maximum of 1000 products in a category with the out-of-the-box Active Commerce product list page. This can be configured via the *DefaultNumberOfResults* setting in *Coveo.SearchProvider.xActiveCommerce.config*.
- The current version of Coveo Cloud does not allow multiple fields with the same name. This will cause issues if the field types are different. For example, Sitecore defines a “Size” *Number* field which Coveo will map as *Long*. If you add a “Size” *Single-Line Text* field to your product template, the values stored will not be as expected in the index (which will prevent the field’s use as a filter or custom sort). Coveo is looking to handle this in a future version, but for now, you’ll have to rename your field.

Installation

1. Follow instructions from the Coveo Developer’s Community for installation of Coveo for Sitecore 4.0.
2. In Sitecore, use the Installation Wizard to install the Active Commerce Coveo plugin.[1]
3. Utilize the Indexing Manager to rebuild the *ac_products* indexes.

Displaying Product Data in Coveo Search Interfaces

Coveo provides a flexible framework for building search interfaces for your Sitecore site, which may be used on your Active Commerce site for site searches and/or replacement of the Active Commerce category / product list pages. To add Active Commerce product data to the Coveo search interface, you need to create additional computed fields which read data from the Active Commerce domain model.

An example of creating an Active Commerce product search with the Coveo search interface is available on our GitHub.

Enabling Customer Wish Lists

Active Commerce features simple wish list functionality which allows customers to:

- Create new wish lists and add products to wish lists from the product detail page
- View, edit, and delete wish lists
- Update/remove items from a wish list
- Add items from their wish list to their shopping cart
- Print a wish list

Wish lists are not enabled by default when you create a new Active Commerce website. To enable wish lists, check the “Enable Wish Lists” option on your site General Settings (*/sitecore/content/<site>/Site Settings/General*) and publish.

The Active Commerce Database

As part of the installation, you should have created an additional database. This database uses the popular NHibernate ORM framework (along with the Fluent counterpart) to store all non-Sitecore entities, such as orders, product stock, and wish lists. One of the key advantages of this approach is the ability to support all popular relational databases. By default, Active Commerce is setup for SQL Server 2012, but this is easy to change.

Changing configured database

1. Open the xActiveCommerce.Data.config file found under App_Config/Include.
2. Look for the <acConfigurationBuilder> pipeline. You’ll see the first processor has a list of configured property values. You’re interested in the “dialect”:

   <property name="dialect">NHibernate.Dialect.MsSql2012Dialect, NHibernate</property>

3. Replace this property value with the SQL dialect appropriate for your database, either directly in the file, or preferably, with a Sitecore config patch. For a list of NHibernate SQL dialects, please see the documentation or the relevant namespace on Github.
4. There are some additional indexes which Active Commerce adds to the database, which are found in *App_Config\Include\NHibernate_Auxillary.sql*. You may wish to update these for your database.

Note: Active Commerce has only been tested on MS SQL Server. If you utilize Active Commerce on a different database, test thoroughly and be aware that our standard support team will be unable to provide assistance for database-specific issues.

Using the SES Order Manager

Active Commerce provides limited support for utilizing the Order Manager application provided by Sitecore with Sitecore E-commerce Services. Out of the box, the Order Manager can be used to:

- View recent orders and search for orders
- Review order details
- Debug failed orders
- Change order status
- Input order notes
- Input shipping tracking URL

Configuring the Shop Context

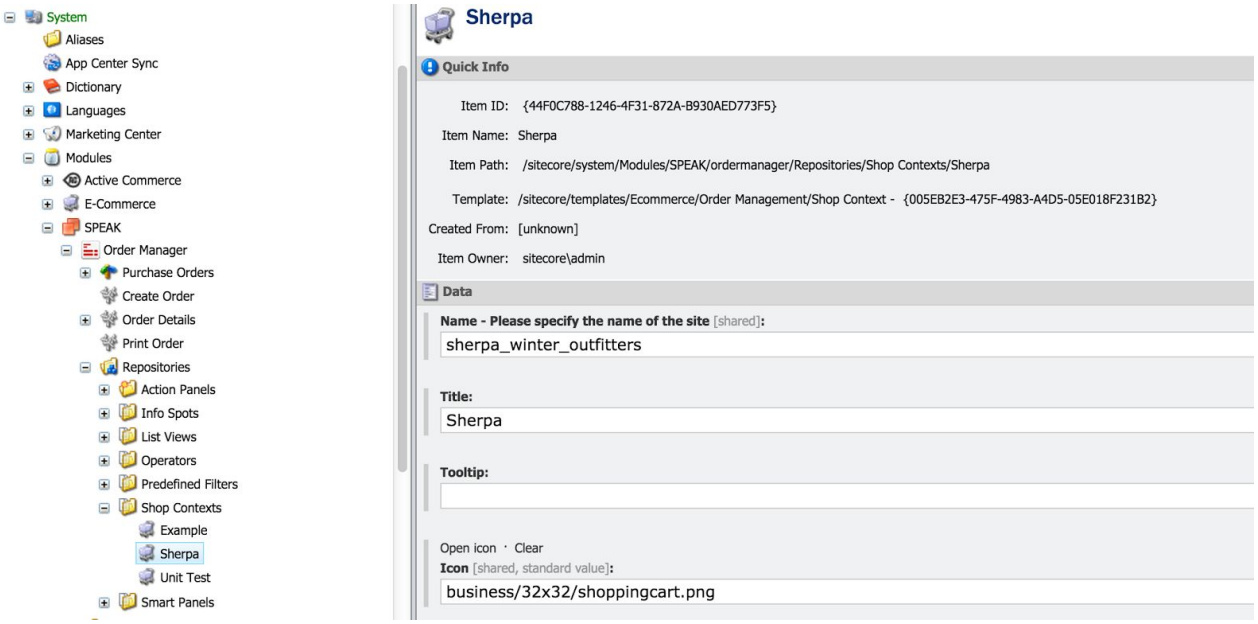
To make an Active Commerce site available in the SES Order Manager, you first need to configure the shop context. In the Content Editor, navigate to the following item:

/sitecore/system/Modules/SPEAK/ordermanager/Repositories/Shop Contexts

Under this item, use the Content Editor to Insert > Shop Context and fill in the required fields:

Name - the name of the site as it is configured in your <site /> definition
Title - the user-friendly name for the site

Then save the item. There is no need to publish.



Accessing the Order Manager

On the Content Management server of your Sitecore installation, you can access the Order Manager at the following URL:

http://<host>/speak/ordermanager

The username and password should be of an administrative user or a member of the *Order Manager Administrators* role.

Sitecore HTML Output Caching

The Sublayout renderings provided by Active Commerce come preconfigured for Sitecore HTML output caching. You should review the settings on these renderings (*/sitecore/layout/Sublayouts/ActiveCommerce*) and tweak these settings based on your requirements, particularly if you have skinned a rendering and created additional cache dependencies, or if you plan on utilizing personalization on a rendering.

The sections below outline some particular issues around caching which you should be aware of. If you are not already familiar with Sitecore output caching, you should review Chapter 4 of the Sitecore Presentation Component Reference and John West's introduction to how Sitecore caches output.

Clearing Output Caches after Publish

To ensure that the output cache is cleared following a Sitecore publish, the Sitecore *HtmlCacheClearer* event handler needs to be configured with the names of the sites that need to be cleared. See the *xActiveCommerce.Sites.config* which ships with the Sherpa demo site for an example of this. Be sure to configure cache clearing for both the *publish:end* and *publish:end:remote* events for each Active Commerce site. Alternatively, you can implement custom code to clear output cache for all managed sites after publication.

Output Caching and Development Environments

Output caching can be a hindrance during local development, especially when you are skinning sublayout renderings. In development environments, you may want to set the *cacheHtml* attribute of your site definition to *false* using a development-only configuration patch. This will disable output caching for that site. You should ensure that output caching is re-enabled in your testing environments so that it does not have any unintended side effects on your solution.

Output Caching and the Skinning System

When cached, a rendering no longer executes. This is why the Active Commerce skinning system

relies on data from the rendering itself to determine what style/script files and bundles to include on a page (i.e. the *Skin References* fields on the rendering). If you need to add rendering-specific scripts to a rendering which require server-side data, you should do this via inline *<script>* blocks rather than the Cassette *Bundles.AddInlineScript* method. The latter will never be invoked once the rendering has been cached.

To pass server-side data to a client script, you can utilize the *extend* client-side function to populate an object property which will hold that configuration for use in the Javascript module. This function will take care of merging namespaces and objects. You can see examples of this in the base skin, such as in *Category-ProductList.ascx* and *browse-product.js*.

```
<script type="text/javascript">
    extend('ActiveCommerce.BrowseProduct.serverConfig', {
        productList: <%= ViewModel.ToJSON() %>,
        productListURL: '<%= ProductListUrl %>',
        compareUrl: '<%= ComparisonUrl %>',
        hasCompareFields: '<%= (Model.CompareFields != null && Model.CompareFields.Any()) %>'
    });
</script>
```

Populating serverConfig in the rendering

```
extend('ActiveCommerce.BrowseProduct', {
    config: {
        productListContainerId: 'products-list',
        productTemplateId: 'product',
        showMoreContainerId: 'products-show-more',
        showAllBtnClass: 'show-all',
        showMoreBtnClass: 'show-more',
        productsPerPage: 8,
        sortById: 'sort-by',
        loaderId: 'products-loader',
        fadeSpeed: 150,
        productList: {}, // set on serverConfig in Category/MobileCategory-ProductList.ascx
        productListURL: '', // set on serverConfig
        compareUrl: '', // set on serverConfig
        hasCompareFields: false // set on serverConfig
    },
```

Declaring config, including properties populated from serverConfig

```
var o = ActiveCommerce.BrowseProduct,
    c = $.extend(true, o.config, o.serverConfig);
```

Extend config with values from serverConfig

```
if ($.isEmptyObject(e.parameters) && c.productList && !$.isEmptyObject(c.productList)) {
    // if no parameters, we can use the unfiltered list already bootstrapped on the page
    o.loadDataSet(c.productList);
} else {
    o.loadDataSet();
}
```

Utilizing values

Changing Cache Settings and Active Commerce Upgrades

If you make adjustments to cache settings on Active Commerce renderings, you should ensure that you can restore these changes if an item is overwritten by an Active Commerce upgrade. Options include:

- Storing any changed items in your Visual Studio project using Team Development for Sitecore (TDS).
- Using Sitecore serialization to keep a copy of these items or store them in source control.
- Create a Sitecore package with these changed items.

Output Caching and Product List Pages

Though caching is enabled on Active Commerce product listing renderings by default, there are many conditions under which you may want to disable caching on the *Category - Product List* and *Mobile Category - Product List* renderings. Some examples would be:

- The use of personalization rules on inline promotional elements in the *ac-promos* placeholder.
- Dynamic pricing based on user role, or dynamic pricing through integration with a back-end system.
- Search result boosting based on user personas (such as that provided by the Coveo for Sitecore search solution).
- Enabling of “out of stock” messaging in the product list.

The scenarios above or other requirements may necessitate disabling caching on these renderings, or implementing a custom cache key by extending the *SkinnedSublayout* class. If you disable caching, ensure that you have sufficient server resources to handle the expected load on these pages by performing load tests which simulate peak traffic levels.

It should also be noted that product listing pages in Active Commerce are highly dependent on search indexes, so the *Category - Product List* and *Mobile Category - Product List* renderings are configured to clear cache on index updates.

Output Caching and Product Detail Pages

The Active Commerce product detail page *Product* rendering is also cached by default. This page is a bit more dynamic thus utilizes additional *VaryBy* parameters:

- *VaryByDevice* - Conditional rendering is utilized to output mobile vs desktop social sharing

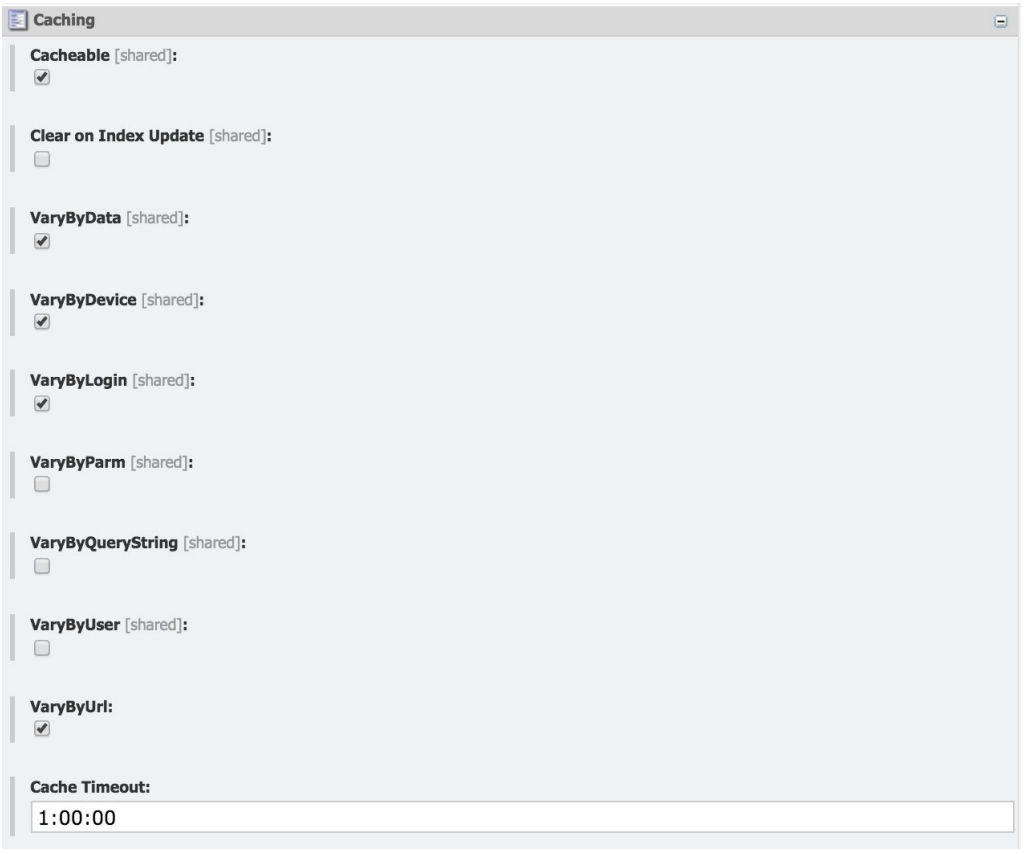
in the *ac-sharing* placeholder of this rendering.

- *VaryByLogin* - The “Add to Wishlist” button on the product detail page outputs and functions differently based on whether the user is authenticated.
- *VaryByUrl* - This is a custom cache key added by Active Commerce so that product detail page breadcrumbs are cached appropriately for each product detail page URL path (e.g. */products/* and */products/category/subcategory/*).

The product detail page may also output differently based on the stock level of a product. Since stock data is not published, the standard output cache clearing will not be triggered on a change to stock level. For this reason, Active Commerce implements time-based output caching for product detail pages. *SkinnedSublayout* renderings allow the configuration of a cache timeout. The default for product detail pages is one hour.

Active Commerce also performs a stock check when products are added to the shopping cart, providing a fallback for cases where a cached product detail page still indicates that a product is in stock. Depending on your specific requirements and factors such as stock volatility and order volume, you may want to make changes to this default implementation such as:

- Increasing or decreasing the cache timeout of the product detail page. This is done in the *Caching* settings of the *Product* rendering.
    - Navigate to */sitecore/layout/Sublayouts/ActiveCommerce/Product* in the Content Editor.
    - Change the value of the *Cache Timeout* appropriately (hours:minutes:seconds).



- Adding a custom HTML cache clear in response to the *stock:updated* and *stock:updated:remote* events. Note that this would result in a cache clear on any update to Active Commerce stock data, including when any order is placed.
- Adding a custom client-side stock check on page load.
- Disabling caching of the *Product* rendering.

If you disable caching of the product detail page, ensure that you have sufficient server resources to handle the expected load on these pages by performing load tests which simulate peak traffic levels.

Scaling Active Commerce

You can setup multiple instances of Active Commerce to improve the scalability, performance and security of your solution. Setting up multiple instances in one or more environments (typically, a Content Management (CM) and a Content Delivery (CD) environment), is a general Sitecore best practice. More information can be found by reading the Sitecore Scaling Guide.

The following sections of the Sitecore Scaling Guide are of particular note for Active Commerce:

- Follow instructions in Section 2 carefully for configuring the CM and CD environments. In particular, ensure that system clocks are synchronized between all CM, CD, **and SQL Server** servers.
- Active Commerce is dependent on the Sitecore Link Database. Section 3.6.1 defines requirements for the Link Database to function in a scaled environment. In particular, the publishing database must utilize the same name across environments, and the *core* database should be shared or synchronized.

In a load balanced environment with multiple CD servers, you can utilize database-persisted session state to avoid the need to configure “sticky” sessions. See HOW TO: Configure SQL Server to Store ASP.NET Session State. Sitecore 7.5 and later requires either SQL Server or MongoDB based sessions in a production environment.

For general performance tuning and implementation scaling tips, you should also reference the Sitecore CMS Performance Tuning Guide and implement the practices suggested within.

The following details additional server-specific configuration steps.

Content Management (CM) Server Configuration

1. Follow the Sitecore Scaling Guide for Content Management server configuration.
2. Add a “publicHostName” attribute to the site definition. The value should be set to the hostname of your CD server. This will be used (instead of the “targetHostName” or “hostName”) wherever a url is generated and exposed publicly via the CM server,

including urls in the product export and also any emails sent out (e.g., scheduled tasks).

3. Set the “enableAnalytics” attribute of the site definition to false. For more information, see the Engagement Analytics Configuration Reference Guide (section 3.6).

Content Delivery (CD) Server Configuration

1. Follow the Sitecore Scaling Guide for Content Delivery server configuration.
2. Enable the Active Commerce SwitchMasterToWeb.config file. This is done by removing the “.example” from the “xActiveCommerce.SwitchMasterToWeb.config.example” file (located under App_Config/Include). This include file will remove Active Commerce references to the Master database.

Active Commerce Database Scaling

When deploying Active Commerce, it is always recommended that you load test your application with realistic traffic levels. If load testing reveals an unacceptable level of deadlocking in the Active Commerce NHibernate database, the following steps can help scale your implementation:

- Ensure you have a maintenance plan in place which rebuilds the indexes on the database on at least a weekly basis. This is a best practice for your Sitecore databases as well. See the Sitecore CMS Performance Tuning Guide.
- Enable *Snapshot Isolation* and *Read Committed Snapshot* options on the Active Commerce database. Before doing so, be sure you understand the impact of this on the *tempdb* and on row locking of select statements. See Microsoft TechNet article Choosing Row-versioning Based Isolation Levels.

Miscellaneous	
Allow Snapshot Isolation	True
ANSI NULL Default	False
ANSI NULLS Enabled	False
ANSI Padding Enabled	False
ANSI Warnings Enabled	False
Arithmetic Abort Enabled	False
Concatenate Null Yields Null	False
Cross-database Ownership Chaining Enabled	False
Date Correlation Optimization Enabled	False
Is Read Committed Snapshot On	True
Numeric Round-Abort	False
Parameterization	Simple
Quoted Identifiers Enabled	False
Recursive Triggers Enabled	False
Trustworthy	False
VarDecimal Storage Format Enabled	True

Database Properties Dialog in SQL Server Management Studio

[1] Sitecore install package can be obtained from your Active Commerce representative.