Sitecore E-Commerce Services 1.2

Developer's Cookbook

Configuring and Developing with Sitecore E-Commerce Services

Table of Contents

Chapter 1

Introduction

This document contains a technical overview of the Sitecore E-Commerce Services (SES). It also describes how to use the Unity application block to configure SES, the SES programming contracts, and includes instructions for configuring SES components.

You can use Sitecore to manage multiple websites. You can configure SES to use different data stores for each managed website. For example, different managed websites can store product, order, and other business information in different locations in Sitecore, and in different external systems.

This document contains the following chapters:

- **Chapter 1 — Introduction**
  This chapter contains a brief description of this manual.

- **Chapter 2 — SES Technical Overview**
  This chapter contains a description of the domain model, the Unity application block, and Sitecore E-Commerce Services product management system.

- **Chapter 3 —**
  Adding Customized Product Search Criteria
  This chapter describes how to extend the product search feature in SES.

- **Chapter 4 — SES Core Configuration**
  This chapter describes the configurable elements in SES.

Chapter 2

SES Technical Overview

This chapter provides a technical overview of Sitecore E-Commerce Services, including the domain model, the Unity dependency injection container, and information about how Sitecore E-Commerce Services manages product information.

This chapter contains the following sections:

- The SES Domain Model

- Unity Application Block Overview

- SES Product Management

2.1 The SES Domain Model

The SES domain model is an API layer that defines contracts to abstract SES functionality, such as product, customer, and order information storage. The `Sitecore.Ecommerce.DomainModel` namespace in the `Sitecore.Ecommerce.DomainModel.dll` assembly contains the SES domain model.

The default implementation of the SES domain model stores data as items in the Sitecore content tree. For example, a product definition item describes each product that the website sells, and the complete SES purchasing process results in a new order definition item in the content tree. You can replace elements of the domain model, and you can use different implementations based on logical conditions. Multiple managed websites can share implementations of the domain model and the data that those implementations abstract, or each managed website can use different implementations and data.

To integrate external systems with SES, you can implement processes that use the default implementation of the domain model to import data into Sitecore, or you can replace components of the SES domain model with custom implementations that access external systems directly.

SES includes a sample implementation that uses presentation components developed for the Web Forms for Marketers module to provide a complete online store. For more information about the Web Forms for Marketers module, see http://sdn.sitecore.net/Products/Web%20Forms%20for%20Marketers.aspx.

You can use the example implementation, or you can learn how to implement a custom solution using the code that it contains.

Important
Whenever possible, use contracts in the domain model rather than the concrete implementations of those contracts.

2.2 Unity Application Block Overview

SES uses the Unity application block (Unity) to support customization and integration with such external systems. The Unity application block is a lightweight, extensible dependency injection container, which among other features, provides symbolic names for different implementations of various SES features described by the domain model. Dependency injection is a strategy for specifying relations between types in object-oriented applications. Dependency injection provides a form of inversion of control, moving logic for type specification from code to the dependency injection container. Unity injects the appropriate types into the application at runtime to allow the use of different implementations of a single function depending on configuration, conditions, and code. Unity provides constructor injection, property injection, and method call injection. The Unity container works like a factory to instantiate objects in a manner similar to the providers pattern, but with greater flexibility.

For more information about the Unity Application Block, see http://unity.codeplex.com/.

Unity can designate the software components an application will use, and which software components other components can use. Complex objects typically depend on other objects. Unity helps to ensure that each object correctly instantiates and populates the right type of object for each such dependency.

The Unity architecture supports the loose coupling of application components. SES developers can reference relatively abstract types, and Unity injects the appropriate implementations at runtime.

The Unity application block provides the following benefits for developers who customize and extend SES:

Flexibility

Unity allows developers to specify types and dependencies through configuration and at runtime, deferring configuration to the container.

Simplification

The simplification of object instantiation code, especially for hierarchical structures with dependencies, which simplifies application code.

Abstraction

The abstraction of requirements through type information and dependencies.

Service locator capability

SES supports the persistence of the container, such as within the ASP.NET session or application, or through Web services or other techniques. For more information about the Service Locator pattern, see http://msdn.microsoft.com/en-us/library/ff649658.aspx.

With Unity, you can easily configure SES to use custom implementations for specific features, including:

- Configuration components, such as general settings.

- Business objects, such as customers and orders.

- Business logic, such as sending e-mail or locating a product.

- Payment providers, such as specific payment gateways.

- Internal logic, such as mapping in-memory storage to long-term storage.

With SES and Unity, you can use different implementations of an interface or descendants of an abstract or another base class to achieve a common function for different managed websites. For example, different managed websites can access customer information from different systems. Unity makes it easier to integrate external business systems that are typically involved in ecommerce into a SES implementation.

In this document, the term *contract* refers to an interface that a class implements, an abstract or concrete base class from which it inherits. The term *implementation* refers to a class that implements a given contract.

The SES entities defined with Unity include:

- Contracts define Application Programming Interfaces (APIs).

- Implementations define concrete instances that implement contracts.

- Mappings configure which implementations to inject.

- Dependencies configure which dependent implementations to inject.

Unity allows you to define contracts using interfaces, abstract classes, and concrete classes. An implementation can implement an interface, inherit from an abstract base class, inherit from a concrete base class, or inherit directly from `System.Object`. A contract defined by a concrete class can serve as its own implementation.

Note

To work with the SES APIs that depend on the Unity application block, you may need to add a reference to the `Microsoft.Practices.Unity.dll` assembly in the `/bin` subdirectory to the Visual Studio project. Remember to set the *Copy Local* property of the reference to *False*.

The following diagram describes the SES API layers. The example UI pages access APIs in the domain model, and SES uses Unity to resolve those API calls to concrete implementations of those contracts.
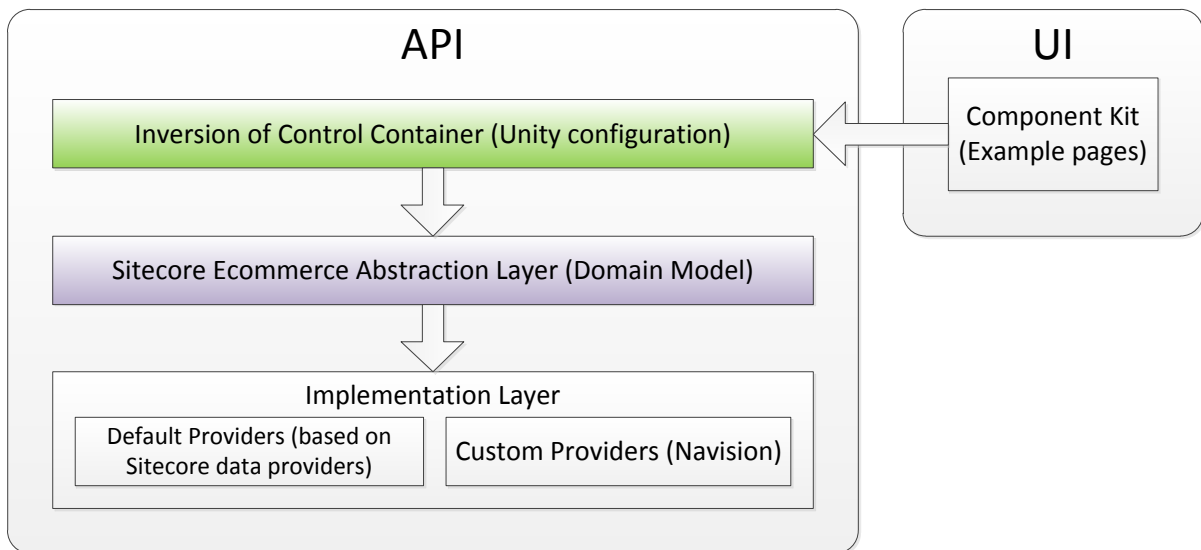
2.2.1 The Unity Configuration File

SES manages the Unity configuration in the `/App_Config/Unity.config` file. The Unity configuration file consists of two main parts:

- Each `/unity/aliases` element in the Unity configuration file defines a type alias, which provides a symbolic name for a contract or implementation, such as an interface, an abstract type, or a concrete type.

- Each `/unity/container/register` element in the Unity configuration file specifies a concrete type that implements a contract identified by a `/unity/alias` element.

2.2.2 The initialize Pipeline

SES adds two processors to the `initialize` pipeline defined in the `Web.config` file.

> **Note**
> SES uses the `/App_Config/Include/Sitecore.Ecommerce.config` file to extend the `Web.config` file.

Based on Unity configuration, the `ConfigureEntities` processor in the `initialize` pipeline initializes the entities that SES uses. This processor loads an inversion of the control container into the SES context as a static resource in memory.

The `RegisterEcommerceProviders` processor in the `initialize` pipeline initializes various SES implementations.

2.2.3 Dependency Injection

With Unity, you can designate dependencies between entities.

For example, for search features, the `IOrderManager` contract depends on an object that implements the `ISearchProvider` contract. The following excerpts from the Unity configuration define that the default implementation of the `IOrderManager` contract uses the `FastQueryItemSearchProvder` implementation of the `ISearchProvider` interface by passing an instance of `FastQueryItemSearchProvder` to the constructor for that `IOrderManager`.

```xml
<unity>
  ...
  <alias alias="IOrderManager"
    type="Sitecore.Ecommerce.DomainModel.Orders.IOrderManager`1..."/>
  ...
  <alias alias="ISearchProvider"
    type="Sitecore.Ecommerce.Search.ISearchProvider, Sitecore.Ecommerce.Kernel"/>
  ...
  <alias alias="OrderManager"
    type="Sitecore.Ecommerce.Orders.OrderManager`1, Sitecore.Ecommerce.Kernel"/>
  ...
  <register type="ISearchProvider" mapTo="FastQuerySearchProvider"
    name="FastQuerySearchProvider" />
  ...
  <container>
    ...
    <register type="IOrderManager" mapTo="OrderManager">
      <lifetime type="perthread" />
      <constructor>
        <param name="searchProvider">
          <dependency name="FastQuerySearchProvider"/>
        </param>
      </constructor>
    </register>
    ...
  </container>
</unity>
```

> **Note**
> To indicate generic type parameters in the Unity configuration, append a single end quotation mark ("`") followed by a number.

For example, to specify the `Sitecore.Ecommerce.DomainModel.Currencies.ICurrencyManager<TTotals, TCurrency>` interface that requires two generic types, specify a type signature followed by a back quote and the number 2:

```
Sitecore.Ecommerce.DomainModel.Currencies.ICurrencyManager2
```

2.2.4 How to Resolve a SES Component

Use the `Sitecore.Ecommerce.Context.Entity.Resolve()` method to resolve a type configured with Unity. Pass the type of the contract to the method as a generic type parameter. For example, to access the default implementation of the `IProductRepository` contract:

```
using Sitecore.Ecommerce;
...
Sitecore.Ecommerce.DomainModel.Products.IProductRepository productRepository =
  Sitecore.Ecommerce.Context.Entity.Resolve
  <Sitecore.Ecommerce.DomainModel.Products.IProductRepository>();
```

The signature of the `Resolve()` method is an extension method in the `Sitecore.Ecommerce.IoCContainerExtensions` class.

To use this signature, add the following line at the top of your class:

```
using Sitecore.Ecommerce;
```

Alternatively, fully designate this implementation of the `Resolve()` method:

```
Sitecore.Ecommerce.DomainModel.Products.IProductRepository productRepository =
  Sitecore.Ecommerce.IoCContainerExtensions.Resolve
  <Sitecore.Ecommerce.DomainModel.Products.IProductRepository>
  (Sitecore.Ecommerce.Context.Entity);
```

To access a named entity, pass the name of an entity as the first parameter to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method.

For example, to retrieve the `IProductRepository` implementation called `MyProductRepository`:

```
Sitecore.Ecommerce.DomainModel.Products.IProductRepository myProductRepository =
  Sitecore.Ecommerce.Context.Entity.Resolve
  <Sitecore.Ecommerce.DomainModel.Products.IProductRepository>("MyProductRepository");
```

For more information about how SES resolves types, see the section *How to Configure* .

2.2.5 How to Add an Implementation to the Unity Configuration

To add an additional implementation of a contract to the Unity configuration:

1. In the Visual Studio project, create a class that implements the required interface or inherits from the appropriate base class.

2. In the Unity configuration, insert an additional `/unity/alias` element.

3. In the new `/unity/alias` element, set the `alias` attribute to a unique alias.

4. In the new `/unity/alias` element, set the `type` attribute to the signature of the .NET class.

For more information about how to configure SES to use the implementation, see the sections *How to Replace a SES Component* and *How to Configure* .

2.2.6 How to Add a Contract to the Unity Configuration

To add a contract to the Unity configuration:

1. In the Unity configuration file, add a `/unity/alias` element. Set the `alias` attribute of the new `/unity/alias` element to a unique value that identifies the contract. Set the `type` attribute of the new `/unity/alias` element to the .NET type of the interface or class that defines the contract. For example:

```
<alias alias="MyType" type="Namespace.MyType, MyAssembly"/>
```

2. If the type that defines the contract does not also serve as the implementation of that contract, then configure one or more implementations of the contract.

For more information about how to define an implementation of the contract, see the section *How to Add an Implementation to the Unity Configuration*.

2.2.7 How to Replace a SES Component

To configure SES to use a custom component for a feature:

1. In the Unity configuration, add a `/unity/alias` element to register the new implementation.

For more information about how to add an implementation to the Unity configuration, see the section *How to Add an Implementation to the Unity Configuration*.

2. In the Unity configuration, set the `mapTo` attribute of the `/unity/container/register` element with a value for the `type` attribute that specifies the value of the `alias` attribute of the `/unity/alias` element that defines the contract or implementation to the value of the `alias` attribute of the new `/unity/alias` element that specifies the implementation.

In the `/unity/container/register` element, the `type` attribute identifies the alias of the contract, the `mapTo` attribute identifies the alias of the implementation, and the optional `name` attribute defines a token with which to resolve the implementation in API calls.

2.2.8 How to Configure Unity for Multiple Implementations of the Same Contract

In Unity, you can define several implementations of a contract.

To use different implementations of contracts for different managed websites:

1. Add any required implementations to the Unity configuration.

For more information about how to add an implementation to the Unity configuration, see the section *How to Add an Implementation to the Unity Configuration*.

2. For each implementation, in the Unity configuration, create a `/unity/container/register` element.

Note
To create the new `/unity/container/register` element, copy an existing `/unity/container/register` element that is associated with the same contract.

3. in the new `/unity/container/register` element, set a unique value for the `name` attribute.

For example, you can configure the `/unity/container/register` elements in the Unity configuration to:

- Make SES use the `ProductCategory` implementation with the alias `MyProductCategory` for the managed websites called `site2` and `site3`.

- Use the default `ProductCategory` implementation with the alias `SitecoreProductCategory` for all the other managed websites.

```xml
<!-- contract -->
<alias alias="ProductCategory"
  type="Sitecore.Ecommerce.DomainModel.Products.ProductCategory..." />
<!-- implementations -->
<alias alias="SitecoreProductCategory"
  type="Sitecore.Ecommerce.Products.ProductCategory, Sitecore.Ecommerce.Kernel" />
<alias alias="MyProductCategory" type="MyNamespace.ProductCategory, MyAssembly" />
<!-- uses -->
<container>
  <register type="ProductCategory" mapTo="SitecoreProductCategory">
    <interceptor type="VirtualMethodInterceptor" />
```

```
      <policyInjection />
   </register>
   <register type="ProductCategory" mapTo="MyProductCategory" name="site2">
     <interceptor type="VirtualMethodInterceptor" />
     <policyInjection />
   </register>
   <register type="ProductCategory" mapTo="MyProductCategory" name="site3">
     <interceptor type="VirtualMethodInterceptor" />
     <policyInjection />
   </register>
```

Use the following setting in Unity to access a named implementation by passing the name of the implementation with the site name to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method:

```
   <register type="ProductCategory" mapTo="MyOtherProductCategory"
     mapTo="MyProductCategory" name="site3MyOtherProductCategory">
```

Use the following code to access the `ProductCategory` implementation called `site3MyOtherProductCategory`:

```
   Sitecore.Ecommerce.DomainModel.Products.ProductCategory productCategory =
     Sitecore.Ecommerce.Context.Entity.Resolve
```

```
<Sitecore.Ecommerce.DomainModel.Products.ProductCategory>("site3MyOtherProductCategory");
```

If you pass a parameter to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method and if an implementation exists, Unity injects that type.

If you do not pass a parameter to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method, Unity injects the default implementation of the contract.

Note
If no default implementation exists, Unity raises an error.

2.3 SES Product Management

SES stores product information in repositories that typically exist outside of the content tree of any managed website, thereby allowing multiple websites to share product repositories.

SES provides logic to generate product URLs that appear to be within the website, and enhances the logic that Sitecore applies to determine and present the product definition items associated with these URLs.

2.3.1 Product URLs and Product Resolution

SES adds the `ProductResolver` processor after the default `ItemResolver` processor in the `httpRequestBegin` pipeline defined in the `Web.config` file. If the default `ItemResolver` cannot resolve the context item from the requested URL, then the `ProductResolver` uses a `VirtualProductResolver` to attempt to determine a product from the requested URL. If the `VirtualProductResolver` can determine the product, it sets the context item to the item that defines that product.

How to Specify the Product URL Format

To specify the product URL format for a managed website or branch:

1. In the **Content Editor**, in the home item for the managed Web site or the root item of the branch, select the **System** section,
2. In the **Display Products Mode** field, select one of the `ProductUrlProcessor` definition items.

Note
If the **Display Products Mode** field does not exist for an item, add the `Ecommerce/Product Categories/Product Search Group Folder` data template to the base templates for the data template associated with the item.

SES uses the value of the **Display Products Mode** field in the nearest ancestor of the context item that defines a value for that field. For example, given the URL `/products.aspx`, if the `<home>/products` item has a value for **Display Products Mode** field, SES applies that value, otherwise SES applies the value of the **Display Products Mode** field in the home item.

2.3.2 Product Presentation

The URLs of SES product pages map to items that do not define layout details. For more information about the layout details, see the Presentation Component Reference at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Reference.aspx.

Important
Do not update the layout details for a product or the standard values of a data template for products.

Note
To preview the presentation of a product, use the **Page Editor** or the **Preview** viewer to navigate from a page that links to the product to the product detail page.

SES replaces the `InsertRenderings` processor in the `renderLayout` pipeline defined in the `Web.config` file with the `ProcessProductPresentation` processor. When processing an HTTP request for a product page, the `ProcessProductPresentation` processor applies the layout details from the item that is specified in the **Product Detail Presentation Storage** field.

This field is in the nearest ancestor of the logical parent item of the virtual product item that defines a value for that field. For example, in the `/products/product_name.aspx` URL, if the

`<home>/products` item has a value in the **Product Detail Presentation Storage** field, SES applies that value, otherwise SES applies the value in the **Product Detail Presentation Storage** field of the *Home* item.

> **Note**
> If the **Product Detail Presentation Storage** field does not appear in an item, add the `Ecommerce/Product Categories/Product Search Group` data template to the base templates of the data template associated with the item.

How to Specify a Product Presentation Format

To specify the presentation format that you want to use to display the products associated with a page:

1. In the **Content Editor**, edit the page definition item.

2. In the page definition item, on the **Content** tab, in the **Products in Category** section, in the **Product Detail Presentation Storage** field, select a product presentation definition item.

How to Update a Product Presentation Format

To update an existing product presentation format:

1. In the **Content Editor**, edit the product presentation definition item. The product presentation definition item is a child of the `/Sitecore/System/Modules/Ecommerce/System/Product Presentation Repository` item.

2. In the product presentation definition item, edit the layout details.

For more information about applying layout details, see the Presentation Component Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Cookbook.aspx.

> **Note**
> You can use access rights to control which users can apply various product presentation formats.

To apply access rights:

1. You can change the type of the **Product Detail Presentation Storage** in the `Ecommerce/Product Categories/Product Search Group` item from **Lookup** to **Droptree**.

2. Create folders under `/Sitecore/System/Modules/Ecommerce/System/Product Presentation Repository` that you can use to store the different groups of presentation format definition items.

3. Apply access rights to those folders.

How to Define a New Product Presentation Format

To define a new product presentation format:

1. In the **Content Editor**, select the `/Sitecore/System/Modules/Ecommerce/System/Product Presentation Repository` item.

2. In the **Content Editor**, insert a new product presentation definition item using the **Ecommerce/Product/Product Presentation Storage** data template.

3. In the new product presentation definition item, update the product presentation format.

   For more information about updating the product presentation format, see the section *How to Update a Product Presentation Format*.

4. Optionally, you can apply the new product presentation format to the existing pages. For more information about applying a product presentation format, see the section *How to Specify a Product Presentation Format*.

Chapter 3

Adding Customized Product Search Criteria

This chapter describes how to extend the product search feature in SES. It shows how to customize the search options and how to have more control over product presentation in both of the frontend and backend. By the frontend we mean the display of search results for the page visitor and by the backend we mean the Content Editor and Template Manager.

This chapter contains the following sections:

- The Need for Product Search Configuration and Extensibility

- Extending the Product Search Group Template

- Extending the Resolve Strategy

- Extending the Product Search Catalog

3.1 The Need for Product Search Configuration and Extensibility

To illustrate the need for changing product search, consider the case of a camera and photographic supply webshop that is divided into sections that contain different models, categories, proficiency levels, and interrelated products. A vendor will not usually show all the cameras on the same page but they will rather show each camera with a group of products of the same proficiency level. For example, professional cameras are usually shown with professional lenses and others accessories. Moreover, one product can be shown in multiple groups.

This chapter explains how to create a different classification than the one used in the repository.
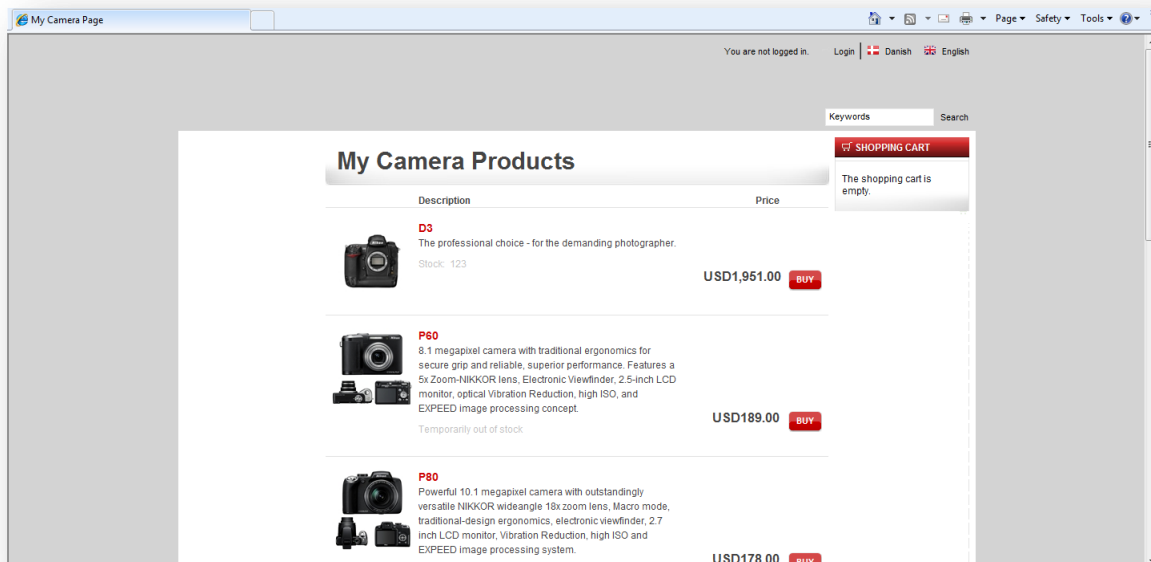
3.2 Extending the Product Search Group Template

This section describes how to classify a product according to your business needs. You must create or edit the classifications that you need in the *Product Search Group* template.

A convenient starting point is to extend this template with additional fields for storing search criteria. You can use the *Product Search Group* template to define a category structure that reflects the way the products are presented on the front end and not in the structure of the repository.

This section describes how to use the Content Editor to add a new search criterion to the *Product Search Group* template by applying an additional filter to the products selected.

To add a new search criterion to the *Product Search Group* template:

1. Log in into the **Content Editor** and navigate to the *Product Search Group* template.
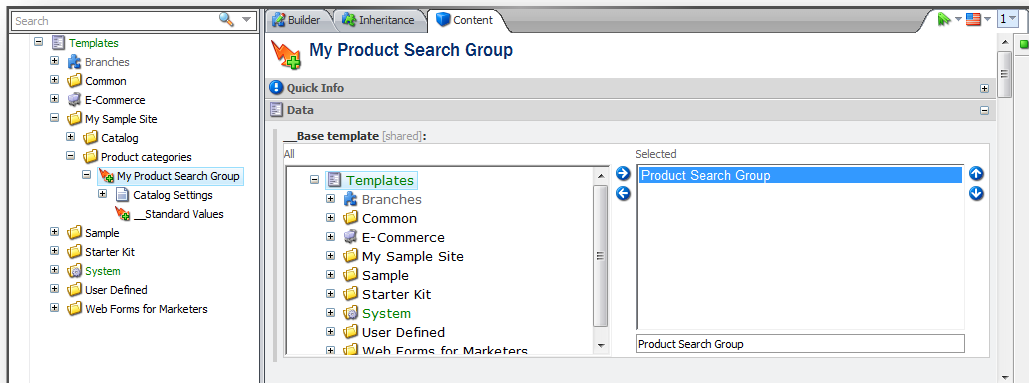
2. In the **Content** tab, create a new template that inherits from the *Product Search Group* template and call it *My Product Search Group*.

3. Click the **Builder** tab and in the **Catalog Settings** section, add a new criterion, call it *Search Treelist*.
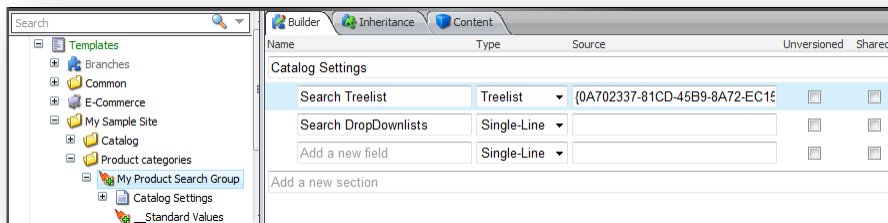
4. In the **Type** field, select *Treelist* as the type. You must select *Treelist* as the type if you want to select multiple folders from the product repository.

5. In the **Source** field, enter the path (or GUID) of the product repository.

6. Create a page item that inherits from the *My Product Search Group* template and call it *mytest*.

   You should now be able to select the domain for your search from the treelist.

In the following image, *Cameras* is the selected domain.
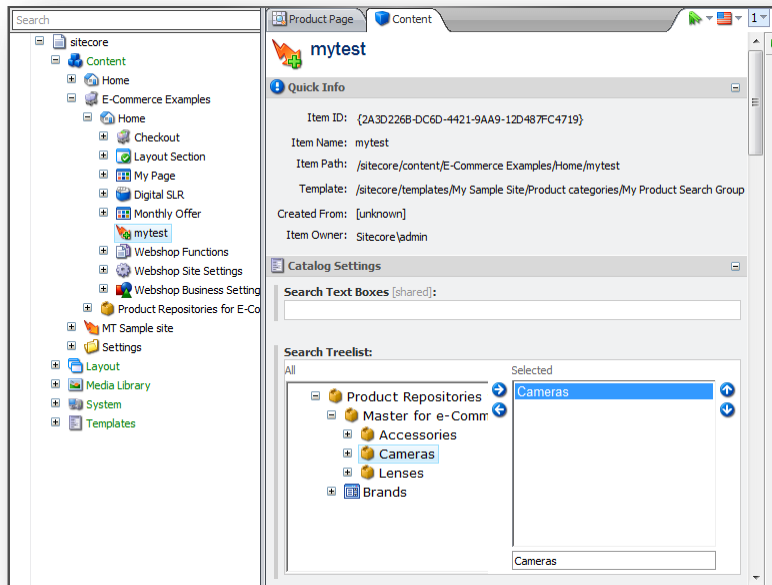
3.3 Extending the Resolve Strategy

To search for products in the domain selected in the Treelist control, you must:

- Extend the `DatabaseCrawler` to index this product category *parent* folder.

- Extend the `QueryCatalogProductResolveStrategy` class to find the products based on a particular product category folder.

Extending the DatabaseCrawler

Essentially, you use the `DatabaseCrawler` class to build product and web indexes.

The `Sitecore.Search.DatabaseCrawler` class scans a specific repository such as a database or file system, extracts information, and stores it in a search index. It then makes this information available to Sitecore Search.

The `Sitecore.Search.DatabaseCrawler` class performs the following functions:

- `IndexAllFields` — Extracts data from a specific document that is requested by the crawler or the monitor. The data extracted consists of metadata and content.
    - `Metadata` — The Indexer extracts metadata that the system understands. You can filter and prioritize the metadata, for example, by using the **_name** or **_template** field.
    - `Content` — The Indexer also extracts body content and prioritizes it. You can use boost to prioritize the content in the document. This is usually only applied to a single field, giving the document a single prioritization.

- `DatabaseCrawler` — Traverses the storage system and uses the indexer to populate the search index.

- `MonitorChanges` — Monitors changes in the repository and updates the search index.

The following code shows how to extend the `DatabaseCrawler` class to add a special field to a document in Lucene that represents the parent category folder in SES:

1. In Visual Studio, create a new project and call it `Sample1`.

2. Add the following class to the project and call it `SampleDatabaseCrawler`.

```
namespace Sample1.Kernel.Search
{
    using Lucene.Net.Documents;
    using Sitecore.Data;
    using Sitecore.Data.Items;

    // SampleDatabaseCrawler class is inherited from
Sitecore.Ecommerce.Search.DatabaseCrawler
    // Created so we can add the needed field to the Lucene index products when
resolving products based on which product category folder they are located in
    public class SampleDatabaseCrawler : Sitecore.Ecommerce.Search.DatabaseCrawler
    {
        // Overridden method for adding special fields to the Lucene product index
        // <param name="document">The Lucene document to add a new field to</param>
        // <param name="item">the item to get the value from</param>
        protected override void AddSpecialFields(Document document, Item item)
        {
        //Call the base class for setting the base special fields on the Lucene
document
            base.AddSpecialFields(document, item);
            //Add the field _parent to the document for the Luceneindexeer
            document.Add(CreateTextField("_parent", ShortID.Encode(item.Parent.ID)));
        }
    }
}
```

Once you have extended the `DatabaseCrawler` class to create the **_parent** field for the Indexer, you are ready to extend the search strategy to use this index.

Extending the ICatalogProductResolveStrategy Class

The `ICatalogProductResolveStrategy` contract defines the way that SES retrieves the products that are displayed on a given webpage.

The implementation of this contract:

1. Reads search criteria form the current item based on the product search group template.
2. Builds and executes a search using the criteria against the product repository.
3. Returns the list of products to display.

The following classes are the default Implementations of the `ICatalogProductResolveStrategy` contract:

ProductListCatalogResolveStrategy

You can use this class to retrieve the products that have been manually selected and associated with the webpage item. (`sitecore/system/Modules/Ecommerce/System/Product Selection Method`).

QueryCatalogProductResolveStrategy

You can use this class to retrieve the products that results from the search and store the query parameters on the webpage item (`sitecore/system/Modules/Ecommerce/System/Product Selection Method`). It implements the `CatalogProductResolveStrategyBase` class which implements the `ICatalogProductResolveStrategy` interface.

You can also extend the class that represents the `QueryCatalogProductResolveStrategy` to accommodate the search:

1. In Visual Studio, open the project called `Sample1` that you created in the last subsection.
2. Add the following class and name it `SampleQueryCatalogProductResolveStrategy`.

```
namespace Sample1.Kernel.Catalogs
{
    using System.Collections.Generic;
    using System.Linq;
    using Sitecore.Data;
    using Sitecore.Data.Items;
    using Sitecore.Diagnostics;
    using Sitecore.Ecommerce;
    using Sitecore.Ecommerce.Configurations;
    using Sitecore.Ecommerce.Search;

    // <summary>
    // SampleQueryCatalogProductResolveStrategy class is inherited from
       Sitecore.Ecommerce.Catalogs.QueryCatalogProductResolveStrategy
    // Created to implement the functionality to resolve products based on which
       repository folder they are located in.
    // </summary>
    public class SampleQueryCatalogProductResolveStrategy :
                Sitecore.Ecommerce.Catalogs.QueryCatalogProductResolveStrategy
    {
        // <summary>
        // The Search TreeList field name
        // </summary>
        private read only string searchTreelistFieldName;

        // <summary>
        // Initializes a new instance of the SampleQueryCatalogProductResolveStrategy
           class.
        // </summary>
        // <param name="searchTextBoxesFieldName">Names of the searchtextboxes</param>
        // <param name="searchChecklistsFieldName">Names of the Checkboxes</param>
```

```csharp
        // <param name="searchTreelistFieldName">name of the treelist field</param>
        public SampleQueryCatalogProductResolveStrategy(string
                searchTextBoxesFieldName, string searchChecklistsFieldName, string
                searchTreelistFieldName)
                : base(searchTextBoxesFieldName, searchChecklistsFieldName)
        {
            // Testing for not null or empty
            Assert.ArgumentNotNullOrEmpty(searchTreelistFieldName,
            "searchTreelistFieldName");
            // Assigning to local variable
            this.searchTreelistFieldName = searchTreelistFieldName;
        }
        // <summary>
        // Overridden method for building the search query for searching the Lucene
           index
        // </summary>
        // <param name="catalogItem">the catalog item we are resolving from (product
           catalog)</param>
        // <returns>The query we build for searching</returns>
        protected override Query BuildSearchQuery(Item catalogItem)
        {
            // Let's resolve the actual field on the current catalog item
            string searchTreelistFieldText =
                    catalogItem[this.searchTreelistFieldName];
            // If nothing defined, returning "error in setup" on template
            if (string.IsNullOrEmpty(searchTreelistFieldText))
            {
                return default(Query);
            }
            // Calling the base class for getting all the query fields defined in the
               base class
            Query query = base.BuildSearchQuery(catalogItem);
            // Getting the configuration from SES
            BusinessCatalogSettings businessCatalogSettings =
            Context.Entity.GetConfiguration<BusinessCatalogSettings>();
            // Testing if configuration is set - if not, fail in setup by user.
            Assert.IsNotNull(businessCatalogSettings, GetType(), "Business Catalog
            settings not found.", new object[0]);
            // Getting the root from where products are located (product repository)
            Item productRepositoryRootItem =
                    catalogItem.Database.GetItem(businessCatalogSettings.ProductsLink);
            // Testing if the root is set - if not, this is a failure from the user.
            Assert.IsNotNull(productRepositoryRootItem, "Product Repository Root Item
            is null.");
            // If the query is empty, we need to add some stuff to it
            if (query == default(Query))
            {
                query = new Query { SearchRoot =
                        productRepositoryRootItem.ID.ToString() };
            }

            // Let´s parse the field from the current catalog items
            if (!string.IsNullOrEmpty(searchTreelistFieldText))
            {
                this.ParseTreelistField(searchTreelistFieldText, ref query);
            }

            return query;
        }

        // <summary>
        // Function for parsing TreeList to query on the catalog item
        // </summary>
        // <param name="ids">string with | separated list of categoryfolder
           Ids</param>
        // <param name="query">the query to append to</param>
        protected virtual void ParseTreelistField(string ids, ref Query query)
        {
            // Creating a list if more than one folder is defined
            List<string> folders = new List<string>();
            if (ids.Contains("|"))
            {
                folders.AddRange(ids.Split('|'));
            }
```

```
            else
            {
                folders.Add(ids);
            }
            Query sub = new Query();
            int count = 0;
            // Iterating through each folder where there's a Sitecore ID
            foreach (string s in folders.Where(ID.IsID))
            {
                // Appending the value of the folder to the query and telling the
                    query to search for the field _parent in the product Lucene index
                sub.AppendField("_parent", ShortID.Encode(s), MatchVariant.Exactly);
                // If more than one - we must add an "Or" to the query
                if (count < (folders.Count - 1))
                {
                    sub.AppendCondition(QueryCondition.Or);
                }
                count++;
            }
            // Appending the built query to the main query
            query.AppendSubquery(sub);
        }
    }
}
```

Configuring SES and Lucene

To register the newly created database crawler and the resolve strategy, you must configure the search in two files — `Sitecore.Ecommerce.config` and `Unity.config`.

1. In the `Sitecore.Ecommerce.config` file, under the `indexes` element, in the `Configuration` element, add the following index:

```xml
            <!-- Products index - Used by SES for resolving products - should not be
                used on frontend for searching-->
            <index id="products" type="Sitecore.Search.Index, Sitecore.Kernel">
              <param desc="name">$(id)</param>
              <param desc="folder">__products</param>
              <Analyzer type="Sitecore.Ecommerce.Search.LuceneAnalyzer,
               Sitecore.Ecommerce.Kernel" />
              <locations hint="list:AddCrawler">
                <master type="Sample1.Kernel.Search.SampleDatabaseCrawler, Sample1">
                  <Database hints="master">master</Database>
                  <!-- Repository root where products are stored-->
                  <!--<Root>{054AEC0D-9D92-4C3A-80AC-A0E78773EAB7}</Root>-->
                  <!-- Repository root where SES products are stored-->
                  <Root hints="masterRoot">{502EA9FA-19E7-4DA5-8EA4-56C374AED45B}</Root>
                  <Tags hint="master products">master products</Tags>
                </master>
                <web type="Sample1.Kernel.Search.SampleDatabaseCrawler, Sample1">
                  <Database hints="web">web</Database>
                  <!-- Repository root where products are stored-->
                  <!--<Root>{054AEC0D-9D92-4C3A-80AC-A0E78773EAB7}</Root>-->
                  <!-- Repository root where SES products are stored-->
                  <Root hints="webRoot">{502EA9FA-19E7-4DA5-8EA4-56C374AED45B}</Root>
                  <Tags>web products</Tags>
                </web>
              </locations>
            </index>
```

2. In the `Search.config` file, in the `Unity` element, add the following alias.

```xml
    <alias alias="SampleQueryCatalogProductResolveStrategy" ´
     type="Sample1.Kernel.Catalogs.SampleQueryCatalogProductResolveStrategy, Sample1"/>
```

3. In the `Search.config` file, in the `Container` element, add the following registry.

```xml
    <register type="ICatalogProductResolveStrategy"
     mapTo="SampleQueryCatalogProductResolveStrategy" name="My product Repository query">
            <lifetime type="singleton" />
            <constructor>
              <param name="searchTextBoxesFieldName">
                <value value="Search Text Boxes"/>
```

```xml
      </param>
      <param name="searchChecklistsFieldName">
        <value value="Search Checklists"/>
      </param>
      <param name="searchTreelistFieldName">
        <value value="Search Treelist"/>
      </param>
    </constructor>
  </register>
```

3.4 Extending the Product Search Catalog

This section describes how to extend the *Product Search Catalog* to accommodate the product search extension in the backend. In other words, it describes how to make the search results visible in the Content Editor.

To extend the Product Search Catalog, you must:

- Extend the `CatalogQueryBuilder`.

- Create a products source.

- Reference this source in the Content Editor.

Extending the CatalogQueryBuilder

The `CatalogQueryBuilder` class builds the search query that is used by SES when querying the product repository.

Note
You can only use the `CatalogQueryBuilder` in the product catalog.

To extend the `CatalogQueryBuilder` class to reflect the search result in the backend:

1. In Visual Studio, open the project called `Sample1` that you created earlier.

2. Add the following class to the project and name it `CatalogQueryBuilder`.

```
namespace Sample1.Shell.Applications.Catalogs.Models.Search
{
    using System.Linq;
    using Sitecore.Ecommerce.Search;
    using Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search;
    using Sitecore.Ecommerce.Configurations;
    using Sitecore.Ecommerce;
    using Sitecore.Diagnostics;
    using System.Collections.Generic;
    using Sitecore.Data;

    // <summary>
    // CatalogQueryBuilder inheriting from
    //   Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search.CatalogQueryBuilder
    // Class is used for implementing functionality for resolving our result on the
    //   product page in the sitecore content editor.
    // </summary>
    public class CatalogQueryBuilder :
      Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search.CatalogQueryBuilder
    {
        // <summary>
        // Buildquery function overridden - used for building the actual query for
        //    searching
        // </summary>
        // <param name="options">Seachoptions</param>
        // <returns>The query to be used for search</returns>
        public override Query BuildQuery(SearchOptions options)
        {
            // Get the base query - we still need the functionality from there
            var query = base.BuildQuery(options);
            // Requesting the id of the item we are resolving from in the content
            //    editor
            var id = Sitecore.Context.Request.QueryString.Get("id");
            // Getting the catalog item from the DB
            var catalogItem = Database.GetDatabase("master").GetItem(new ID(id));
            // Let's resolve the actual field on the current catalog item
            var searchTreelistFieldText = catalogItem["Search Treelist"];
            // Returning (error in set up)on the template, if nothing is defined
            if (string.IsNullOrEmpty(searchTreelistFieldText))
            {
```

```csharp
                return query;
            }
            // Getting the configuration from SES
            var businessCatalogSettings =
            Context.Entity.GetConfiguration<BusinessCatalogSettings>();
            // Testing if configuration is set - if not, fail in setup by user
            Assert.IsNotNull(businessCatalogSettings, GetType(), "Business Catalog
            settings not found.", new object[0]);
            // Getting the root from where products are located (product repository)
            var productRepositoryRootItem =
            catalogItem.Database.GetItem(businessCatalogSettings.ProductsLink);
            // Testing if the root is set - if not this is a fail from the user
            Assert.IsNotNull(productRepositoryRootItem, "Product Repository Root Item
            is null.");
            // If the query is empty - we need to add some stuff to it
            if (query == default(Query))
            {
                query = new Query { SearchRoot =
                productRepositoryRootItem.ID.ToString() };
            }
            // let's parse the treelist field from the current catalog items
            if (!string.IsNullOrEmpty(searchTreelistFieldText))
            {
                ParseTreelistField(searchTreelistFieldText, ref query);
            }
            return query;
        }
        // <summary>
        // Function for parsing treelist to query on the catalog item
        // </summary>
        // <param name="ids">string with | separated list of category folder
            Ids</param>
        // <param name="query">the query to append to</param>
        protected virtual void ParseTreelistField(string ids, ref Query query)
        {
            // Creating a list if more than one folder is defined
            var folders = new List<string>();
            if (ids.Contains("|"))
            {
                folders.AddRange(ids.Split('|'));
            }
            else
            {
                folders.Add(ids);
            }
            var sub = new Query();
            var count = 0;
            // Iterating through each folder where there is a Sitecore ID
            foreach (var s in folders.Where(ID.IsID))
            {
                // Appending the value of the folder to the query and telling the
                    query to search for the field _parent in the product Lucene index
                sub.AppendField("_parent", ShortID.Encode(s), MatchVariant.Exactly);
                // If more than one, we of course need to add a or to the query
                if (count < (folders.Count - 1))
                {
                    sub.AppendCondition(QueryCondition.Or);
                }
                count++;
            }
            // If the query is not empty, we need to be sure to add a AND condition.
            if (!query.IsEmpty())
            {
                query.AppendCondition(QueryCondition.And);
            }
            // Appending the built query to the main query
            query.AppendSubquery(sub);
        }
    }
}
```

Creating a Products Source

The main class that you should use in this scenario is the `ProductsSource` class. You can use the methods in this class to initialize the search, build the query using the `CatalogQueryBuilder` mentioned earlier, and return the result.

To create a products source, you should extend the class called `ProductsSource` — `Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search.ProductsSource`

1. In Visual Studio, open the project named `Sample1` that you created earlier.

    Add the following class to the project and name it `ProductsSource`:

```csharp
namespace Sample1.Shell.Applications.Catalogs.Models.Search
{
    using System.Linq;
    using System.Collections.Generic;
    using Sitecore.Ecommerce.DomainModel.Products;
    using Sitecore.Ecommerce.Search;
    using Sitecore.Ecommerce.Utils;
    using Sitecore.Ecommerce;
    using Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search;
    using Sitecore.Ecommerce.Shell.Applications.Catalogs.Models;

    // <summary>
    // ProductsSource inheriting from
    //    Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search.ProductsSource
    // this class is created so we can call the new query functionality we need for
    //    showing the result in the Sitecore content editor.
    // this class is also referred to on the copy made in Sitecore based on
    //    /sitecore/system/Modules/Ecommerce/Catalogs/Product Catalog
    // </summary>
    class ProductsSource :
          Sitecore.Ecommerce.Shell.Applications.Catalogs.Models.Search.ProductsSource
    {
        // <summary>
        // Gets the entries.
        // </summary>
        // <param name="pageIndex">Index of the page.</param>
        // <param name="pageSize">Size of the page.</param>
        // <returns>Returns Entries</returns>
        public override IEnumerable<List<string>> GetEntries(int pageIndex, int
                        pageSize)
        {
            // Let's get the query
            var builder = new CatalogQueryBuilder();
            var query = builder.BuildQuery(SearchOptions);
            // Let's resolve the product repository
            var productRepository = Context.Entity.Resolve<IProductRepository>();
            // Let's do the search
            var products = productRepository.Get<ProductBaseData, Query>(query,
            pageIndex, pageSize);
            // let's return the result
            return !products.IsNullOrEmpty() ? new
                    EntityResultDataConverter<ProductBaseData>().Convert(products,
                    SearchOptions.GridColumns).Rows : new GridData().Rows;
        }
        // <summary>
        // Gets the entry count
        // </summary>
        // <returns>Returns enties count.</returns>
        public override int GetEntryCount()
        {
            // Let's get the query
            var builder = new CatalogQueryBuilder();
            var query = builder.BuildQuery(SearchOptions);
            // Let's resolve the product repository
            var productRepository = Context.Entity.Resolve<IProductRepository>();
            return productRepository.Get<ProductBaseData, Query>(query).Count();
        }
    }
}
```
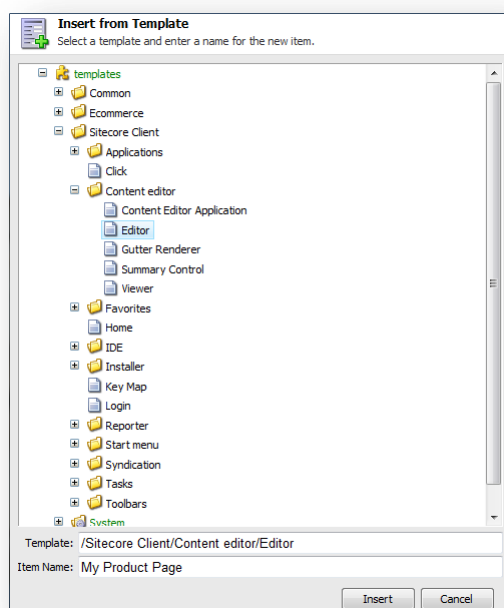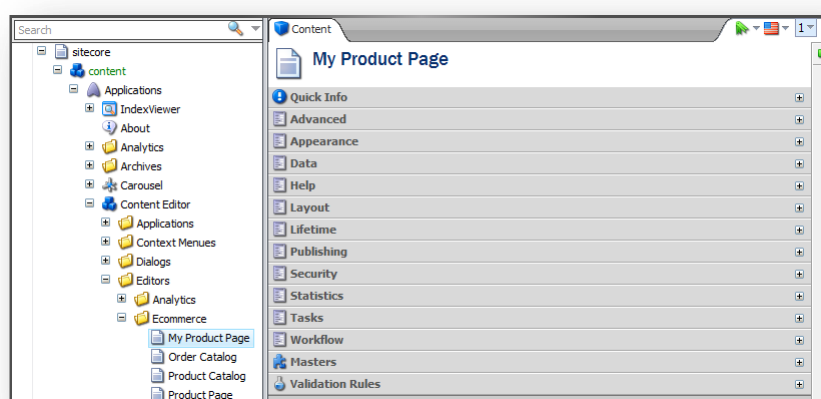
Defining a New Editor in the Core Database

When you create a product catalog, you must also define a new editor in the *Core* database. You place the search catalog in the editor.

To create the editor:

1. Switch to the *Core* database.

2. Log in to the **Content Editor**.

3. Browse to the *My Product Page* item (`Sitecore/content/Content Editor/Ecommerce/My Product Page`) and insert from template.

4. Select *Editor* as the template (`/Sitecore Client/Content editor/Editor`).

You should now be able to see the new editor created under Ecommerce.


Creating a Product Catalog

The last part of this task is to create a product catalog. You should also reference the product source and the editor defined in the core database.

To create a product catalog:

1. Switch to the *Master* database.

2. Under `Sitecore/System/Modules/E-Commerce/Catalogs`, create a new catalog and call it *My Product Catalog*.

3. In the *My Product Catalog* item, in the **Catalog Data Source** field, enter the products source reference.
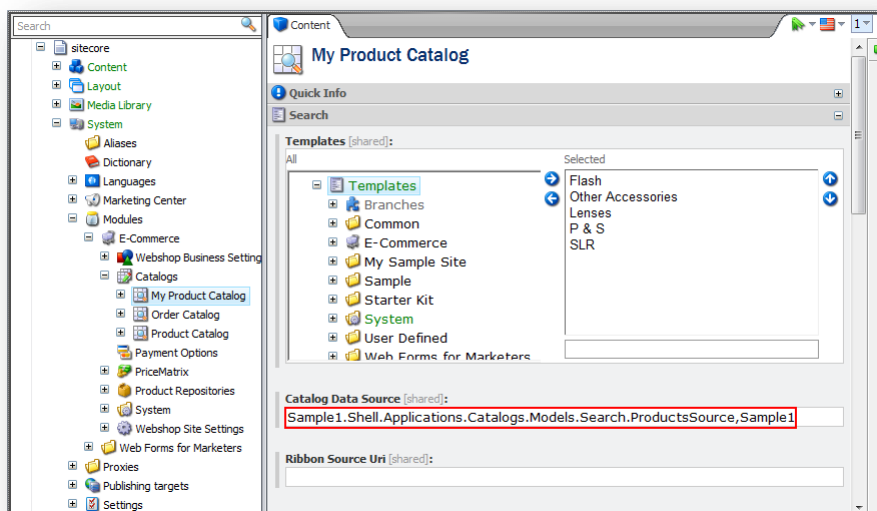
4. Browse to the standard values of the *My Product Search Group* template (`Sitecore/Templates/My Sample Site/Products categories/ My Product Search Group /_Standard Values`).

5. On the **Content** tab, in the **Editors** field, click **Edit** and select the editor you defined in the last section — *My Product Page*.
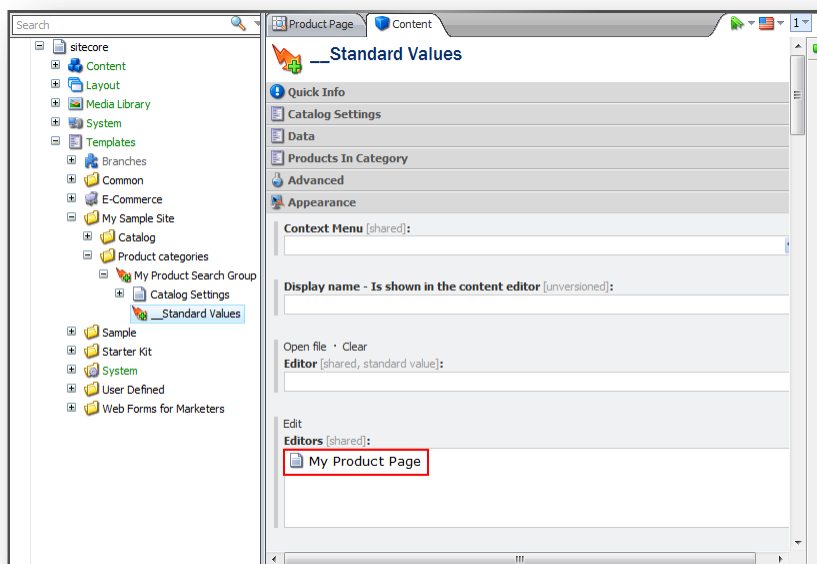
Chapter 4

SES Core Configuration

This chapter guides you through the key configuration settings in SES.

This chapter contains the following sections:

- Configuration

  This section presents the SES configuration files.

- Commands

  This section describes the <commands> element.

- Events.

  This section describes the <events> element.

- XSLExtensions

  This section describes the <xslExtension> element.

- Settings

  This section describes the <settings> element.

- Pipelines

  This section describes the <pipelines> element.

- Search

  This section describes the <search> element.

4.1 Configuration

There are two important configuration files in the Sitecore installation:

- Unity.config
- Sitecore.Ecommerce.config

This chapter focuses on the `Sitecore.Ecmmerce.config` file because it contains the configuration settings that do not exist in the content.

For information about the `Unity.config`, see the section *Unity Application Block Overview*.

Note
SES uses the `/App_Config/Include/Sitecore.Ecommerce.config` file to extend the `Unity.config` file.

The following sections describe the key configuration elements in SES.

4.2 Commands

This section describes the Ecommerce specific commands that are used in the Sitecore shell. These commands are used to define the business logic for each of the UI controls in SES.

The following snippet presents the commands that are registered in the `Sitecore.Ecommerce.config` file:

```
<commands>
    <command name="ordercatalog:changeorderstatus"
     type="Sitecore.Ecommerce.Shell.Applications.OrderCatalog.Commands.
     ChangeOrderStatus,Sitecore.Ecommerce.Shell"/>
    <command name="ordercatalog:editorder"
     type="Sitecore.Ecommerce.Shell.Applications.OrderCatalog.Commands.
     EditOrder,Sitecore.Ecommerce.Shell"/>
    <command name="ordercatalog:editorderlines"
     type="Sitecore.Ecommerce.Shell.Applications.OrderCatalog.Commands.
     EditOrderLines,Sitecore.Ecommerce.Shell"/>
</commands>
```

The following table describes the commands in the `Sitecore.Ecommerce.config`:

Command Name	Command Type	Description
Ordercatalog:changeorderstatus	Sitecore.Ecommerce.Shell.Applications.OrderCatalog.Commands.ChangeOrderStatus,Sitecore.Ecommerce.Shell	Calls the `execute` method of the `ChangeOrderStatus` class. This command changes the status of an order to one of the following: • `Authorized` • `Captured` • `New` • `Pending` • `Processing` • `Completed` • `Canceled` • `Closed` • `Held` It changes the status according to the rules defined for each state. In the following image, you can see where you can change the status of an order. On the **Order** tab, in **the Order Status** group, you select the status for the order.
Ordercatalog:editorder	Sitecore.Ecommerce.Shell.Applications.OrderCatalog.Commands.EditOrder,Sitecore.Ecommerce.Shell	Calls the `execute` method of the `EditOrder` class. This command launches the **Field Editor** dialog box where you can change the content of the order based on the fields in the order template. To edit an order, in the **Operations** group, click **Edit Order** or **Edit Header**.

Command Name	Command Type	Description
`Ordercatalog:editororderlines`	`Sitecore.Ecommerce.Shell.Applications.OrderCatalog.Commands.EditOrderLines,Sitecore.Ecommerce.Shell`	Calls the `execute` method of the `EditOrderLines` class. This command moves the focus of the Content Editor to the selected order allowing you to modify the order line that is located under the *Order* item. In the **Operations** group, click **Edit Orde**r.
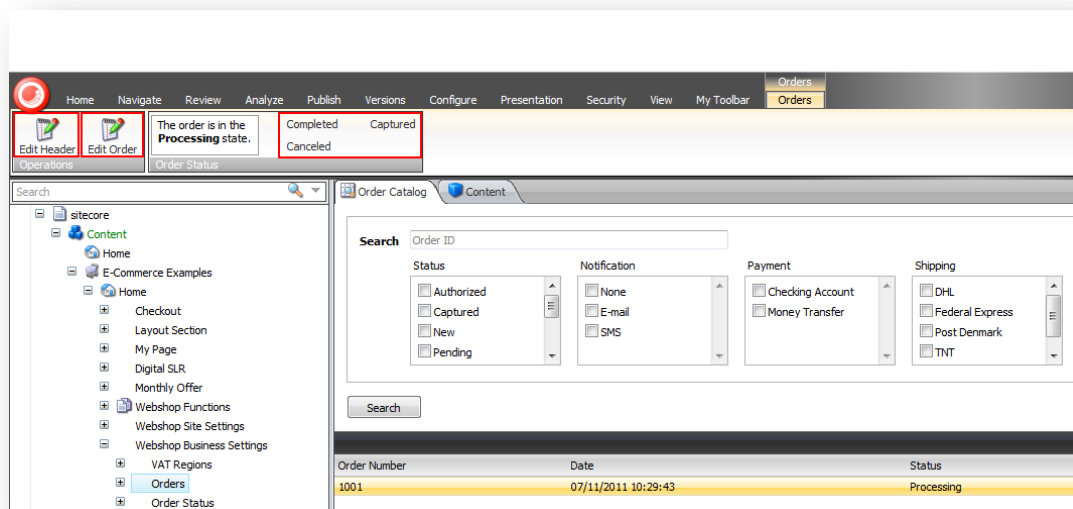
4.3 Events

You can associate your Sitecore instance to a number of events in Sitecore. You can see the list of predefined events in the `<events>` section of the `Web.config` file.

The following snippet contains the events that are registered in the `Sitecore.Ecommerce.config` file:

```xml
<events>
      <event name="item:moved">
        <handler type="Sitecore.Ecommerce.StructuredData.EnableStructuredDataModule,
         Sitecore.Ecommerce.Kernel" method="OnItemSaved" />
      </event>
      <event name="item:saved">
        <handler type="Sitecore.Ecommerce.StructuredData.EnableStructuredDataModule,
         Sitecore.Ecommerce.Kernel" method="OnItemSaved" />
        <handler type="Sitecore.Ecommerce.Unity.ClearSiteSettingsCacheEventHandler,
         Sitecore.Ecommerce.Kernel" method="OnItemSaved" />
        <handler type="Sitecore.Ecommerce.Catalogs.VirtualProductResolverCleaner,
         Sitecore.Ecommerce.Kernel" method="OnItemSaved" />
      </event>
</events>
```

The following table describes the `<events>` elements in the `Sitecore.Ecommerce.config`:

Event Name	Event Type	Description
item:moved	Sitecore.Ecommerce.StructuredData.EnableStructuredDataModule, Sitecore.Ecommerce.Kernel	Used to move an order from one location to another. It executes the OnItemSaved method that ensures that the item which is based on the order template is saved below the order repository. It creates the structured tree on the fly.
item:saved	Sitecore.Ecommerce.StructuredData.EnableStructuredDataModule, Sitecore.Ecommerce.Kernel	Used to save an order in a location. It executes the OnItemSaved method that ensures that the item which is based on the order template is saved below the order repository. It creates the structured tree on the fly.
	Sitecore.Ecommerce.Unity.ClearSiteSettingsCacheEventHandler, Sitecore.Ecommerce.Kernel	
	Sitecore.Ecommerce.Catalogs.VirtualProductResolverCleaner, Sitecore.Ecommerce.Kernel	

4.4 XSLExtensions

XSLT is a technology that can be used to output HTML from XML. XSLT can be used instead of sublayouts, whenever there is no need for complex logic. However sometimes you need to perform a little chunk of logic or execute a simple operation in your XSLT. XSL allows you to call some C# / VB methods from your XSLT.

Note
The `xslExtensions` methods could also be called directly.

The following are the XSL extensions in the SES core module.

```xml
<xslExtensions>
    <extension mode="on"
     type="Sitecore.Ecommerce.Analytics.Components.Xsl.XslExtensions,
     Sitecore.Ecommerce.Analytics"
     namespace="http://www.sitecore.net/ecommerceanalytics" singleInstance="true" />
</xslExtensions>
```

XSLT Method Name	Description
AddToShoppingCart	This method is used when a visitor adds a product to the shopping cart. It triggers the AddToShoppingCart event. Parameters: • ProductCode • ProductName • Quantity • Price
ShoppingCartEmptied	This method is used when a visitor decides to empty the shopping cart. It triggers the ShoppingCartEmptied event. Parameters: • ShoppingCartContent • ItemsinShoppingCart
ShoppingCartContinueShopping	This method is used when a visitor decides to continue shopping. It triggers the event called ShoppingCartContinueShopping.
ShoppingCartUpdated	This method is used when a visitor decides to update the shopping cart. It triggers the ShoppingCartUpdated event.
GoToShoppingCart	This method is used when a visitor decides to view the shopping cart. It triggers the GoToShoppingCart event.
ShoppingCartItemRemoved	This method is used when a visitor decides to remove an item from a specific product in the shopping cart. It triggers the ShoppingCartItemRemoved event. Parameters: • ProductCode • ProdcutName • Amount

XSLT Method Name	Description
ShoppingCartItemUpdated	This method is used when a visitor decides to update a shopping cart item. It triggers the ShoppingCartItemUpdated event. Parameters: • ProductCode • ProductName • Amount
ShoppingCartProductRemoved	This method is used when a visitor decides to remove a product from the shopping cart. It triggers the ShoppingCartProductRemoved event. Parameters: • ProductCode • ProductName • Amount
ShoppingCartViewed	This method is used when a visitor decides to view shopping cart. It triggers the ShoppingCartViewed event.
GoToCheckOut	This method is used when a visitor decides to checkout. It triggers the GoToCheckOut event.
CheckoutDeliveryNext	This method is used when the visitor clicks **Next** on the delivery page in the checkout process. It triggers the CheckoutDeliveryNext event. Parameters: • DeliveryAlternativeOption • NotificationOption • NotificationText
CheckoutDeliveryOptionSelected	This method is used when a visitor selects a checkout delivery option. It triggers the CheckoutDeliveryOptionSelected event. Parameter: • DeliveryAlternativeOption
CheckoutPaymentMethodSelected	This method is used when a visitor selects a checkout payment method. It triggers the CheckoutPaymentMethodSelected event. Parameters: • OptionTitle • OptionCode
CheckoutNext	This method is used when a visitor clicks **Next** on any page in the checkout process. It triggers the CheckoutNext event.
CheckoutPaymentNext	This method is used when a visitor clicks **Next** on the payment page in the checkout process. It triggers the CheckoutPaymentNext event.
CheckoutNotificationOptionSelected	This method is used when a visitor selects a checkout notification option. It triggers the CheckoutNotificationOptionSelected event. Parameter: • DeliveryNotificationOption

XSLT Method Name	Description
CheckoutPrevious	This method is used when a visitor clicks **Previous** during the checkout process. It triggers the CheckoutPrevious event.
AuthentificationClickedLoginButton	This method is used when a visitor clicks the login button. It triggers the AuthentificationClickedLoginButton event.
AuthentificationClickedLoginLink	This method is used when a visitor clicks the login link. It triggers the AuthentificationClickedLoginLink event.
AuthentificationUserLoggedOut	This method is used when a visitor logs out. It triggers the AuthentificationUserLoggedOut event. Parameter: • UserName
AuthentificationUserLoginSucceeded	This method is used when a visitor logs in successfully. It triggers the AuthentificationUserLoginSucceeded event. Parameter: • UserName
AuthentificationUserLoginFailed	This method is used when a visitor's login fails. It triggers the AuthentificationUserLoginFailed event. Parameter: • UserName
AuthentificationAccountCreationFailed	This method is used when a visitor's attempt to create an account fails. It triggers the AuthentificationAccountCreationFailed event.
AuthentificationAccountCreated	This method is used when a visitor creates an account. It triggers the AuthentificationAccountCreated event.
NavigationTabSelected	This method is used when a visitor clicks a navigation tab. It triggers the NavigationTabSelected event. Parameter: • TabName
NavigationProductReviewed	This method is used when a visitor chooses to review a product. It triggers the NavigationProductReviewed event. Parameters: • Code • Name • Title • Text • Rate
NavigationFollowListHit	This method is used when a visitor hits the follow list. It triggers the NavigationFollowListHit event.

XSLT Method Name	Description
Search	This method is used when a visitor searches for items on the front end. It enters a record about this search in the *Analytics* database. Parameters: • `Query` — the query used for the search. • `Hits` — the number of found items.
AddFollowListToQueryString	This method is used to return the URL concatenated with the parameters that are read from the `Ecommerce.Analytics.EventQueryStringKey` setting. Paramters: • `URL` • `ListName`
AddFollowHitToQueryString	This method is used to call the method named `AddFollowHitToQueryString` in the namespace `Sitecore.Analytics.Extensions.AnalyticsPageExtensions`. Parameters: • `URL` • `Search`
AddTriggerEventStringToQueryString	This method is used when a visitor clicks a link. It adds the trigger event — `EventName` parameter — to the query string. Parameters: • `URL` − the link that the user selects. • `EventName` − the trigger event name to be added to the query string.
GetVirtualProductUrlWithAnaliticsQueryString	This method is used when a visitor gets a virtual product's URL with an Analytics query parameter. It triggers the `GetVirtualProductUrlWithAnaliticsQueryString` event. Parameters: • `FolderNi` • `ProductNi`
GetVirtualProductUrlWithAnaliticsQueryString	This method is used when a visitor gets a virtual product's URL using an Analytics query. It triggers the `GetVirtualProductUrlWithAnaliticsQueryString` event. Parameter: • `ProsductItem`
GetItem	This method is used when a visitor user gets an item. It triggers the `GetItem` event. Prameter: • `Iterator`

4.5 Settings

This section lists the miscellaneous value pair settings in SES.

The following snippet presents these miscellaneous settings that can be configured in the `Sitecore.Ecommerce.config` file:

```xml
<settings>
    <!-- Ecommerce.Product.BaseTemplateId-->
    <setting name="Ecommerce.Product.BaseTemplateId" value="{02870C17-4273-4242-
        89A4-E973C3CF8EC0}" />
    <!-- Ecommerce.Order.OrderItemTempalteID-->
    <setting name="Ecommerce.Order.OrderItemTempalteId" value="{2769D69F-E217-4C0A-
        A41F-2083EC165218}" />
    <!-- Ecommerce.Order.OrderLineItemTempalteID-->
    <setting name="Ecommerce.Order.OrderLineItemTempalteId" value="{9A0E680B-B84E-
        42F6-9E48-68878591705B}" />
    <!-- Ecommerce.Settings.SettingsRootTemplateId-->
    <setting name="Ecommerce.Settings.SettingsRootTemplateId" value="{AC4841C3-9B0E-
        4AFD-B14B-5F280E34FBD5}" />
    <!-- Ecommerce.Analytics.EventQueryStringKey-->
    <setting name="Ecommerce.Analytics.EventQueryStringKey" value="ec_trk" />
    <!-- Ecommerce.EnableStructuredDataModule-->
    <setting name="Ecommerce.EnableStructuredDataModule" value="true" />
    <!-- Query.MaxItems specifies the max number of items in a query result set.
        If the number is 0, all items are returned. This may affect system
        performance, if a large query result is returned. This also controls the
        number of items in Lookup, Multilist and Valuelookup fields.
        Default value: 100-->
    <setting name="Query.MaxItems" value="0" />
    <!-- Orders.OpenInNewWindow specifies whether a new content editor window must
        be open when editing orders-->
    <setting name="Orders.OpenInNewWindow" value="false"/>
    <setting name="GridPageSize">
      <patch:attribute name="value">10</patch:attribute>
    </setting>
</settings>
```

The following table describes the `<Settings>` elements in the SES core.

Setting	Description
`Ecommerce.Product.BaseTemplateId`	Defines the ID of the product base template used in the domain model.
`Ecommerce.Order.OrderItemTempalteId`	Defines the ID of the order item template used in the domain model.
`Ecommerce.Order.OrderLineItemTempalteId`	Defines the ID of the order line item template used in the domain model.
`Ecommerce.Settings.SettingsRootTemplateId`	Defines the ID in Sitecore for the settings root template used in the domain model.
`Ecommerce.Analytics.EventQueryStringKey`	Defines the variable that is assigned to a string that represents a query.
`Ecommerce.EnableStructuredDataModule`	This setting is checked within the `OnItemSaved` method. If this setting is set *true*, the system puts the saved item according to the unified tree structure in Sitecore.
`Query.MaxItems`	Specifies the maximum number of items that should be shown in the results of a query. If the value is 0, all the items are returned. This may affect system performance, if a large query result is returned. This also controls the number of items in *Lookup*, *Multilist* and *Valuelookup* fields. The default value is 100.

Setting	Description
`Orders.OpenInNewWindow`	Specifies whether a new Content Editor window should open when you edit orders.
`GridPageSize`	Defines the number of rows in a user interface grid.

4.6 Pipelines

Two groups of pipelines exist in the `Sitecore.Ecommerce.config` file:

- The first group is defined within the `/configuration/sitecore/pipelines` element.

- The second group is defined within the `/configuration/sitecore/processors` element.

4.6.1 The <pipelines> Element

These are the pipelines that are grouped within the `/configuration/sitecore/pipelines` element. They define system processes.

```xml
<pipelines>
      <initialize>
        <!-- Processor initialize the Unity container configuration on the first
             start. -->
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.ConfigureEntities,
                    Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Pipelines.Loader.
                    EnsureAnonymousUsers, Sitecore.Kernel']">
         <UnityConfigSource>/App_Config/Unity.config</UnityConfigSource>
        </processor>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders,
                    Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializePaymentSystemProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeShippingSystemProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeNotificationOptionProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeCountryProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeCurrencyProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeVatRegionProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeOrderStatusProvider"/>
        <processor type="Sitecore.Ecommerce.Pipelines.Loader.
                    RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel"
                    patch:after="processor[@type='Sitecore.Ecommerce.Pipelines.Loader.
                    ConfigureEntities, Sitecore.Ecommerce.Kernel']"
                    method="InitializeBusinessCatalogProviders"/>
      </initialize>
      <preprocessRequest>
        <processor type="Sitecore.Pipelines.PreprocessRequest.FilterUrlExtensions,
                    Sitecore.Kernel">
```

```xml
                    <param desc="Allowed extensions (comma separated)">aspx, ashx,
                    asmx, svc</param>
                </processor>
        </preprocessRequest>
        <httpRequestBegin>
            <processor patch:after="*[@type='Sitecore.Pipelines.HttpRequest.ItemResolver,
                    Sitecore.Kernel']"  type="Sitecore.Ecommerce.Pipelines.HttpRequest.
                    ProductResolver, Sitecore.Ecommerce.Kernel"/>
        </httpRequestBegin>
        <getConfiguration>
            <processor type="Sitecore.Ecommerce.Pipelines.GetConfiguration.
                    GetFromContextSite, Sitecore.Ecommerce.Kernel" />
            <processor type="Sitecore.Ecommerce.Pipelines.GetConfiguration.GetFromWebSite,
                    Sitecore.Ecommerce.Kernel" />
            <processor type="Sitecore.Ecommerce.Pipelines.GetConfiguration.
                    GetFromLinkManager, Sitecore.Ecommerce.Kernel" />
            <processor type="Sitecore.Ecommerce.Pipelines.GetConfiguration.
                    GetFromResolver, Sitecore.Ecommerce.Kernel" />
        </getConfiguration>
        <startTracking>
            <processor patch:after="*[@type='Sitecore.Analytics.Pipelines.StartTracking.
                    ProcessQueryString,Sitecore.Analytics']"
                    type="Sitecore.Ecommerce.Analytics.Pipelines.StartTracking.
                    ProcessQueryString, Sitecore.Ecommerce.Analytics"/>
        </startTracking>
        <orderCreated>
            <processor type="Sitecore.Ecommerce.Pipelines.OrderCreated.SendMailToAdmin,
                    Sitecore.Ecommerce.Kernel"/>
            <processor type="Sitecore.Ecommerce.Pipelines.OrderCreated.SendMailToClient,
                    Sitecore.Ecommerce.Kernel"/>
        </orderCreated>
        <customerCreated>
            <processor type="Sitecore.Ecommerce.Pipelines.CustomerCreated.
                    ConfigureSecurity, Sitecore.Ecommerce.Kernel"/>
            <processor type="Sitecore.Ecommerce.Pipelines.CustomerCreated.LogIn,
                    Sitecore.Ecommerce.Kernel"/>
            <processor type="Sitecore.Ecommerce.Pipelines.CustomerCreated.
                    SendNotification, Sitecore.Ecommerce.Kernel"/>
        </customerCreated>
        <paymentStarted>
            <processor type="Sitecore.Ecommerce.Pipelines.PaymentStarted.StartPayment,
                    Sitecore.Ecommerce.Kernel"/>
        </paymentStarted>
        <renderLayout>
            <processor type="Sitecore.Pipelines.RenderLayout.InsertRenderings,
                    Sitecore.Kernel">
                <patch:attribute name="type">Sitecore.Ecommerce.Pipelines.RenderLayout.
                            ProcessProductPresentation, Sitecore.Ecommerce.Kernel
                </patch:attribute>
            </processor>
        </renderLayout>
        <getContentEditorFields>
            <processor type="Sitecore.Shell.Applications.ContentEditor.Pipelines.
                    GetContentEditorFields.GetFields, Sitecore.Client" >
                <patch:attribute name="type">Sitecore.Ecommerce.Shell.Applications.
                            ContentEditor.Pipelines.GetContentEditorFields.GetFields,
                            Sitecore.Ecommerce.Shell
                </patch:attribute>
                <HiddenFields>{81AD5AA7-316C-4F79-9DFF-8FEBFCFBFB4E}|{4423D09D-E95A-4827-
                            B12D-E682BE2DE834}|{39BB71D9-E6B4-4F50-BFAC-1C586724D3B9}|
                            {4200DA93-E824-4FA0-B93B-5F9AB662E3DC}
                </HiddenFields>
            </processor>
        </getContentEditorFields>
    </pipelines>
```

<initialize>

This pipeline initializes the Sitecore application.

The processor methods that start with *initialize*:

- Instantiate an instance of the provider.

- Create a name-value collection for this instance with the following attributes:
  - description
  - settings name
  - default container name
  - containers item template ID
- Register this provider instance.

Processor Method	Processor Type	Description
Process — Default method	`Sitecore.Ecommerce.Pipelines.Loader.ConfigureEntities, Sitecore.Ecommerce.Kernel`	This is the default method for this pipeline. It initializes the `Unity.config`.
`InitializePaymentSystemProvider`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the payment system provider.
`InitializeShippingSystemProvider`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the shipping system provider.
`InitializeNotificationOptionProvider`	`Sitecore.Ecommerce.Pipelines.CustomerCreated.SendNotification, Sitecore.Ecommerce.Kernel`	Initializes the notification option provider.
`InitializeCountryProvider`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the country provider.
`InitializeCurrencyProvider`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the currency provider.
`InitializeVatRegionProvider`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the VAT region provider.
`InitializeOrderStatusProvider`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the order status provider.
`InitializeBusinessCatalogProviders`	`Sitecore.Ecommerce.Pipelines.Loader.RegisterEcommerceProviders, Sitecore.Ecommerce.Kernel`	Initializes the business catalog provider.

<preprocessRequest>

This pipeline is invoked for each HTTP request managed by ASP.Net, but aborted for some requests. It is more common to use the `<httpRequestBegin>` pipeline for request processing logic, but the `preprocessRequest` pipeline is mentioned because a processor within this pipeline may prevent Sitecore from processing requests with specific extensions other than `.aspx`.

Processor Method	Processor Type	Description
Process — Default method	`Sitecore.Pipelines.PreprocessRequest.FilterUrlExtensions, Sitecore.Kernel`	This is the default pipeline that Sitecore uses to support different web pages extensions. SES uses this to present virtual products with an extension.

<httpRequestBegin>

This pipeline defines the context of Sitecore. It is invoked for each HTTP request that is not directed to ASP.NET by the `preprocessRequest` pipeline.

Processor Method	Processor Type	Description
Process – Default method	`Sitecore.Ecommerce.Pipelines.HttpRequest.ProductResolver, Sitecore.Ecommerce.Kernel`	This processor contains the implemented logic for resolving a product by its URL. See the section *SES Product Management*.

<getConfiguration>

This pipeline is executed when Sitecore initializes the basic SES components configured in Unity.

Processor Method	Processor Type	Description
Process — Default method	`Sitecore.Ecommerce.Pipelines.GetConfiguration.GetFromContextSite, Sitecore.Ecommerce.Kernel`	Uses the context item to search for the site settings.
Process — Default method	`Sitecore.Ecommerce.Pipelines.GetConfiguration.GetFromWebSite, Sitecore.Ecommerce.Kernel`	Uses the context item to search for the site settings trying to resolve a website
Process — Default method	`Sitecore.Ecommerce.Pipelines.GetConfiguration.GetFromLinkManager, Sitecore.Ecommerce.Kernel`	Uses the Link database to Search for the site settings
Process — Default method	`Sitecore.Ecommerce.Pipelines.GetConfiguration.GetFromResolver, Sitecore.Ecommerce.Kernel`	Resolves the configuration in the Unity configuration file.

<startTracking>

Processor Method	Processor Type	Description
Process — Default method	`Sitecore.Ecommerce.Analytics.Pipelines.StartTracking.ProcessQueryString, Sitecore.Ecommerce.Analytics`	This processor is used to trigger the `FollowList`, and the `FollowHit` events.

<orderCreated>

This pipeline is executed after an order has been created by the Webshop. Currently, it contains two processors that are responsible for sending out confirmation emails to the customers and the

Webshop owner.

Processor Method	Processor Type	Description
`Process` — Default method	`Sitecore.Ecommerce.Pipelines.Order Created.SendMailToAdmin, Sitecore.Ecommerce.Kernel`	Sends an e-mail to the administrator.
`Process` — Default method	`Sitecore.Ecommerce.Pipelines.Order Created.SendMailToClient, Sitecore.Ecommerce.Kernel`	Sends an e-mail to the customer.

<customerCreated>

This pipeline is executed after a visitor creates a new account on the webshop.

Processor Method	Processor Type	Description
`Process` — Default method	`Sitecore.Ecommerce.Pipelines.Custo merCreated.ConfigureSecurity, Sitecore.Ecommerce.Kernel`	Configures the visitor's security settings.
`Process` — Default method	`Sitecore.Ecommerce.Pipelines.Custo merCreated.LogIn, Sitecore.Ecommerce.Kernel`	Logs a customer in to the website.
`Process` — Default method	`Sitecore.Ecommerce.Pipelines.Custo merCreated.SendNotification, Sitecore.Ecommerce.Kernel`	Sends notification to the customer.

<paymentStarted>

This pipeline starts during the checkout process after a visitor clicks Confirm as part of the Payment step. The processor calls the selected Payment provider.

Processor Method	Processor Type	Description
`Process` — Default method	`Sitecore.Ecommerce.Pipelines.Payme ntStarted.StartPayment, Sitecore.Ecommerce.Kernel`	Invokes the capture method on the payment provider interface.

<renderLayout>

This pipeline is used by the CMS layout engine to resolve the layout, sub-layout, XSLT and web controls to render the current page based on the given URL.

Processor Method	Processor Type	Description
`Process` — Default method	`Sitecore.Pipelines.RenderLayout.In sertRenderings, Sitecore.Kernel`	Renders the layout that is defined in **Product Detail Presentation Storage** field.

<getContentEditorFields>

This pipeline defines the fields to display in the **Content Editor**.

4.6.2 The <Processors> Element

These are the pipelines that are grouped within the `/configuration/sitecore/processors` element. These pipelines operate for UI requests and interact with the user.

```
<processors>
    <uiDeleteItems>
      <processor mode="on" type="Sitecore.Ecommerce.Orders.OrderItemEventHandler,
                 Sitecore.Ecommerce.Kernel"
                 patch:before="processor[@type='Sitecore.Shell.Framework.Pipelines.
                 DeleteItems,Sitecore.Kernel' and @method='Execute']"
                 method="OnItemDeleted" />
    </uiDeleteItems>
    <saveUI>
      <processor mode="on" type="Sitecore.Ecommerce.Orders.OrderItemEventHandler,
                 Sitecore.Ecommerce.Kernel" patch:after="processor[@type=
                 'Sitecore.Pipelines.Save.Save, Sitecore.Kernel']"
                 method="OnItemSaved"/>
    </saveUI>
    <uiDuplicateItem>
       <processor mode="on" type="Sitecore.Ecommerce.Orders.OrderItemEventHandler,
                  Sitecore.Ecommerce.Kernel" patch:after="processor[@type='Sitecore.
                  Shell.Framework.Pipelines.DuplicateItem, Sitecore.Kernel'
                  and @method='Execute']" method="OnItemDuplicated"/>
    </uiDuplicateItem>
    <uiCopyItems>
      <processor mode="on" type="Sitecore.Ecommerce.Orders.OrderItemEventHandler,
                 Sitecore.Ecommerce.Kernel"
                 patch:after="processor[@type='Sitecore.Shell.Framework.
                 Pipelines.CopyItems,Sitecore.Kernel' and @method='Execute']"
                 method="OnItemCopied" />
    </uiCopyItems>
</processors>
```

The following table describes the pipelines in the `/configuration/sitecore/processors` element.

Processor	Description
`<uiDeleteItems>`	Deletes an item and its descendants.
`<saveUI>`	Saves an item.
`<uiDuplicateItem>`	Duplicates an item.
`<uiCopyItems>`	Copies an item and its descendants.

4.7 Search

This is the default configuration of the product catalog.

```
<search>
    <configuration>
      <indexes>
        <index id="products" type="Sitecore.Search.Index, Sitecore.Kernel">
          <param desc="name">$(id)</param>
          <param desc="folder">__products</param>
          <Analyzer type="Sitecore.Ecommerce.Search.LuceneAnalyzer,
                    Sitecore.Ecommerce.Kernel"/>
          <locations hint="list:AddCrawler">
            <master type="Sitecore.Ecommerce.Search.DatabaseCrawler,
                    Sitecore.Ecommerce.Kernel">
              <Database>master</Database>
              <Root>{0A702337-81CD-45B9-8A72-EC15D2BE1635}</Root>
              <Tags>master products</Tags>
            </master>
            <web type="Sitecore.Ecommerce.Search.DatabaseCrawler,
                Sitecore.Ecommerce.Kernel">
              <Database>web</Database>
              <Root>{0A702337-81CD-45B9-8A72-EC15D2BE1635}</Root>
              <Tags>web products</Tags>
            </web>
          </locations>
        </index>
      </indexes>
    </configuration>
</search>
```

Note
If you are not using the default configuration, change the Root identification to refer to your products repository. See the section *Extending the Resolve Strategy*.