Sitecore E-Commerce Fundamental Edition 1.1

Sitecore E-Commerce Developer's Cookbook

Configuration and Development with the Sitecore E-Commerce Fundamental Edition

Table of Contents

Chapter 1

Introduction

This document provides a technical overview of the Sitecore E-Commerce Fundamental Edition (SEFE), introduces SEFE configuration using the Unity application block, describes the SEFE programming contracts, and includes instructions to configure SEFE components.

You can use SEFE as a final product, or you can integrate SEFE with other applications. Sitecore designed SEFE for integration and extension. This document provides information about SEFE Application Programming Interfaces (APIs), how you can configure the .NET types used to implement various features, and the programming contracts that SEFE employs.

You can use the information in this document to understand, customize, and extend SEFE functionality. Sitecore administrators and developers should read this document before extending or customizing SEFE, such as to integrate SEFE with an external system.

You can use Sitecore to manage multiple websites. You can configure SEFE 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 — SEFE Technical Overview**
  This chapter contains a description of the domain model, the Unity application block, and SEFE's product management system.
- **Chapter 3 — The SEFE Contracts**
  This chapter describes the SEFE contracts.
- **Chapter 4 — Adding Customized Product Search Criteria**
  This chapter describes how to extend the product search feature in SEFE.

Chapter 2

SEFE Technical Overview

This chapter provides a technical overview of SEFE, including the domain model, the Unity dependency injection container, and information about how SEFE manages product information.

This chapter contains the following sections:

- The SEFE Domain Model
- Unity Application Block Overview
- SEFE Product Management

2.1 The SEFE Domain Model

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

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

SEFE includes a sample implementation that uses presentation components developed for the Web Forms for Marketers module to provide a complete online store.[1] 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.

[1] For more information about the Web Forms for Marketers module, see http://sdn.sitecore.net/Products/Web%20Forms%20for%20Marketers.aspx.

2.2 Unity Application Block Overview

SEFE 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 SEFE 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 loose coupling of application components. SEFE developers reference relatively abstract types, and Unity injects the appropriate implementations at runtime.

The Unity application block provides the following benefits for developers customizing and extending SEFE:

- **Flexibility** to specify types and dependencies through configuration and at runtime, deferring configuration to the container.
- **Simplification** of object instantiation code, especially for hierarchical structures with dependencies, which simplifies application code.
- **Abstraction** of requirements through type information and dependencies.
- **Service locator capability** supports persistence of the container, such as within the ASP.NET session or application, or through Web services or other techniques.[2]

With Unity, you can easily configure SEFE 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 SEFE and Unity, you can use different implementations of an interface or descendants of an abstract or other 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 typically involved in ecommerce into an SEFE 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.

[2] For more information about Service Locator pattern, see http://msdn.microsoft.com/en-us/library/ff649658.aspx.

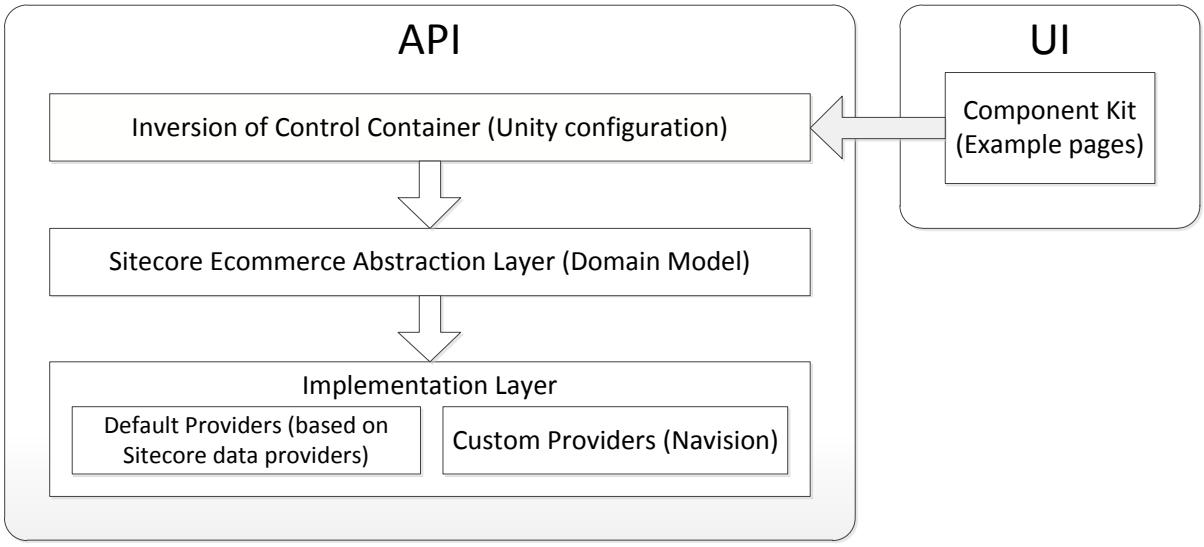
SEFE entities defined with Unity include:

- Contracts define Application Programming Interfaces (APIs).
- Implementations define concrete instances that implement contracts.
- Mappings configure which implementations to inject.
- Dependencies configure which dependant 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 SEFE 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 SEFE API layers. The example UI pages access APIs in the domain model, and SEFE uses Unity to resolve those API calls to concrete implementations of those contracts.



2.2.1 The Unity Configuration File

SEFE manages 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, which can be an interface, and 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

SEFE adds two processors to the `initialize` pipeline defined in the `web.config` file.

Note
SEFE 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 SEFE uses. This processor loads an inversion of control container into the SEFE context as a static resource in memory.

The `RegisterEcommerceProviders` processor in the `initialize` pipeline initializes various SEFE 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 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 the IOrderManager.

```
<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 Unity configuration, append a back quote character (“`”) 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.ICurrencyManager`2
```

2.2.4 How to Resolve a SEFE 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>();
```

Note
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 named `MyProductRepository`:

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

For more information about how SEFE resolves types, see the section *How to Configure SEFE for Multiple Managed Websites*.

2.2.5 How to Add an Implementation to the Unity Configuration

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

1. In the **Visual Studio** project, create a class that implement the required interface or inherits from the appropriate base class.
2. In 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 instructions to configure SEFE to use the implementation, see the sections *How to Replace a SEFE Component* and *How to Configure SEFE for Multiple Managed Websites*.

2.2.6 How to Add a Contract to the Unity Configuration

To add a contract to 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 instructions 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 SEFE Component

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

1. In the Unity configuration, add a `/unity/alias` element to register the new implementation. For instructions 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 SEFE for Multiple Managed Websites

Different managed websites can use different implementations of each SEFE contract. When SEFE resolves dependencies, it first checks for the existence of types named after the context site.

To use different implementations of contracts for different managed websites:

1. Add any required implementations to the Unity configuration. For instructions to add an implementation to Unity configuration, see the section *How to Add an Implementation to the Unity Configuration*.
2. For each implementation, create a `/unity/container/register` element in the Unity configuration.

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

3. Set the value of the `name` attribute in the new `/unity/container/register` element to the name of the managed website.

For example, to cause SEFE to use the ProductCategory implementation with alias `MyProductCategory` for the managed websites named `site2` and `site3`, and use the default

ProductCategory implementation with alias `SitecoreProductCategory` for all other managed websites, configure the following `/unity/container/register` elements in the Unity configuration:

```
...
<!-- 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>
```

To access a named implementation, pass the name of the implementation without the site name to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method. For example, given the following Unity configuration:

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

If the name of the context site is `site3`, then the following code accesses the ProductCategory implementation named `site3MyOtherProductCategory`:

```
Sitecore.Ecommerce.DomainModel.Products.ProductCategory productCategory =
  Sitecore.Ecommerce.Context.Entity.Resolve
    <Sitecore.Ecommerce.DomainModel.Products.ProductCategory>("MyOtherProductCategory");
```

If you do not pass a parameter to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method, if an implementation exists with the same name as the context site, Unity injects that type. Otherwise, Unity injects the default implementation of the contract. If no default implementation exists, Unity raises an error.

If you pass a parameter to the `Sitecore.Ecommerce.Context.Entity.Resolve()` method, if an implementation exists with a name that matches the name of the context site concatenated with the value of the parameter, Unity injects that type. Otherwise, Unity injects the default implementation for the contract. If no default implementation exists, Unity raises an error.

2.3 SEFE Product Management

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

SEFE 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 item associated with such a URL.

2.3.1 Product URLs and Product Resolution

SEFE 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. For more information about the VirtualProductResolver, see the section *The SEFE ProductUrlProcessor Contract*.

How to Specify the Product URL Format

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

- In the **Content Editor**, in the home item for the managed Web site or the root item of the branch, in the **System** section, 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.

SEFE 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, SEFE applies that value, otherwise SEFE applies the value of the **Display Products Mode** field in the home item.

2.3.2 Product Presentation

The URLs of SEFE product pages map to items that do not define layout details.[3]

Important
Do not update 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** user interface to navigate from a page that links to the product to the product detail page.

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

SEFE 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 layout details from the item specified in the **Product Detail Presentation Storage** field in the nearest ancestor of the logical parent item of the virtual product item that defines a value for that field. For example, given the URL `/products/product_name.aspx`, if the `<home>/products` item has a value for **Product Detail Presentation Storage** field, SEFE applies that value, otherwise SEFE applies the value of the **Product Detail Presentation Storage** field in 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 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 layout details.[4]

Note
You can use access rights to control which users can apply various product presentation formats. You can change the type of the **Product Detail Presentation Storage** in the `Ecommerce/Product Categories/Product Search Group` from **Lookup** to **Droptree**, create folders under `/Sitecore/System/Modules/Ecommerce/System/Product Presentation Repository` to contain different groups of presentation format definition items, and 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 instructions to update the product presentation format, see the section *How to Update a Product Presentation Format*.

[4] For instructions to apply layout details, see the *Presentation Component Cookbook* at http://sdn.sitecore.net/Reference/Sitecore%206/Presentation%20Component%20Cookbook.aspx.

4. Optionally, apply the new product presentation format to existing pages. For instructions to apply a product presentation format, see the section *How to Specify a Product Presentation Format*.

Chapter 3

The SEFE Contracts

This chapter describes the SEFE contracts used to abstract various SEFE implementations.

SEFE uses Unity to configure a number of contracts that generally exist in assemblies that match their namespaces, such as the `Sitecore.Ecommerce.DomainModel.dll` assembly, the `Sitecore.Ecommerce.Kernel.dll` assembly, and the `Sitecore.Analytics.dll` assembly, all in the `/bin` directory.

This chapter contains the following sections:

- SEFE Configuration Contracts
- SEFE Business Object Contracts
- SEFE Business Logic Contracts
- SEFE Payment Providers
- SEFE Data Contracts
- The SEFE ISearchProvider Contract
- The SEFE ICatalogProductResolveStrategy Contract
- The SEFE AnalyticsHelper Contract
- The SEFE ProductUrlProcessor Contract

3.1 SEFE Configuration Contracts

The SEFE configuration contracts allow multiple websites to share common data and configuration for some purposes, such as a product repository, while using separate data for other purposes, such as order management. SEFE defines the following contracts that expose configuration information.

Note
Do not extend or replace the configuration contracts or implementations. Instead, create your own contract and implementation to manage configuration settings.

3.1.1 The SEFE BusinessCatalogSettings Contract

The default data managers use the BusinessCatalogSettings contract (`Sitecore.Ecommerce.DomainModel.Configurations.BusinessCatalogSettings`) to determine the root items for various SEFE business information stores, such as the product and order stores.

The default implementation (`Sitecore.Ecommerce.Configurations.BusinessCatalogSettings`) of the BusinessCatalogSettings contract retrieves field values from the child named **Business Catalog** of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

3.1.2 The SEFE DesignSettings Contract

The DesignSettings contract (`Sitecore.Ecommerce.DomainModel.Configurations.DesignSettings`) exposes layout and presentation configuration settings for presentation components on the managed website(s).

The default implementation (`Sitecore.Ecommerce.Configurations.DesignSettings`) of the DesignSettings contract retrieves field values from the child named **Design Settings** of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Design Settings`).

3.1.3 The SEFE GeneralSettings Contract

The GeneralSettings contract (`Sitecore.Ecommerce.DomainModel.Configurations.GeneralSettings`) exposes global configuration settings.

The default implementation (`Sitecore.Ecommerce.Configurations.GeneralSettings`) of the GeneralSettings contract retrieves field values from the **General** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/General`).

3.1.4 The SEFE ShoppingCartSettings Contract

The ShoppingCartSettings contract (`Sitecore.Ecommerce.DomainModel.Configurations.ShoppingCartSettings`) exposes configuration settings for individual shopping carts.

The default implementation (`Sitecore.Ecommerce.Configurations.ShoppingCartSettings`) of the ShoppingCartSettings contract manages information in the **Shopping Cart** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Shopping Cart`).

3.1.5 The SEFE ShoppingCartSpotSettings Contract

The ShoppingCartSpotSettings contract (`Sitecore.Ecommerce.DomainModel.Configurations.ShoppingCartSpotSettings`) exposes configuration settings for presentation components that display an individual shopping cart.

The default implementation (`Sitecore.Ecommerce.Configurations.ShoppingCartSpotSettings`) of the ShoppingCartSpotSettings contract accesses the **Shopping Cart Spot** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Shopping Cart Spot`).

3.2 SEFE Business Object Contracts

SEFE defines the following contracts that represent business objects.

3.2.1 The SEFE AddressInfo Contract

The AddressInfo contract (`Sitecore.Ecommerce.DomainModel.Addresses.AddressInfo`) exposes information about a physical address.

The default implementation (`Sitecore.Ecommerce.Addresses.AddressInfo`) of the AddressInfo contract represents typical address information.

3.2.2 The SEFE Country Contract

The Country contract (`Sitecore.Ecommerce.DomainModel.Addresses.Country`) exposes information about a country.

The default implementation (`Sitecore.Ecommerce.Addresses.Country`) of the Country contract represents the children of the item specified by the **Countries Link** field in the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

3.2.3 The SEFE Currency Contract

The Currency contract (`Sitecore.Ecommerce.DomainModel.Currencies.Currency`) exposes information about a currency.

The default implementation (`Sitecore.Ecommerce.Currencies.Currency`) of the Currency contract represents the children of the item specified by the **Currencies Link** field in the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

3.2.4 The SEFE CustomerInfo Contract

The CustomerInfo contract (`Sitecore.Ecommerce.DomainModel.Users.CustomerInfo`) exposes information about a customer.

The default implementation (`Sitecore.Ecommerce.Users.CustomerInfo`) of the CustomerInfo contract provides basic customer information.

3.2.5 The SEFE IProductRepositoryItem Contract

The IProductRepostiryItem contract (`Sitecore.Ecommerce.DomainModel.Products.IProductRepositoryItem`) represents any item in a product repository, such as a product or product category. All items in a product repository implmement the IProductRepositoryItem contract. For more information about product repositories, see the section The SEFE IProductRepository Contract. For more information about products, see the section The SEFE ProductBaseData Contract. For more information about product categories, see the section The SEFE ProductCategory Contract.

The default implementations of the IProductRepositoryItem contract include the ProductBaseData contract and the ProductCategory contract.

3.2.6 The SEFE NotificationOption Contract

The NotificationOption contract (`Sitecore.Ecommerce.DomainModel.Shippings.NotificationOption`) exposes information about how a customer prefers to receive notification about the status of an order.

The default implementation (`Sitecore.Ecommerce.Shippings.NotificationOption`) of the NotificationOption contract causes customers to receive an e-mail about each order.

3.2.7 The SEFE Order Contract

The Order contract (`Sitecore.Ecommerce.DomainModel.Orders.Order`) exposes information about individual orders.

The default implementation (`Sitecore.Ecommerce.Orders.Order`) of the Order contract represents the descendants of the item specified by the **Orders Link** field in the **Business Catalog** child in the **System Links** section of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

To integrate an external order management system, you do not need to implement the Order contract. Instead, implement the IOrderManager contract to manage Orders. For more information about the IOrderManager contract, see the section *The SEFE IOrderManager Contract*.

How to Implement the Order Contract

To implement the Order contract:

1. In the **Visual Studio** project, create a class that implements the Order contract (`Sitecore.Ecommerce.DomainModel.Orders.Order`) to abstract information about an order.
2. In the new class, implement a constructor that accepts an object that implements the OrderStatus contract. For more information about the OrderStatus contract, see the section *The SEFE OrderStatus Contract*.
3. Optionally, implement the OrderLine contract. For more information about the OrderLine contract, see the section *The SEFE OrderLine Contract*.
4. Update Unity configuration to use your implementation of the Order contract. For instructions to update Unity configuration, see the section *How to Replace a SEFE Component*. For example:

```
<alias alias="MyOrder" type="MyNamespace.MyOrder, MyAssembly" />
...
<register type="Order" mapTo="MyOrder">
...
```

3.2.8 The SEFE OrderLine Contract

The OrderLine contract (`Sitecore.Ecommerce.DomainModel.Orders.OrderLine`) exposes information about a line item on an order.

The default implementation (`Sitecore.Ecommerce.Orders.OrderLine`) of the OrderLine contract represents the descendants of an order item as described in the section *The SEFE Order Contract*.

3.2.9 The SEFE OrderStatus Contract

The OrderStatus contract (`Sitecore.Ecommerce.DomainModel.Orders.OrderStatus`) represents the status of an order. Each implementation of the OrderStatus contract can contain logic to apply when the system updates the status of an order to that OrderStatus implementation.

The default OrderStatus implementations include:

- Completed (`Sitecore.Ecommerce.Orders.Statuses.Completed`)
- Closed (`Sitecore.Ecommerce.Orders.Statuses.Closed`)
- Held (`Sitecore.Ecommerce.Orders.Statuses.Held`)
- Pending (`Sitecore.Ecommerce.Orders.Statuses.Pending`)
- Processing (`Sitecore.Ecommerce.Orders.Statuses.Processing`)
- Canceled (`Sitecore.Ecommerce.Orders.Statuses.Canceled`)
- New (`Sitecore.Ecommerce.Orders.Statuses.New`)

How to Override an OrderStatus Implementation

To override the logic that SEFE applies when an order reaches an existing order status:

1. In the **Visual Studio** project, create a class that inherits from the `Sitecore.Ecommerce.Orders.Statuses.OrderStatusBase` class, or from the class that provides the default implementation of the order status.
2. In the new class, implement the `Process()` method, which may call the `Process()` method in the base class.
3. In Unity configuration, create a new `/unity/alias` element to register the new implementation. For instructions to add an implementation to Unity configuration, see the section *How to Add an Implementation to the Unity Configuration*.
4. In Unity configuration, update the `/unity/container/register` element for the order status to use your implementation. For instructions to update Unity configuration, see the section *How to Replace a SEFE Component*.

How to Implement a New Order Status

To implement a new order status:

1. In the **Visual Studio** project, create a class that inherits from the `Sitecore.Ecommerce.Orders.Statuses.OrderStatusBase` class.
2. In the new class, implement the `Process()` method to contain logic for SEFE to apply when placing the order into that status.
3. In Unity configuration, add a `/unity/alias` element to register the new implementation For instructions to add an implementation to Unity, see the section *How to Add an Implementation to the Unity Configuration*. For example:

```
<alias alias="ShippedOrderStatus" type="MyNamespace.ShippedOrderStatus, MyAssembly" />
```

4. In Unity configuration, add a `/unity/container/register` element to define a mapping for the new implementation. Set the `type` attribute of the new `/unity/container/register` element to OrderStatus. Set the `mapTo` attribute of the new `/unity/container/register`

element to the `alias` attribute of the new `/unity/alias` element. Set the `name` attribute of the `/unity/container/register` element to identify the status. For example:

```
<register type="OrderStatus" mapTo="ShippedOrderStatus" name="Shipped">
  <interceptor type="VirtualMethodInterceptor" />
  <policyInjection />
</register>
```

5. In the **Content Editor**, select the item specified in the field named **Order Statuses Link** in the **System Links** section of the child named **Business Catalog** of the **Site Settings** child of the home item of the managed website (`<home>/Site Settings/Business Catalog`).
6. In the **Content Editor**, insert an order status definition item using the `Ecommerce/Business Catalog/Order Status` data template.
7. In the new order status definition item, in the **Data** section, in the **Code** field, enter the `name` attribute of the new `/unity/container/register` element in Unity configuration.
8. In the new order status definition item, in the **Data** section, in the **Title** field, enter the label that should appear in the user interface to transition an order to this status. Enter the same value for the **Name** field in the **Data** section.
9. In the new order status definition item, in the **Data** section, in the **Available List** field, select the order status(es) that the user can apply to an order currently associated with this order status.

How to Resolve an Order Status

You can use the `Sitecore.Ecommerce.Entity.Resolve()` method to resolve an order status. For example, to assign `Shipped` order status to an order:

```
using Sitecore.Ecommerce.DomainModel.Orders;
...
IOrderManager<Order> orderManager = Sitecore.Ecommerce.Context.Entity.Resolve
  <IOrderManager<Order>>();
Order order = orderManager.GetOrder("order number");
order.Status = Sitecore.Ecommerce.Context.Entity.Resolve<OrderStatus>("Shipped");
orderManager.SaveOrder(order);
```

3.2.10 The SEFE PaymentSystem Contract

The PaymentSystem contract (`Sitecore.Ecommerce.DomainModel.Payments.PaymentSystem`) exposes information about an online payment provider gateway. For more information about payment providers, see the section *SEFE Payment Providers*.

The default implementation (`Sitecore.Ecommerce.Payments.PaymentSystem`) of the PaymentSystem contract represents a child of the item specified by the **Payment Systems Link** field in the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

3.2.11 The SEFE ProductBaseData Contract

The ProductBaseData contract (`Sitecore.Ecommerce.DomainModel.Products.ProductBaseData`) represents basic information about a product.

The default implementation (`Sitecore.Ecommerce.Products.Product`) of the ProductBaseData contract represents common information about a product, such as product name and product description.

3.2.12 The SEFE ProductCategory Contract

The ProductCategory contract (`Sitecore.Ecommerce.DomainModel.Products.ProductCategory`) represents a category of products.

The default implementation (`Sitecore.Ecommerce.Products.ProductCategory`) of the ProductCategory contract represents basic information about a product category, such as product category name and product category code.

3.2.13 The SEFE ProductLine Contract

The ProductLine contract (`Sitecore.Ecommerce.DomainModel.Products.ProductLine`) represents information about a specific product in a business entity, such as the quantity of a product in a shopping cart or order.

The default implementations of the ProductLine contract include the OrderLine contract and the ShoppingCartLine contract. For more information about the OrderLine contract, see the section *The SEFE OrderLine Contract*. For more information about the ShoppingCartLine contract, see the section *The SEFE ShoppingCartLine*.

3.2.14 The SEFE ShippingProvider Contract

The ShippingProvider contract (`Sitecore.Ecommerce.DomainModel.Shippings.ShippingProvider`) exposes information about a shipping system.

The default implementation (`Sitecore.Ecommerce.Shippings.ShippingProvider`) of the ShippingProvider contract represents the children of the item specified by the **Shipping Providers Link** field in the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

3.2.15 The SEFE ShoppingCart Contract

The ShoppingCart contract (`Sitecore.Ecommerce.DomainModel.Carts.ShoppingCart`) exposes information about the state of an individual shopping cart, such as its contents.

The default implementation (`Sitecore.Ecommerce.Carts.ShoppingCart`) of the ShoppingCart contract implements typical shopping cart functionality.

3.2.16 The SEFE ShoppingCartLine Contract

The ShoppingCartLine contact (`Sitecore.Ecommerce.DomainModel.Carts.ShoppingCartLine`) exposes information about an item in a shopping cart.

The default implementation (`Sitecore.Ecommerce.Carts.ShoppingCartLine`) of the ShoppingCartLine contract implements typical shopping cart line functionality.

3.2.17 The SEFE Totals Contract

The Totals contract (`Sitecore.Ecommerce.DomainModel.Prices.Totals`) exposes information about pricing totals for an order.

The default implementation (`Sitecore.Ecommerce.Prices.Totals`) of the Totals contract stores data in session during transactions and persists that data in order items as described in the previous section, *The SEFE Order Contract*.

3.2.18 The SEFE VatRegion Contract

The VatRegion contract (`Sitecore.Ecommerce.DomainModel.Addresses.VatRegion`) exposes information about a tax region.

The default implementation (`Sitecore.Ecommerce.Addresses.VatRegion`) of the VatRegion contract exposes the descendants of the **VAT Regions** child of the **Business Catalog** child of the home item for the context site (`<home>/Business Catalog/VAT Regions`).

3.3 SEFE Business Logic Contracts

SEFE defines the following contracts that abstract business logic.

3.3.1 The SEFE ICheckOut Contract

The ICheckOut contract (`Sitecore.Ecommerce.DomainModel.CheckOuts.ICheckOut`) defines a programming interface to determine or alter the state of the shopping checkout process.

Pages in the checkout process access properties and methods of the default implementation of the ICheckOut contract.

3.3.2 The SEFE ICurrencyManager Contract

The ICurrencyManager contract (`Sitecore.Ecommerce.DomainModel.Currencies.ICurrencyManager`) defines a programming interface for currency conversion.

The default implementation (`Sitecore.Ecommerce.Currencies.CurrencyManager`) of the ICurrencyManager contract uses information in the descendants of the item specified by the **Currency Matrix Link** field in the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

3.3.3 The SEFE ICustomerManager Contract

The ICustomerManager contract (`Sitecore.Ecommerce.DomainModel.Users.CustomerManager`) defines a programming interface for managing information about customers.

The default implementation (`Sitecore.Ecommerce.Users.CustomerManager`) of the ICustomerManager contract manages customer information in the Sitecore ASP.NET membership database.

3.3.4 The SEFE IMail Contract

The IMail contract (`Sitecore.Ecommerce.DomainModel.Mails.IMail`) defines a programming interface for sending e-mail.

The default implementation (`Sitecore.Ecommerce.Mails.Mail`) of the IMail contract uses the `MailServer`, `MailServerUserName`, `MailServerPassword`, and `MailServerPort` settings in the `web.config` file.

3.3.5 The SEFE IOrderManager Contract

The IOrderManager contract (`Sitecore.Ecommerce.DomainModel.Orders.IOrderManager`) defines a programming interface for managing information about orders.

The default implementation (`Sitecore.Ecommerce.Orders.OrderManager`) of the IOrderManager contract accesses the descendants of the item specified by the **Orders Link** field in the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).

Note
The default implementation of the IOrderManager contract writes order information to the Sitecore Master database.

How to Integrate an Order Management System

To integrate an external order management system:

1. Optionally, implement the Order contract. For more information about the Order contract, see the section *The SEFE Order Contract*.
2. In the **Visual Studio** project, create a class that implements the IOrderManager contract to abstract the order management system.
3. In the new class, implement the `GetOrder()` method to retrieve information about an order from the external order management system, and return an object that implements the Order contract to contain that information.
4. In the new class, implement the `GetOrders()` method to retrieve orders matching a given query from the external order management system.
5. In the new class, implement the `CreateOrder()` method to create an order in the external order management system.
6. In the new class, implement the `SaveOrder()` method to update an order in the external order management system.
7. In the new class, implement the `GenerateOrderNumber()` method to generate an order number appropriate for the external order management system.
8. In Unity configuration, add an `/alias/alias` element for your IOrderManager implementation. For instructions to add an implementation to Unity configuration, see the section *How to Add an Implementation to the Unity* Configuration.
9. Configure SEFE to use the IOrderManager implementation. Update the `mapTo` attribute of the `/unity/container/register` element named `IOrderManager` to the value of the alias attribute of the new `/unity/alias` element that specifies your IOrderManager implementation. For instructions to configure SEFE to use your implementation, see the section *How to Replace a SEFE Component*.

For additional information about Unity configuration, including instructions to use different implementations under different conditions, see the section *Dependency Injection*.

Note
If you integrate SEFE with an external order management system, Sitecore recommends that you also write orders data to Sitecore, so that the website can continue to process orders even when the external order management system is unavailable.

3.3.6 The SEFE IProductRepository Contract

The IProductRepository contract (`Sitecore.Ecommerce.DomainModel.Products.IProductRepository`) defines a programming interface for managing a product catalog.

The default implementation (`Sitecore.Ecommerce.Products.ProductRepository`) of the IProductRepository contract manages the descendants of the item specified by the **Products Link** field in

the **System Links** section of the **Business Catalog** child of the **Site Settings** child of the home item of the context site (`<home>/Site Settings/Business Catalog`).[5]

If you implement the IProductRepository contract, you should also implement the IProductPriceManager contract. For more information about the IProductPriceManager contract, see the section *The SEFE IProductPriceManager Contract*.

3.3.7 The SEFE IProductPriceManager Contract

The IProductPriceManager contract (`Sitecore.Ecommerce.DomainModel.Prices.IProductPriceManager`) defines a programming interface for product pricing.

The default implementation (`Sitecore.Ecommerce.Prices.ProductPriceManager`) of the IProductPriceManager contract applies pricing information stored in the **Price** field in the **Product Meta Info** section of the product definition item, accounting for member and non-member pricing types, plus the VAT percentage associated with the region of purchase. For more information about VAT, see the section *The SEFE VatRegion Contract*.

How to Add a Price Type to the Default IProductPriceManager Implementation

To add a price type to the default IProductPriceManager implementation:

1. In the **Content Editor**, select the `/Sitecore/System/Modules/Ecommerce/PriceMatrix/Shop` item.
2. In the **Content Editor**, insert a new price type definition item using the `Ecommerce/Price Field/PriceMatrixPrice` data template.
3. In the new price type definition item, in the **Data** section, in the **Title** field, enter the label for the new price type.
4. In the **Content Editor**, sort the price type definition items to control their order of appearance in the **Price** field of product definition items.
5. In the **Content Editor**, edit product definition items. In the **Product Meta** Info section, in the **Price** field, enter values for the new price type.
6. Update rendering components to apply the new price type as appropriate.

To access the new price type for a product, pass the value of the **Title** field in the product price type definition item as the second parameter to the `GetPriceMatrixPrice()` method of the IProductPriceManager contract.

3.3.8 The SEFE IShoppingCartManager Contract

The IShoppingCartManager contract (`Sitecore.Ecommerce.DomainModel.Carts.IShoppingCartManager`) defines a programming interface for managing information about an individual shopping cart.

The default implementation (`Sitecore.Ecommerce.Carts.ShoppingCartManager`) stores information in ASP.NET session.

[5] For information about APIs for importing product data into Sitecore, see the *Content API Cookbook* at http://sdn.sitecore.net/Reference/Sitecore%206/Content%20API%20Cookbook.aspx.

3.3.9 The SEFE ITransactionData Contract

The ITransactionData contract (`Sitecore.Ecommerce.DomainModel.Payments.ITransactionData`) defines a programming interface to persist payment transaction information between HTTP requests.

The default implementation (`Sitecore.Ecommerce.Payments.TransactionData`) of the ITransactionData contract stores data in ASP.NET session.

3.4 SEFE Payment Providers

SEFE payment providers represent individual payment systems.[6] SEFE supports two types of payment providers: online and offline.

3.4.1 SEFE Online Payment Providers

Online payment providers implement the IOnlinePayementProvider contract (`Sitecore.Ecommerce.DomainModel.Payments.IOnlinePaymentProvider`) and represent online payment gateways.

The default online payment provider implementations include:

- Amazon (`Sitecore.Ecommerce.Payments.Amazon.AmazonPaymentProvider`).
- Authorize.NET (`Sitecore.Ecommerce.Payments.AuthorizeNet.AuthorizeNetPaymentProvider`).
- BBS (`Sitecore.Ecommerce.Payments.BBS.BBSPaymentProvider`).
- DIBS (`Sitecore.Ecommerce.Payments.DIBS.DIBSPaymentProvider`).
- ePay (`Sitecore.Ecommerce.Payments.EPay.EPayPaymentProvider`).
- PayEx (`Sitecore.Ecommerce.Payments.PayEx.PayExPaymentProvider`).
- QuickPay (`Sitecore.Ecommerce.Payments.QuickPay.QuickPayPaymentProvider`).

3.4.2 SEFE Offline Payment Providers

Offline payment providers represent offline payment methods, such as a paper check or money order. Offline payment providers implement the IOfflinePaymentProvider contract (`Sitecore.Ecommerce.DomainModel.Payments.IOfflinePaymentProvider`).

The default implementation (`Sitecore.Ecommerce.Payments.OfflinePaymentProvider`) of the IOfflinePaymentProvider contract creates an order without processing payment.

[6] For more information about payment providers, see the *Payment Method Reference Guide* at http://sdn.sitecore.net/Products/ECommerce/Documentation.aspx.

3.5 SEFE Data Contracts

SEFE provides the following contracts for managing representations of data.

3.5.1 The SEFE IDataMapper Contract

The IDataMapper contract (`Sitecore.Ecommerce.Data.IDataMapper`) defines a programming interface to help various data managers abstract storage.

The default implementation (`Sitecore.Ecommerce.Data.DataMapper`) of the IDataMapper contract represents data as Sitecore items.

The default IDataMapper implementation uses the `Entity` attribute in .NET to determine the data templates and fields associated with various data elements. For example, in the following example, the `Entity` attributes in square brackets ("`[]`") define the ID of a data template for products and the name of a field in that data template that contains the specified property:

```
[Entity(TemplateId = '{B87EFAE7-D3D5-4E07-A6FC-012AAA13A6CF}')]
public class Product : DomainModel.Products.ProductBaseDate, IEntity
{
  [Entity(FieldName = '__Display name')]
  public override string Name { get ; [NotNullValue] set; }
...
```

3.5.2 The SEFE EntityHelper Contract

The EntityHelper contract (`Sitecore.Ecommerce.Data.EntityHelper`) providers an API that the default implementation of the IDataMapper contract uses to access the value of the `Entity` attributes in .NET code. The class that defines the EntityHelper contract also serves as the default implementation of the contract. For more information about the IDataMapper contract, see the section *The SEFE IDataMapper Contract*.

3.5.3 The SEFE IEntityProvider Contract

The IEntityProvider contract (`Sitecore.Ecommerce.DomainModel.Data.IEntityProvider`) provides an abstract API to access a variety of similar data types.

The default implementation (`Sitecore.Ecommerce.Data.EntityProvider`) of the IEntityProvider contract manages data in items based on the `Ecommerce/Business Catalog/Option Value` data template or any data template that inherits from that data template.

You can use the IEntityProvider contract to access information about countries, country states, currencies, delivery alternatives, language option values, notification options, payments, and VAT option values. For example, to access information about all countries:

```
using Sitecore.Ecommerce.DomainModel.Data;
using Sitecore.Ecommerce.DomainModel.Addresses;
...
IEntityProvider<Country> countries =
  Sitecore.Ecommerce.Context.Entity.Resolve<IEntityProvider<Country>>();

foreach(Country country in countries.GetAllEntities())
{
...
}
```

To access a specific country by country code:

```
Country unitedStates = countries.GetEntityByCode("US");
```

You can create data templates based on the `Ecommerce/Business Catalog/Option Value` data template and use the IEntityProvider contract to access the stored data.

3.5.4 The SEFE IMappingRule Contract

The IMappingRule contract (`Sitecore.Ecommerce.Data.IMappingRule`) defines a programming interface to define adapters for mapping between physical and logical storage for complex types, including conversion between system and Sitecore internal data types such as dates in the ISO string format used by Sitecore.

Sitecore provides two default implementations of the IMappingRule contract:

- The Order mapping rule (`Sitecore.Ecommerce.Data.OrderMappingRule`) implementation of the IMappingRule contract adapts orders from items in the content tree.
- The OrderLine mapping rule (`Sitecore.Ecommerce.Data.OrderLineMappingRule`) implementation of the IMappingRule contract adapts order lines from items in the content tree.

The default implementation of the IDataMapper contract uses these implementations of the IMappingRule contract.

3.6 The SEFE ISearchProvider Contract

The ISearchProvider contract (`Sitecore.Ecommerce.Search.ISearchProvider`) defines a programming interface for locating items matching specific criteria.

SEFE provides three implementations of the ISearchProvider contract:

- The Lucene search provider (`Sitecore.Ecommerce.Search.LuceneSearchProvider`) uses a Lucene search index.
- The Sitecore Query search provider (`Sitecore.Ecommerce.Search.SitecoreQuerySearchProvider`) uses Sitecore query.
- The Fast Query search provider (`Sitecore.Ecommerce.Search.FastQuerySearchProvider`) uses Sitecore fast query.

3.7 The SEFE ICatalogProductResolveStrategy Contract

The ICatalogProductResolveStrategy contract (`Sitecore.Ecommerce.DomainModel.Catalogs.ICatalogProductResolveStrategy`) defines an API to retrieve specified products from a product catalog.

Sitecore provides two default implementations of the ICatalogProductResolveStrategy contract:

- The Product List product resolution strategy (`Sitecore.Ecommerce.Catalogs.ProductListCatalogResolveStrategy`) retrieves one or more items based on their IDs.
- The Query product resolution strategy (`Sitecore.Ecommerce.Catalogs.QueryCatalogProductResolveStrategy`) returns products that match search query.

When a user creates an item to present some number of products on a website, they select one of the ICatalogProductResolveStrategy implementations to determine how to specify the products to display. SEFE stores the user's selections as parameters in fields of the item, and presentation components use those fields to determine which products to display.

The Product Page custom editor that appears for items based on the `Ecommerce/Product Categories/Product Search Group` data template uses these two ICatalogProductResolveStrategy implementations. SEFE manages ICatalogProductResolveStrategy definition items beneath the `/Sitecore/System/Modules/Ecommerce/System/Product Selection Method` item.

The `Sitecore.Ecommerce.Xsl.XslExtensions.GetProductsForCatalog()` XSL extension method (intended for use with items based on the `Ecommerce/Product Categories/Product Search Group` data template) returns the list of products retrieved using the strategy selected in the context item. To expose this method as `sc:GetProductsForCatalog()` in an XSL rendering, add the following attribute to the `/xsl:stylesheet` element in the `.xslt` file:

```
        xmlns:ec="http://www.sitecore.net/ec"
```

3.8 The SEFE AnalyticsHelper Contract

The AnalyticsHelper contract supports integration between the Sitecore Online Marketing Suite (OMS) and SEFE.[7] For example, the AnalyticsHelper contract exposes an API to identify specific OMS events as goals.[8]

The class that defines the AnalyticsHelper contract also serves as the default implementation of the AnalyticsHelper contract.

[7] For more information about the Sitecore Online Marketing Suite (OMS), see http://www.sitecore.net/en/Products/Sitecore-Online-Marketing-Suite.aspx.

[8] For information about APIs to access SEFE events, see the classes in the `Sitecore.Ecommerce.Analytics.Components.PageEvents` namespace and the `Sitecore.Ecommerce.Analytics.AnalyticsHelper` class.

3.9 The SEFE ProductUrlProcessor Contract

The ProductUrlProcessor contract (`Sitecore.Ecommerce.Catalogs.ProductUrlProcessor`) defines a programming interface that determines the URL of a product item and another that determines the product specified by a URL. Product resolvers control how SEFE constructs and parses the URLs of product pages.

SEFE provides multiple implementations of the ProductUrlProcessor contract:

- The NameProductUrlProcessor (`Sitecore.Ecommerce.Catalogs.NameProductUrlProcessor`) implementation of the ProductUrlProcessor contract uses product names.
- The NameAndCodeProductUrlProcessor (`Sitecore.Ecommerce.Catalogs.NameAndCodeProductUrlProcessor`) implementation of the ProductUrlProcessor contract uses product names and codes.
- The CodeProductUrlProcessor (`Sitecore.Ecommerce.Catalogs.CodeProductUrlProcessor`) implementation of the ProductUrlProcessor contract uses product codes.

By default, product URLs begin with the path to the page that links to the product. For example, if the item named `products` under the home item of a managed website contains a link to a product named `product_name` and code `product_id`, the default URL generated for that product is `/products/product_name.aspx`, `/products/product_name_product_id.aspx`, or `/products/product_id.aspx`, depending on the ProductUrlProcessor implementation that SEFE applies. For more information about the ProductUrlIProcessor implementation that SEFE applies, see the section *How to Specify the Product URL Format*.

3.9.1 The SEFE VirtualProductResolver Contract

The VirtualProductResolver contract (`Sitecore.Ecommerce.Catalogs.VirtualProductResolver`) defines an API to determine the product specified by a URL generated by a ProductUrlProcessor implementation. The VirtualProductResolver applies the ProductUrlProcessor appropriate to the context to determine the product specified by the URL. The `ProductResolver` processor that SEFE adds to the `httpRequestBegin` pipeline defined in the `web.config` file uses the VirtualProductResolver to determine the product associated with a requested URL.

The class that defines the VirtualProductResolver contract also serves as the default implementation of the VirtualProductResolver contract. For more information about product URLs and product resolution, see the section *Product URLs and Product Resolution*.

3.9.2 How to Add a ProductUrlProcessor Implementation

You can add a ProductUrlProcessor implementation to define a custom format for product URLs.

To add an implementation of the ProductUrlProcessor contract:

1. In the **Visual Studio** project, add a class that inherits from the ProductUrlProcessor base class (`Sitecore.Ecommerce.Catalogs.ProductUrlProcessor`).

2. In the new class, implement a constructor that accepts an object based on the ISearchProvider contract. For more information about the ISearchProvider contract, see the section *The SEFE ISearchProvider Contract*.
3. In the new class, implement the `GetProductUrl()` method to return the URL to use for a product.
4. In the new class, implement the `ResolveProductItem()` method to return the product item associated with a URL of a product.
5. In Unity configuration, add a `/unity/alias` element. Set the `name` attribute of the new `/unity/alias` element to the name of the class. Set the `type` attribute of the new `/unity/alias` element to the .NET type of the class. For example:

```
<alias name="MyProductUrlProcessor"
  type="MyNamespace.MyProductUrlProcessor, MyAssembly" />
```

6. In Unity configuration, add a `/unity/container/register` element. Set the `type` attribute of the new `/unity/container/register` element to `ProductUrlProcessor`. Set the `mapTo` attribute of the new `/unity/container/register` element to the `name` attribute of the new `/unity/alias` element. Set the `name` attribute of the new `/unity/container/register` element to a unique prefix based on the implementation, such as `My`. Copy the elements enclosed in one of the other `/unity/container/register` elements with a value of `ProductUrlProcessor` for the `type` attribute. For example:

```
<register type="ProductUrlProcessor"
  mapTo="MyProductUrlProcessor" name="My">
  <lifetime type="perthread" />
  <constructor>
    <param name="searchProvider">
      <dependency name="FastQuerySearchProvider"/>
    </param>
  </constructor>
</register>
```

7. In the **Content Editor**, beneath the `/Sitecore/System/Modules/Ecommerce/System/Display Product Modes` item, insert a ProductUrlProcessor definition item using the `Ecommerce/Settings/Settings Item` data template. Give the new ProductUrlProcessor definition item a meaningful name based on the implementation, such as `My Product Url Processor`.
8. In the new ProductUrlProcessor definition item, in the **Data** section, in the **Key** field, enter the value of the `name` attribute of the new `/unity/container/register` element, for example `My`.

For more information about Unity configuration, see the section *Unity Application Block Overview*.

Chapter 4 Adding Customized Product Search Criteria

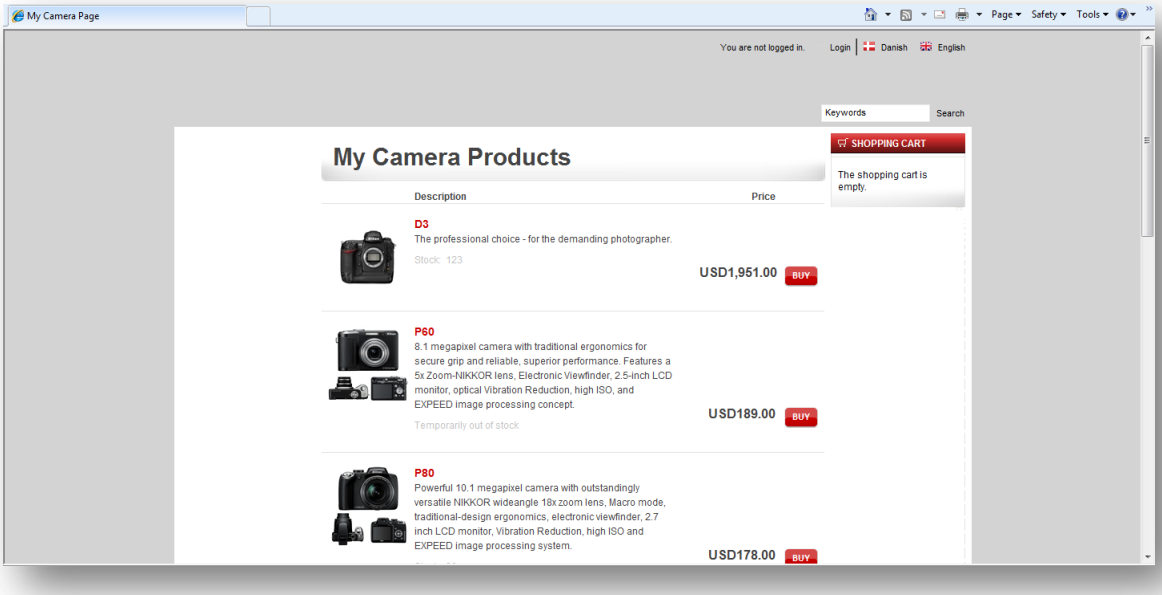
This chapter describes how to extend the product search feature in SEFE. It shows how to customize the search options and how to have more control over the products presentation in both of the frontend and backend. By the front end we mean the display of the search end result for the page visitor and by the backend we mean the content editor and template manager views.

This chapter contains the following sections:

- The Need for the Product Search Configuration and Extensibility
- Extending the Product Search Group Template
- Extending the Resolve Strategy
- Extending the Product Search Catalog

4.1 The Need for the Product Search Configuration and Extensibility

To illustrate the need for changing in the product search, the typical example is that of a camera and photographic supply webshop. This is because it is full of different models, categories, proficiency levels and interrelated products. Hence, sellers do not usually show all the cameras together but they rather show each camera with the proper group of products of the same proficiency level. For example, professional cameras usually exist with professional lenses and others accessories. Moreover, one product can exist in multiple groups. This chapter is useful in general when there is a need to create any classification that is different from that of the repository.



4.1 Extending the Product Search Group Template

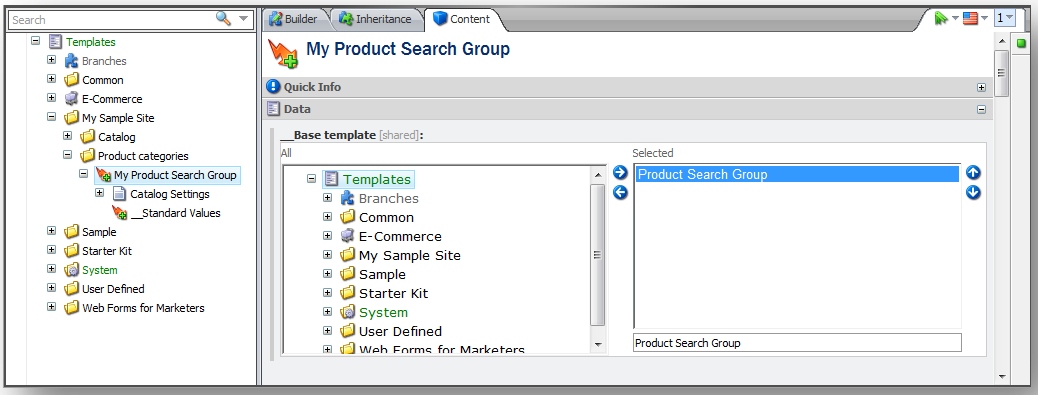
This section describes how to classify a product according to your business needs. You must create or edit the classifications that you need in the *Product Search Group* template.

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 not 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. This is to apply an additional filter to the products selected.

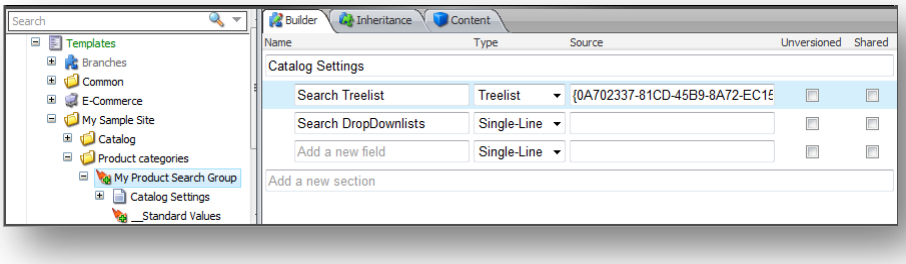
To extend the *Product Search Group* template:

1. Login into the **Content Editor**.
2. Create a new template that inherits from the *Product Search Group* template and call it *My Product Search Group*.



3. In the **Builder** menu, 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.

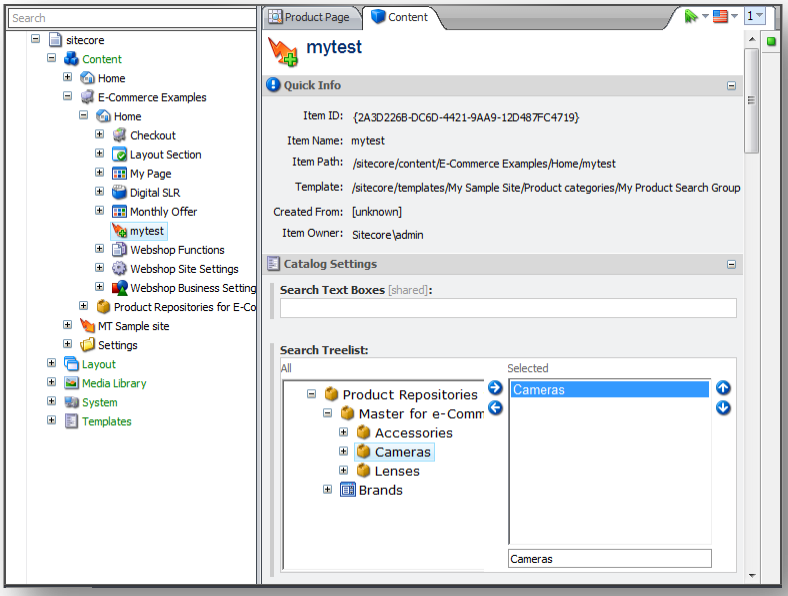
In the **Source** field, enter the path to the product repository.



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

   At this point, you should be able to set the domain of your search from the Treelist.

In the following image, the search is executed for the Cameras domain only.



4.2 Extending the Resolve Strategy

To search for products in the 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 several functions:

- Indexer — It 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.
- Crawler — Traverses a storage system and uses the indexer to populate the search index.
- Monitor — 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 SEFE:

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;

            // <summary>
            // 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
            // </summary>
            public class SampleDatabaseCrawler : Sitecore.Ecommerce.Search.DatabaseCrawler
            {
                // <summary>
                // Overridden method for adding special fields to the Lucene product index
                // </summary>
                // <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 to retrieve the products that should be displayed on a given webpage without having to worry about the actual storage of search criteria and how to search the products selected on the webpage item.

You can configure the implementations of the `ICatalogProductResolveStrategy` contract to search for specific fields on the webpage item in the repository to return the products on the webpage item.

You can use this contract to retrieve the products shown on a given page in two situations:

- While creating the web index.
- While configuring a webpage.

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 stored on the webpage item. (`sitecore/system/Modules/Ecommerce/System/Product` Selection Method)

QueryCataloPgroductResolveStrategy

You can use this class to retrieve the products that rewsults 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 for the search as follows:

1. In **Visual Studio**, open the project named `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 so we can 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 readonly 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>
                // <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 SEFE
                    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 offcourse need to 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 SEFE 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.

```
                <!-- Products index - Used by SEFE 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 MT engros products are stored-->
                        <!--<Root>{054AEC0D-9D92-4C3A-80AC-A0E78773EAB7}</Root>-->
                        <!-- Repository root where SEFE engros 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 MT engros products are stored-->
                        <!--<Root>{054AEC0D-9D92-4C3A-80AC-A0E78773EAB7}</Root>-->
                        <!-- Repository root where SEFE engros 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.

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

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

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

4.3 Extending the Product Search Catalog

This section describes how to extend the Product Search Catalog to accommodate for 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 is used for building the search query used by SEFE when querying the product repository.

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

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

1. In **Visual Studio**, open the project named `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 termplate, if nothing is defined
                    if (string.IsNullOrEmpty(searchTreelistFieldText))
                    {
                        return query;
                    }

                    // Getting the configuration from SEFE
                    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 | separeted 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
                    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))
```

```
                    {
                        // Sppending 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 ofcourse 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 beforehand, and return the result.

To create a products source, you should extend the class named `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 call it `ProductsSource`:

```
        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();
                }
            }
        }
```

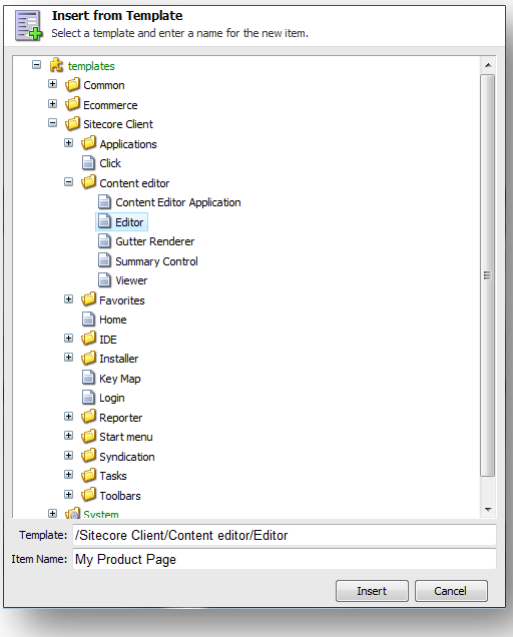
Defining a New Editor in the Core Database

When you create a product catalog, you must also define a new editor in the Core database. The editor is the location where you place the search catalog.

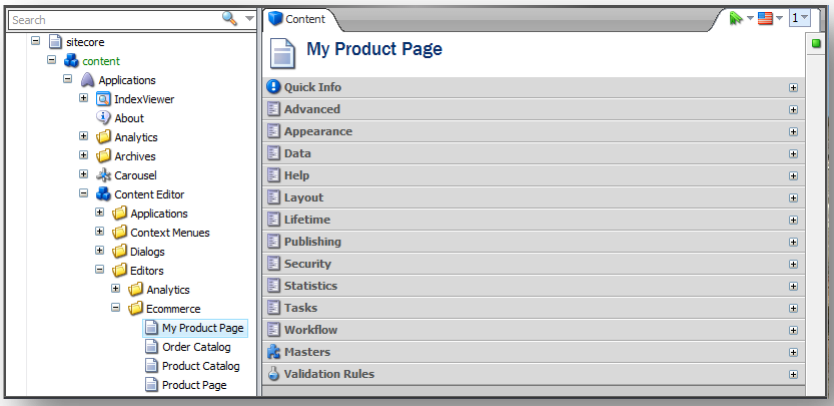
To create the editor:

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 as follows.



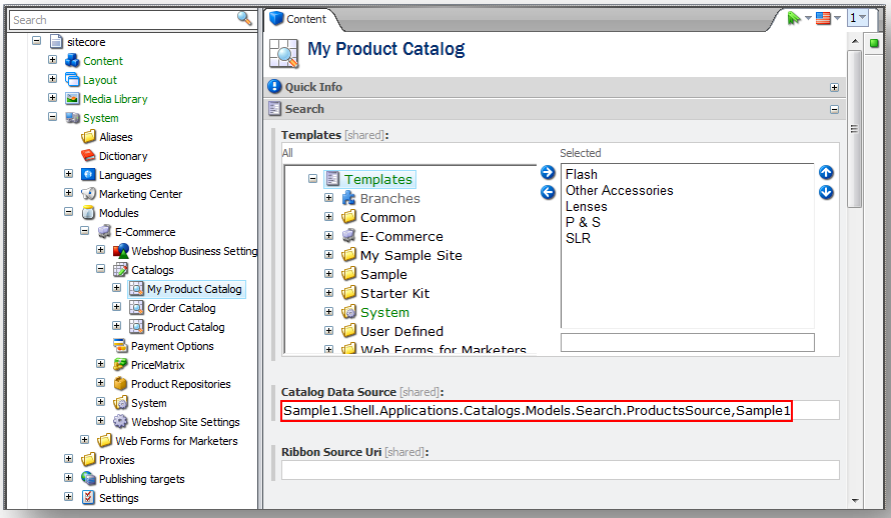
Creating a Product catalog

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 as illustrated in the following image.



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`) and in the **Content** menu, in the **Editors** field, click **Edit** and select the editor you defined in the last section — *My Product Page*.