Sitecore Commerce Connect

The Commerce Connect Integration Guide

A Developer's Guide to integrating Commerce Connect with an external commerce system

Table of Contents

Chapter 1

Introduction

Commerce Connect is an e-commerce framework designed to integrate Sitecore with different external commerce systems and at the same time incorporate customer engagement functionality provided in the Sitecore Customer Engagement Platform (CEP).

Commerce Connect consists of an integration API that incorporates customer tracking by triggering goals and page-events and uses engagement automation plans for following-up on customer interaction, both as part of the API implementation. In addition Commerce Connect comes with e-commerce specific rendering rule conditions for acting on the customer interactions, cart content and orders placed etc.

For a general introduction and overview of the components in Commerce Connect, see the Commerce Connect Overview document.

This guide describes the architecture, API and configuration of Commerce Connect for backend developers who create connectors for integration with external commerce systems.

If you are a developer who create Sitecore solutions and are looking for information about how to use Commerce Connect for creating B2C or B2B shops with e-commerce functionality, see the Commerce Connect developer's guide

- **Chapter 1 — Introduction**
  This chapter contains an introduction for this guide.

- **Chapter 2 — Integrating with Commerce Connect**
  This chapter describes the Commerce Connect Connector architecture and how to customize it as a backend developer creating integration with an external commerce system.

Chapter 2

Integrating with Commerce Connect

A Commerce Connect connector is needed to integrate Commerce Connect with an external commerce system (**ECS**). The typical connector consists of a number of custom processors inserted, in the Commerce Connect defined pipelines, and works with the ECS, either directly or through a web service.

Different service layers work independently of each other and can therefore be integrated independently. For example, products, carts and prices could be integrated from three different systems.

There are common features and unique features for each ECS and typically data is stored in different ways.

2.1 Overview

All of the Commerce Connect integration service layers are based on:

- A customizable domain model.

- An API exposed as a service layer with the methods accepting customizable request objects and returning customizable result objects.

- A number of pipelines, one or more per service method.

The following sections describe each of the above bullet points in more details

2.1.1   A Customizable Domain Model

Each service layer contains a set of entity classes reflecting the domain model. The domain model objects are used when operating with the APIs. The APIs accepts the objects as part of the input parameters and return objects.

The domain model has purposely been kept at a minimum, knowing that all vendors to some degree store different information and one model will not fit all. However the domain models include enough information, domain objects and parameters to cover the common scenarios that are used in all shops.

It is therefore expected that some of the domain model objects are customized for each integration with a different ECS. Commerce Connect might contain domain objects that have no corresponding implementation in the ECS and in those cases it is OK to leave them as-is. For more information see the details in the corresponding section for each service layer.

With product synchronization there is in addition to the domain model objects also a corresponding item domain model matching the entity classes. The entities can be customized by changing the configuration section. For more information see the next section and section 2.2.1.

The domain models are customizable so that:

- All domain model objects can be inherited and extended with custom properties

   o All nested objects can be inherited and extended with custom properties

   o All service methods keep the existing defined signature, even when used with customized domain objects

It's recommended to create an abstraction layer on top of the ECS that extracts and manage the information to be exchanged with Commerce Connect. The approach is similar to the Bridge design pattern used in computer science and makes it easier to continuously mange the integration as both Commerce Connect and the ECS evolves over time. It also makes it easier to exchange the information, if the Commerce Connect domain model and the ECS abstraction layer objects have corresponding and similar object signatures.

Some of the service layers saves data in Sitecore as well as pass it on to the ECS. Whenever data is persisted in Sitecore, the Repository pattern is used to manage loading and saving data and makes it easy to replace the actual repository where data is persisted. For more information see the Service Layer API section as well as MSDN and Fowler.

2.1.2   Service Layer API

Every service layer API contains a number of abstract and generic methods for communicating with the ECS. Information flows in both directions. Product information, prices, and stock information needs to be read from the ECS so that it can be presented to the visitor on the UI. Shopping Cart content, customer account information and shipping information must be parsed over to the ECS so that an order can be created.

The default service layers should be sufficient in most cases, but they can be customized and substituted. For more information, see the appropriate sub system below.

Each method on the service layers accepts a single Request object and returns a single Result object. Both the Request and Result objects can be customized individually for each method for maximum flexibility.

That means, the service layer interface remains the same, even when the domain model objects are customers as well as the parameters going in and the results getting returned.

If you have customized the Request or Result object for a method, then you can use the corresponding extension method, which accepts generics.

Example:

The default method signature for adding a line to a shopping cart looks like this:

```
public virtual CartResult AddCartLines([NotNull] AddCartLinesRequest request)
```

and the generic version of the same method looks like this:

```
public static TAddCartLinesResult AddCartLines<TAddCartLinesResult>([NotNull] this
CartServiceProvider cartProvider, [NotNull] AddCartLinesRequest request)
```

Service Methods

Where possible all methods on a service provider should use the following naming convention:

- CreateEntityName (e.g. `CreateCart`)
- GetEntityName (e.g. `GetCart`)
- DeleteEntityName (e.g. `DeleteCart`)
- UpdateEntityName (e.g. `UpdateCart`)

Where possible all methods that manipulate related items on an entity should use the following naming convention:

- AddRelatedEntityName (e.g. `AddLineItem`)
- RemoveRelatedEntityName (e.g. `RemoveLineItem`)
- UpdateRelatedEntityName (e.g. `UpdateLineItem`)

Request Parameters

A service method should only ever take a single request object as a parameter and that request object must inherit from a `ServiceProviderRequest`. Using a single request object instead of multiple parameters means that the same service methods remain usable regardless of the customization. As service methods require additional data to function, simply extending the request object with new parameters will expose the newly required data to the presentation tier without having to modify the service method.

Customizing request objects

There are two options when extending a request object to handle more parameters. The first option is to simply extend the appropriate request class, similar to the example below

```
public class CustomLoadCartRequest : LoadCartRequest
{
    public CustomLoadCartRequest(string shopName, string cartId,
        string userId, string customProperty)
        :base(shopName, cartId, userId)
    {
        this.customProperty = customProperty;
    }

    public string customProperty { get; protected set; }
}
```

In some cases you will not be able to extend a request, in cases like this you can use the property bag on the request to pass down any properties you like:

```
request.Properties["customProperty"] = "customValue";
```

Result objects

Result objects more or less mirror request objects, but the only difference is that they inherit from `ServiceProviderResult` and have a collection for system messages. Where possible you should always try to return messages as opposed to throwing exceptions. There will be occasions where it makes sense to thrown an exception but graceful recovery and exceptions are expensive actions.

You can set messages on a result very simply by following the pattern below

```
var message = (SystemMessage)this.entityFactory.Create("SystemMessage");
message.Message = "your custom error message goes here";
args.Result.SystemMessages.Add(message);
```

2.1.3   Pipelines

Each service method launches a pipeline with the same name. One or more additional or shared pipelines might be called and executed as part of the initial pipeline being executed, for example `SaveCart` or `SynchronizeProductArtifacts`.

In Sitecore, the default pipeline arguments contain `Request` and `Result` properties, which have `Properties` of type `dictionary` and it can contain arbitrary data to be used by pipeline processors.

Commerce Connect uses the `Request.Properties` dictionary to store data that you need to synchronize. There are processors that read and write the custom data.

Values that are stored in `Request.Properties` are internal temporary data used to carry information between the processors in the pipeline. For example, the `CreateOrResumePipeline` includes the `FindCartInEAState` processor that stores the ID of the cart. This ID is then used the `RunLoadCart` processor to specify the ID of the cart to be loaded.

Data read and stored in the `Request.Properties` dictionary is visible between processors within the pipeline.

The following table contains the data of the cart related the pipelines stored in the pipeline arguments `Request.Properties`:

Pipeline	Property Name	Data Description
`CreateOrResumeCart`	`CartId`	Holds the ID of the cart found in the writer processor and consumed by the reader processor in order to load the cart from the external system. Writer processor `FindCartInEaState` Reader processor `RunLoadCart`
`ResumeCart`	`CartSourceStateId`	Holds the ID of the cart state that the `MoveVisitorToInitialState` processor moves visitors from. Writer processor: `CheckCanBeResumed` Reader processor: `MoveVisitorToInitialState`
	`PreviousStateName`	Holds the name of state that the `TriggerCartResumedPageEvent` processor uses to resume a cart from. Writer processor `CheckCanBeResumed` Reader processor `TriggerCartResumedPageEvent`
	`CartDestinationStateId`	Holds the ID of the cart state that the `MoveVisitorToInitialState` processor moves visitors to. Writer processor `CheckCanBeResumed` Reader processor `MoveVisitorToInitialState`

The following table contains the data of the product related pipelines:

Pipeline	Property Name	Custom Data Description
`GetSitecoreProductList`	`SitecoreProductIds`	Holds a list of the product IDs of Sitecore. Writer processor `GetSitecoreProductList` Reader processor `EvaluateProductListUnionToSynchronize`

Pipeline	Property Name	Custom Data Description

`SynchronizeClassifica tions`	`SitecoreClassificati onGroups`	Holds the classification groups in Sitecore to be synchronized.
		
		Writer processor:
		`ReadSitecoreClassifications`
		Reader processor
		`ResolveClassificationsChanges`

	`ClassificationGroups`	Holds the classification groups in the external commerce system to be synchronized. Writer processor: `ReadExternalCommerceSystemClas sifications` Reader processors: • `ResolveClassificationsCh anges` • `SaveProductClassificatio nsToSitecore`

`SynchronizeClassifica tionsSpecifications`	`ProductClassificatio nGroups`	Holds the product classification groups to be synchronized.
		
		Writer processor:
		`ReadExternalCommerceSystemClas sificationsSpecifications`
		Reader processor:
		`SaveClassificationsSpecificati onsToSitecore`

`SynchronizeDivisions`	`SitecoreDivisions`	Holds product divisions in Sitecore to be synchronized.
		
		Writer processor
		`ReadSitecoreDivisions`
		Reader Processor
		`ResolveDivisionsChanges`

	`Divisions`	Holds the product divisions in the external commerce system to be synchronized. Writer processors: • `ResolveManufacturersChan ges` • `ReadExternalCommerceSyst emManufacturers` Reader processor: `ResolveDivisionsChanges`

Pipeline	Property Name	Custom Data Description
`SynchronizeManufacturers`	`SitecoreManufacturers`	Holds the Sitecore manufacturer to be synchronized. Writer processor `ReadSitecoreManufacturers` Reader processor `ResolveManufacturersChanges`
	`Manufacturers`	Holds a list of the manufacturers in the external commerce system to be synchronized. Writer processor: `ReadExternalCommerceSystemManufacturers` Reader processors: • `ResolveManufacturersChanges` • `SaveManufacturersToSitecore`
`SynchronizeProductEntity`	`ProductFromSitecore`	Holds the products in Sitecore to be synchronized with the external commerce system. Writer processor: `ReadProductFromSitecore` Reader processor: `ResolveProductChanges`
	`Product`	Holds a product from the external commerce system with Sitecore. Writer processor: • `ReadExternalCommerceSystemProduct` • `ResolveProductChanges` Reader processor: `ResolveProductChanges`
`SynchronizeTypes`	`SitecoreProductTypes`	Holds the product types in Sitecore to be synchronized with the external commerce systems. Writer processor: `ReadSitecoreTypes` Reader processor: `ResolveTypesChanges`

Pipeline	Property Name	Custom Data Description

	`ProductTypes`	Holds the product types in the external commerce systems to be synchronized with Sitecore.
		
		Writer processor:
		`ReadExternalCommerceSystemTypes`
		Reader processors:
		    `ResolveTypesChanges` `SaveTypesToSitecore`

`SynchronizeGlobalSpecifications`	`Specifications`	Holds the product specifications to be synchronized.
		
		Writer processor:
		`ReadExternalCommerceSystemGlobalSpecifications`
		Reader processor:
		`SaveGlobalSpecificationsToSitecore`

`SynchronizeProductDivisions`	`DivisionIds`	Holds the division IDs to be synchronized.
		
		Write processor:
		`ReadExternalCommerceSystemProductDivisions`
		Reader processor:
		`SaveProductDivisionsToSitecore`

`SynchronizeProductManufacturers`	`ManufacturerIds`	Holds the manufacturer IDs to be synchronized.
		
		Writer processor:
		`ReadExternalCommerceSystemProductManufacturers`
		Reader processor:
		`SaveProductManufacturersToSitecore`

`SynchronizeProductResources`	`ProductResources`	Holds the product resources to be synchronized.
		
		Writer processor:
		`ReadExternalCommerceSystemProductResourceBase`
		Reader processor:
		`SaveProductResourcesToSitecore`

Pipeline	Property Name	Custom Data Description
SynchronizeProducts	ExternalCommerceSystemProductIds	Holds the product IDs in the external commerce systems to be synchronized. Writer processor: GetExternalCommerceSystemProductList (pipeline: GetExternalCommerceSystemProductList) Reader processor: EvaluateProductListUnionToSynchronize
SynchronizeProductTypes	ProductTypeIds	Holds the product type IDs to be synchronized. Writer processor: ReadExternalCommerceSystemProductTypes Reader processor: SaveProductTypesToSitecore
SynchronizeResources	Resources	Holds the product resources to be synchronized. Writer processor: ReadExternalCommerceSystemResources Reader processor: SaveResourcesToSitecore
SynchronizeProductRelations	RelatedProducts	Holds the related products to be synchronized. Writer processor: ReadExternalCommerceSystemProductRelationsBase Reader processor: SaveProductRelationsToSitecore
SynchronizeProductResources	ProductResources	Holds the product resources to be synchronized. Writer processor: ReadExternalCommerceSystemProductResourceBase Reader processor: SaveProductResourcesToSitecore

Pipeline	Property Name	Custom Data Description
`SynchronizeTypeSpecifications`	`SpecificationCollection`	Holds the specification collection to be synchronized. Writer processor: `ReadSitecoreTypeSpecifications` Reader processor: `SaveTypeSpecificationsToExternalCommerceSystem`

2.1.4   Passing Data between Pipeline Components

While all pipeline components in a pipeline should operate independently without knowledge of what another component has done, in reality there are going to be occasions (such as performance) where you will need to pass information between components to avoid repeating the same action over and over again.

For situations like this the base request object has a property called `RequestContext` with a property bag where you can store any information you need to pass between components.

```
request.RequestContext.Properties["componentSensitiveData"] = "customValue";
```

2.1.5   System Messages

The base result object returned from all pipeline requests contains a `SystemMessages` collection, the collection should be used by all pipeline processors to communicate any messages from the ECS to the presentation tier.

2.1.6   Success

The base result object returned from all pipeline requests contains a Boolean property called Success. This property should be used to indicate to the presentation tier that the request initially passed down to the pipeline was, or was not executed successfully. If a request fails to execute successfully, it is recommend that you add a message about the failure into the `SystemMessages` collection, as well as setting the Success property to false.

2.2 Configuration

Each service layer has its associated configuration stored in a separate include configuration file:

- Cart Service Layer – `/App_Config/Include/Sitecore.Commerce.Carts.Config`
- Orders Service Layer – `/App_Config/Include/Sitecore.Commerce.Orders.Config`
- Inventory Service Layer –
  `/App_Config/Include/Sitecore.Commerce.Inventory.Config`
- Customers and Users Service Layer –
  `/App_Config/Include/Sitecore.Commerce.Customers.Config`
- Pricing Service Layer – `/App_Config/Include/Sitecore.Commerce.Prices.Config`
- Product Synchronization Service Layer –
  `/App_Config/Include/Sitecore.Commerce.Products.Config`
  There's an additional config file
  `Sitecore.Commerce.Products.DelayedSyncProductRepository.config.disabled`
  which can be enabled if the synchronization of products into the Bucket should occur at the end
  of Commerce Connect synchronization instead of immediately. For more information see section
  Error! Reference source not found..

The `Sitecore.Commerce.Config` file contains the global configuration of Commerce Connect:

- `EntityFactory`
- `EaPlanProvider`
- `VisitorFactory`
- `ItemClassificationService`
- `CommereContext`

Each is described in the following sections

2.2.1   EntityFactory

All entities used in Commerce Connect can be customized via configuration using an entity factory. The
entity factory is based on the Factory design pattern and the default implementation is based on the
standard Sitecore factory.

If another factory, Dependency Injection (DI) or Inversion of Control (IOC) implementation is preferred, the
default implementation can be replaced.

In order to use a custom factory it is necessary to perform the following two steps:

1. Create new custom factory class and implement `IEntityFactory` interface.

    The interface has one Create method that accepts a string containing the name of the entity to be
    instantiated

```
namespace Sitecore.Commerce.Entities
{
  /// <summary>
  /// Creates an entity by entity name. The IEnityFactiry allows to substitute the
default entity with the extended one.
  /// </summary>
  public interface IEntityFactory
  {
    /// <summary>
    /// Creates the specified entity by name.
    /// </summary>
    /// <param name="entityName">Name of the entity.</param>
    /// <returns>The entity.</returns>
```

```
        [NotNull]
        object Create([NotNull] string entityName);
    }
}
```

2. Register custom `EntityFactory` class in Sitecore.Commerce.config.

To do this, change type attribute value of "entityFactory" element to the custom EntityFactory type.

```
<!--  ENTITY FACTORY
            Creates an entity by entity name. Allows to substitute default entity with
extended one.
        -->
        <entityFactory type=" Sitecore.Commerce.Entities.EntityFactory, Sitecore.Commerce"
singleInstance="true" />
```

The default implementation lookup the actual type to instantiate in the configuration. Each service layer has its own section called `<commerce.Entities>`. Below are the default entities for Carts listed:

```
<!--  COMMERCE ENTITIES
            Contains all the Commerce Connect cart entities.
            The configuration can be used to substitute the default entity implementation
with extended one.
        -->
        <commerce.Entities>
          <CartBase type="Sitecore.Commerce.Entities.Carts.CartBase, Sitecore.Commerce" />
          <Cart type="Sitecore.Commerce.Entities.Carts.Cart, Sitecore.Commerce" />
          <CartAdjustment type="Sitecore.Commerce.Entities.Carts.CartAdjustment,
Sitecore.Commerce" />
          <CartLine type="Sitecore.Commerce.Entities.Carts.CartLine, Sitecore.Commerce" />
          <CartProduct type="Sitecore.Commerce.Entities.Carts.CartProduct, Sitecore.Commerce"
/>
          <CartOption type="Sitecore.Commerce.Entities.Carts.CartOption, Sitecore.Commerce"
/>
        </commerce.Entities>
```

In order to use a custom entity it is necessary to perform the following two steps:

1. Create a new Entity class

2. Register the custom Entity class in the configuration section `<commerce.Entities>`.

To do this, change type attribute value of "entityFactory" element to the custom EntityFactory type.

2.2.2   EaPlanProvider

This class is used to figure out an engagement plan name based on the current store name in combination with and engagement plan name or state name. It is possible to implement your own version of this by implementing `IEaPlanProvider` and registering that class name in the `eaPlanProvider` section of the Sitecore.Commerce.config.

2.2.3   VisitorFactory

This class is used to get the id of the current runtime user. The default implementation is dependent on Sitecore Analytics for tracking; if this does not suit your needs you can change it by extending the `VisitorFactory` class and overriding the `GetVisitor` method.

Below is a copy of how the default instance works. Once you get the id of your user from the ECS you should populate the `Tracker.Visitor.ExternalUser` with that id, and from that point on this id will be returned by the `VisitorFactory`. If no id is available from the external user then the id created by Sitecore Analytics will be used instead.

```
public virtual string GetVisitor()
{
 var user = Tracker.Visitor.ExternalUser;
 if (string.IsNullOrEmpty(user))
 {
   user = Tracker.Visitor.VisitorId.ToString();
 }
 return user;
}
```

2.2.4   ItemClassificationService

This is a simple class that is used to help determine what type something is, the current version is used to verify if an item is a product, if a template is a product template, and to get the product id from an item.

2.2.5   CommerceContext

The `CommerceContext` class is used to determine the current product and inventory location that is the focus of the site. This class is currently only used by the inventory rule conditions when no stock location or product id is provided to calculate against.

2.2.6   ServiceProviders

Each service layer has its own interface which can be customized, these providers contain the service methods for interacting with the appropriate sub system. All service providers should inherit from `ServiceProvider` and it is recommended to have a generics version of the class in which each service method is generics based.

Sample service method:

```
public virtual GetCartsResult GetCarts([NotNull] GetCartsRequest request)
{
 return this.RunPipeline<GetCartsRequest, GetCartsResult>(PipelineName.GetCarts, request);
}
```

Generics extension method example:

```
public static TGetCartsResult GetCarts<TGetCartsRequest, TGetCartsResult>([NotNull] this
CartServiceProvider cartProvider, [NotNull] TGetCartsRequest request)
 where TGetCartsRequest : GetCartsRequest
 where TGetCartsResult : GetCartsResult, new()
{
 return cartProvider.RunPipeline<GetCartsRequest, TGetCartsResult>(PipelineName.GetCarts,
request);
}
```

If an existing service provider required a new service method, consider extending the service provider and adding the new method instead of creating a whole new service provider. The various sub systems and their service providers are listed below.

Shopping Cart

Sitecore.Commerce.Services.Carts.CartServiceProvider

Orders

Sitecore.Commerce.Services.Orders.OrderServiceProvider

Pricing

Sitecore.Commerce.Services.Prices.PricingServiceProvider

Product Synchronization

Sitecore.Commerce.Services.Products.ProductSynchronizationProvider

Customers and Users

Sitecore.Commerce.Services.Customers.CustomerServiceProvider

Inventory

Sitecore.Commerce.Services.Inventory.InventoryServiceProvider

2.3 Service Layer Specifics

Each service layer in Commerce Connect follows the same design pattern, in the sections below the specific properties and configuration options unique to each service layer are described.

2.3.1   Cart Service Layer

Different ways to work with an ECS

The cart integration can be done in four different ways:

1. The cart is only passed to the external commerce system when submitting an order.
    If the integration is made against an ERP system, shopping cart functionality is typically not provided and needs to be handled elsewhere, in this case in Commerce Connect.
    In this case the cart related pipelines only contain the default Commerce Connect provided processors. This is very easy to setup as no work is involved in creating the integration other than adding a pipeline in the Orders service layer, which will take the shopping cart as input for creating an order.
2. The cart is only "saved" in the external commerce system after each change (`OnSave`)
    This option will minimize the number of calls to the external commerce system and thereby improve performance. It will however be more difficult for the external commerce system to act upon updates made to the cart in Sitecore, like making changes to cart lines when products are added to the cart e.g.:
    *   There might be a discount that needs to be added due to a sale or due to adding a bundled product or a certain combination of products triggering a discount.
    *   There might be an additional "free" product that needs to be added due to a sale.
3. All cart actions are forwarded to the external commerce system (`AddLine`, `UpdateCart`, etc.).
    This option provides the most flexibility for advanced scenarios as explained in #2, but it also makes more calls to the external commerce system decreasing performance.
4. Cart data is only persisted in the external commerce system. In order to do so, the Commerce Connect specific processors `LoadCartFromEAState` (`LoadCart`), `SaveCartToEAState` (`SaveCart`), `FindCartInEAState` & `RunResumeCart` (`CreateOrResumeCart`), `DeleteCartFromEaState` (`deleteCart`) and `BuildQuery` + `ExecuteQuery` (`GetCarts`) must be removed from the pipelines mentioned in parentheses.

The default is option number 2.

Configuration

All configuration for the cart subsystem can be found in the Sitecore.Commerce.Carts.config file. Here you will find all details for the entities, pipelines, and repositories used by the cart system.

Entities

The default cart entities for Commerce Connect only assume some of the basic cart information that will be used across all commerce systems, it is expected that you will need to extend these entities. When you need to extend any of the default entities you can achieve this by creating a new class that inherits from the appropriate type, and then patching the appropriate entity under `<commerce.Entities>` in the Sitecore.Commerce.Carts.config file. You can read more about individual entities in the Developer Guide.

Storing a copy of the cart locally

Commerce Connect gives you the option of storing a copy of your cart locally to help reduce round trips to your ECS or implement functionality that the destination ECS might now support. You are not required to use this functionality, and you will not miss out on any functionality by not using it.

If you are not going to make use of this functionality the Commerce Connect specific processors `LoadCartFromEAState` (LoadCart), `SaveCartToEAState` (SaveCart), `FindCartInEAState` & `RunResumeCart` (CreateOrResumeCart), `DeleteCartFromEaState` (deleteCart) and `BuildQuery` + `ExecuteQuery` (GetCarts) must be removed from the pipelines mentioned in parentheses.

To store the data locally you must create a class that implements `Sitecore.Commerce.Data.Carts.ICartRepository` and patch the `eaStateCartRepository` element in the Sitecore.Commerce.Carts.config with the new full class name. Commerce Connect ships with two sample repositories called `EaStateSqlBasedCartRepository` and `EaStateCartRepository`, these are only sample repositories and should not be used in a production scenario.

Abandoned Cart Engagement Automation plan

The plan is provided as a branch template and multiple instances can be created. There should be one instance per shop. The default plan can be customized with different or more states as is needed.

To make the plan work, two OBEC specific conditions and an action has been provided:

- Condition: Has Empty Cart?
  The condition will retrieve the cart of the current visitor and check if it's empty or not, e.g. if there are any cart lines in it. By default this will on work with one cart per user.

- Condition: Has Provided E-mail?
  The condition will retrieve an e-mail for the current visitor if they have one.

- Action: Set Cart Status
  The status of the cart itself is also set to "abandoned". It is reflected when searching for carts across all visitors using the `GetCarts` method on the service layer.

The plan also uses the standard Send E-Mail Message action, which is provided with CEP, to send out the notification e-mail informing the user about the abandoned cart an encouraging him/her to return and complete the purchase.

2.3.2 Orders Service Layer

The orders service layer is essentially an extension of the cart service layer.

Configuration

All configuration for the order subsystem can be found in the Sitecore.Commerce.Orders.config file. Here you will find all details for the entities, pipelines, and repositories used by the cart system.

Entities

For the most part the default order entities for Commerce Connect are the same classes used by cart with the exception of `Order` and `OrderHeader`, it is expected that you will need to extend these entities. The `Order` entity simply extends `Cart` and adds an `OrderId` property, and `OrderHeader` extends from CartBase which simple services as a class with some basic info about an Order. When you need to

extend any of the default entities you can achieve this by creating a new class that inherits from the appropriate type, and then patching the appropriate entity under `<commerce.Entities>` in the Sitecore.Commerce.Orders.config file. You can read more about individual entities in the Developer Guide.

New Order Placed Engagement Automation Plan

The plan is provided as a branch template and multiple instances can be created. There should be one instance per shop. The default plan only comes with an initial state and can be customized with different or more states as is needed.

Pipelines

The order layer only ships with four pipelines `submitVisitorOrder`, `getVisitorOrder`, `getVisitorOrders`, and `visitorCancelOrder`. By default all that these pipelines will do is trigger a goal, except for `submitVisitorOrder`, which will also add the order to an Engagement Automation Plan. Each of these pipelines must be filled in with an appropriate processor that knows how to communicate to an ECS. For more details on the requests and results used by these pipelines please check out the Developer Guide.

2.3.3   Inventory Service Layer

The inventory service layer provides read-only integration with inventory / stock information from an ECS. However, this service layer can be extended to support read-write integration if desired.

Pipelines

The pipelines of the inventory service layer can be split into four categories:

1. Runtime Integration
    a. commerce.inventory.getStockInformation
    b. commerce.inventory.getPreOrderableInformation
    c. commerce.inventory.getBackOrderableInformation
2. Search Integration
    a. commerce.inventory.stockStatusForIndexing
3. Event Raising
    a. commerce.inventory.visitedProductStockStatus
    b. commerce.inventory.productsAreBackInStock
    c. commerce.carts.addCartLines
4. Products Back In Stock Engagement Plan
    a. commerce.inventory.visitorSignUpForStockNotification
    b. commerce.inventory.removeVisitorFromStockNotification
    c. commerce.inventory.getBackInStockInformation

Of these pipelines, only the following require integration with the ECS:

- commerce.inventory.getStockInformation
- commerce.inventory.stockStatusForIndexing
- commerce.inventory.getPreOrderableInformation
- commerce.inventory.getBackOrderableInformation
- commerce.inventory.getBackInStockInformation

Extending the other pipelines is purely optional.

Configuration

All configuration for the inventory service layer can be found in the Sitecore.Commerce.Inventory.config file. Here you will find all details for the entities, pipelines, and repositories used by the inventory system. It is highly recommend that you do not modify this file when adding your ECS connector components to the pipelines, overriding entity definitions, etc. Instead, use Sitecore configuration patching, and include all of your ECS configuration in a separate file named {ECSName}.Connectors.Inventory.config.

Entities

In the stock inventory system, there is no inheritance hierarchy for entities, and all of the connect pipelines treat them as read-only entities. If you wish to support updating stock information through the inventory system, you will need to extend the system with your own pipelines and service provider methods.

StockStatus and StockDetailsLevel Entities

The `StockStatus` and `StockDetailsLevel` entities are slightly different from traditional entities, in that they are intended to represent enumeration values. `StockStatus` represents a standard enumeration, and `StockDetailsLevel` represents a flags enumeration. If either of these entities need to be extended for an ECS, the extended entities should also expose constants / read-only properties that represent the possible values for the entity. For example, if extending `StockStatus` to contain a new `Downloadable` value, then the extended `EcsStockStatus` entity should expose a `static readonly` field that represents the `Downloadable` value (i.e. `public static StockStatus Downloadable = new EcsStockStatus(5, "Downloadable);`)

The InventoryProductBuilder

The `InventoryProductBuilder` is a helper class used in the inventory system to build `InventoryProduct` entities based on the current site context, compare `InventoryProduct` entities, etc. If you extend the `InventoryProduct` entity, this class will also need to be extended. Configuration for the `InventoryProductBuilder` is located in configuration at sitecore/inventoryProductBuilder

The InventoryAutomaionProvider

The `InventoryAutomationProvider` is a helper class used by the conditions an actions in the "Products Back in Stock" engagement automation plan to access automation state data as strongly-typed classes. Automation state data in the inventory system is stored as JSON serialized strings. The `InventoryAutomationProvider` is responsible for serializing and deserializing information stored in the automation state data row.

Products Back in Stock Engagement Automation Plan

The plan is provided as a branch template and multiple instances can be created. There should be one instance per shop. The default plan can be customized with different or more states as is needed. Its purpose is to notify customers by email when a product they are interested in is back in stock and available for order.

This automation plan maintains state data that is serialized in JSON format. The following values are used to track customer 'back in stock' notification requests, all of which represent a list of `StockNotificationRequest` objects:

- commerce.productNotifications
  Contains the list of valid notification requests that the customer is interested in.

- commerce.expiredNotifications
  Contains the list of notification requests that have expired.

- commerce.backInStockProducts
  Contains the list of products that are back in stock.

To support this automation plan, two new conditions and two actions have been created.

- Action: Remove Expired Back In Stock Notifications
  This action will update the automation plan state data, and remove 'back in stock' notification requests that have past their interest date. The default interest date is 180 days after the day the customer requested to be notified when a product is back in stock.

- Action: Send Back In Stock Notification Email
  This action sends email to the customers when a product they are interested has become back in stock. This action should be customized for each shop to contain the correct email address and email body branding.

- Condition: Are Products Back In Stock Condition
  This condition checks if any products that customers are interested in have become back in stock. If at least one product is back in stock, this condition will evaluate as true.

- Condition: Has List Of Visitor Notifications Expired Condition
  This condition checks if the customer still has any valid 'back in stock' notification requests. If at least one 'back in stock' notification request exists that has not expired, this condition will evaluate to false.

All of these conditions and actions rely on the `InventoryAutomationProvider` to access automation state data and perform notification comparisons. So, customizing the conditions and actions directly should not be necessary. Instead, the `InventoryAutomationProvider` should be updated to extend any functionality needed in this automation plan.

Extending the InventoryProduct Entity

The `InventoryProduct` is used to uniquely identify a product / stock information in the ECS. If the default `InventoryProduct` is not sufficient to identify stock information, you will need to extend this entity, as well as a few provider classes in the inventory system, using the following steps:

1. Create an `EcsInventoryProduct` class that derives from `InventoryProduct` that contains the information required to identify stock information in your ECS
2. Create an `EcsCommerceContext` that derives from `CommerceContextBase` that exposes properties that represent the additional information required to identify stock information in your ECS. It will be the responsibility of the client site / application to set these properties based on client state.
3. Create an `EcsInventoryProductBuilder` class that derives from `InventoryProductBuilder`, and override all methods of the base class to properly handle your `EcsInventoryProduct`. In particular, you will need to use the new `EcsCommerceContext` inside `CreateInventoryProduct`() to populate the additional properties of you `EcsInventoryProduct`. For example:

```
var ecsProductInfo = ((EcsCommerceContext)this.CommerceContext).EcsProductInfo;
```

4. Create an `EcsInventoryAutomationProvider` class that derives from `InventoryAutomationProvider`, and override the `GetProductNotifications`, `GetExpiredNotifications`, and `GetProductsBackInStock` methods. These methods will need to return an `EcsInventoryProduct` for the `StockNotificationRequest.Product` property. Automation state data in the inventory system is stored as JSON serialized strings, so this will usually require some custom deserialization code.

5. Register your `EcsInventoryProduct` entity at sitecore/commerce.Entities/InventoryProduct
6. Register your `EcsCommmerceContext` at sitecore/commerceContext
7. Register your `EcsInventoryProductBuilder` at sitecore/inventoryProductBuilder
8. Register your `EcsInventoryAutomationProvider` at sitecore/inventoryAutomationProvider

2.3.4 Customers and Users

What is the difference between a User and a Customer

Both of these entities are consumers of your ECS webshop. The User (`CommerceUser`) account is primarily for authentication purposes and exposing the user to DMS. The customer (`CommerceCustomer`) account is for representing the customer in the ECS who will receive and pay for the submitted orders. In simple B2C scenarios the `CommerceUser` and the `CommerceCustomer` represent two aspects of the customer, where in B2B scenarios the `CommerceUser` represents the person acting on behalf of the customer, which typically represents an organization or company.

There is a many-to-many relationship between customers and users and there could be customers without users (anonymous checkout, without registration for example), but normally users would not be without customers.

Different ways to work with an ECS

There are multiple scenarios to use Connect to work with an ECS for customers and users.

Some usage examples of the domain model are:

- To pass customer and user information between the external commerce system and Sitecore
- To set and/or get customer information during checkout
- To register accounts for new users
- To authenticate, e.g. login or logout registered users
- To enter a user into an EA plan when creating a user account and trigger goals when logging in

You can read more about the domain model for customers and users in the Connect Developer Guide.

Configuration

All configuration for the customer subsystem can be found in the Sitecore.Commerce.Customers.config file. Here you will find all details for the entities, pipelines, and repositories used by the customer and user system.

Entities

The default customer entities for Commerce Connect only assume some of the basic customer and user information that will be used across all commerce systems, it is expected that you will need to extend these entities.

There are five (5) entities defined in the Connect system for customers and users, all of which you may choose to extend to suite your needs.

CommerceCustomer

The concept of a customer is determined by the integrated commerce system and the e-shop solution. In B2C solutions, the customer typically represents a person whereas in B2B scenarios a customer typically represents a company.

The `CommerceCustomer` entity will always be extended to include custom information particular to the ECS.

CommerceUser

The `CommerceUser` class is responsible for representing a user account. A user resembles a visitor of a webshop (website) who has identified him- or herself explicitly by creating a login account by which the person can be (re-)authorized.

The `CommerceUser` entity can be extended to include custom information particular to the ECS, but the default implementation will work if users are stored in Sitecore only for authentication purposes.

CustomerParty

`CustomerParty` is used to represent the type and 0-to-many relationship between the customer and a list of parties, where parties are of type `Party`.

CustomerPartyType

`CommercePartyType` is used to indicate the type of relationship between the customer and party.

The class is introduced as an extensible enum. In order to extend and customize the CustomerPartyTypes options. Connect has two different party types, `AccountingParty` and `BuyerParty`.

Party

Party is a shared entity between carts service layer and customer and users service layer. This entity stores party information for example: address information.

When you need to extend any of these default entities you can achieve this by creating a new class that inherits from the appropriate type, and then patching the appropriate entity under `<commerce.Entities>` in the Sitecore.Commerce.Customers.config file.

You can read more about individual entities in the Developer Guide.

Pipelines

There are numerous pipelines for Customers and Users allow most basic functionality.

Some example of the pipelines allow for:

- Creation of Customer and Users via `CreateCustomer` and `CreateUser`.
- Updating of Customers and Users via `UpdateCustomer` and `UpdateUser`.
- Deletions via `DeleteCustomer` and `DeleteUser`.
- Associating Customer to Users via `AddCustomers` and `AddUsers`.
- Adding of `Party` information via `AddParties`.

You can read more about the customer and user pipelines in the Developer Guide.

Chapter 3   Product Synchronization

Sitecore Connect contains a service layer for synchronizing product data between Sitecore and one or more external commerce systems.

Having access to product data is essential for any shop, but using the product synchronization layer is optional with Connect. By design, the service layers work independently and all the other service layers only care about a product ID, which is typically provided in parameters.

Note
For more information about benefits and drawbacks for using product synchronization compared to other approaches like use of data providers, see the Connect Overview document.

Note
For a description of the service layer methods, pipelines and domain objects, see the Sitecore Commerce Connect Developer Guide

The following sections describe:

1. The basics of product synchronization

2. The Connect product data model

3. The item data structure

4. The object domain model

5. A number of examples of how to implement and customize the default implementation

3.1 The basics of product synchronization

There are a couple of different ways to synchronize one or more products ranging from explicit to implicit specification of the products to synchronize:

- Synchronize All Products
  The service method SynchronizeProducts synchronizes all products and related product repositories (a.k.a. artifacts), that needs to be synchronized. A part of the logic, retrieves a list of updated products from the external commerce system and a list from Sitecore and compares them to implicitly determine which products to synchronize and which to delete.
  After determining which products to synchronize, due to being newly added, updated or deleted, the next method SynchronizeProductList is called, specifying the list of products to synchronize.
  Before calling Synchronize Product List, all the related product repositories are synchronized.

- Synchronize Product List
  The service method SynchronizeProductList accepts a list of product IDs which it iterates over and calls the Synchronize Product method.
  No related product repositories are synchronized as part of this, but are assumed to be up-2-date.

- Synchronize Product
  The service method SynchronizeProduct accepts a single product ID for which the data is synchronized.
  No related product repositories are synchronized as part of this, but are assumed to be up-2-date.

- Synchronize Artifacts
  The related product repositories like Manufacturers, Product Types, Classifications (categories) and global specifications are referred to as product artifacts.
  The service method SynchronizeArtifacts will synchronize all the repositories separately.

While synchronizing all or a list of products, a number of Sitecore Disablers are temporarily activated to speed up performance, like EventDisabler, SecurityDisabler etc.

Note
The item IDs generated in Sitecore, for the product data in the external system, are based on a direct mapping of external IDs to Sitecore Item IDs. That means the same specific item ID is always generated for a specific external ID. The implication is, that product data can be synchronized, even if the related product repositories are not up to date. When the related product data is synchronized the connection is automatically established because the Sitecore item ID was already known and configured. For more information see section

3.1.4 Integrating with Connect

Integrating products with Sitecore Commerce Connect via product synchronization requires:

Potentially customizing the product domain model, although the default model will cover most scenarios and carry the needed information for presentation purposes. For more information on customizing the domain model, see section 3.4, The Object Domain Model

The object diagram below visualizes the object domain model. For more information, see the Sitecore Commerce Connect Developer Guide.

There is a one to one correspondence between the product item templates and the objects.
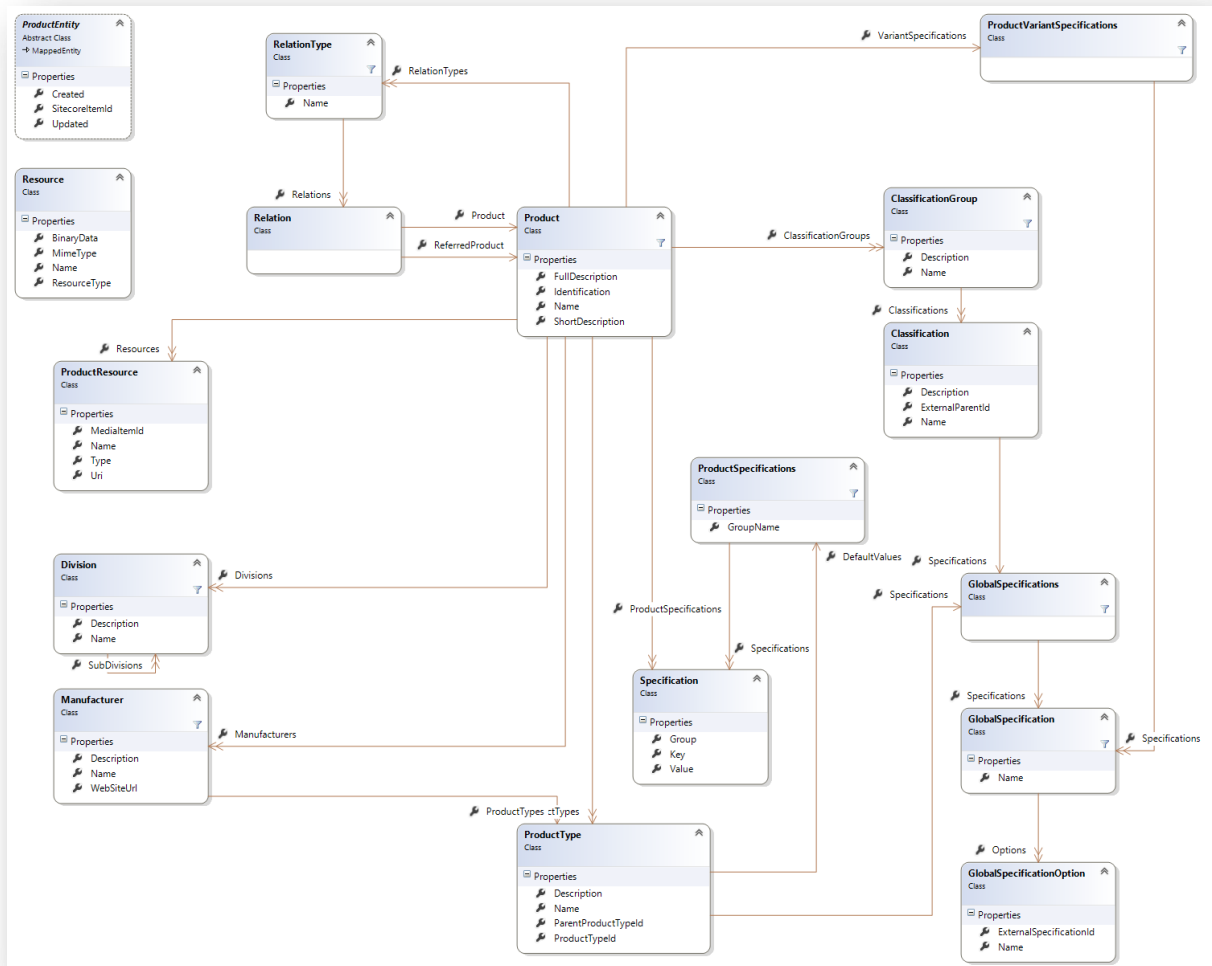
1. How to Implement a Custom Product Entity. The domain model consists of a list of Sitecore product template and corresponding object types.
2. Creating a custom processor for each pipeline that reads data externally and stores it in an instance of the corresponding domain model object type, for which the pipeline is responsible.
3. If the synchronization needs to go both ways, two additional processor must be created for each pipeline
   - A processor which stores the product data externally. The product data will be given in a an instance of the corresponding domain model object type, for which the pipeline is responsible
   - A processor, which resolves the differences and determines the resulting output. For more information, see section 3.8 How to Create a Custom Synchronization Strategy.

Note
When reading data in the external system and populating an instance of a domain model entity (step #2), it's important to provide unique names for entities of the same type so that it will result in unique item names in Sitecore. If not, it cannot be guaranteed that the items can be accessed. An example is

resources, where two images with identical names for the same product, will result in only one of them being shown on the website, because Sitecore always picks the first item with a given name

3.1.1   Repository design pattern

Each Connect processor that writes data to Sitecore content is based on the Repository design pattern and has an associated configuration entry in the Sitecore.Commerce.Products.config file.

In the following snippet, the default configuration for the ManufacturerRepository is displayed.

```xml
        <manufacturerRepository type="Sitecore.Commerce.Data.Products.ManufacturerRepository,
Sitecore.Commerce" singleInstance="true">
        <path ref="paths/manufacturers" />
        <template>{8ECDC0A6-3A85-4F89-8F49-8A53AA75595E}</template>
        <prefix>Manufacturer_</prefix>
        </manufacturerRepository>
```

All repository configuration have the following in common

- A name and a type attribute that refers to the implementation. The name is the element name and the naming scheme is: [entity name in singular] + "Repository", like "manufacturerRepository".
- A <path> parameter element which refers to the main <paths> element, to specify where the root of the repository is located.
- A <template> parameter element that contains the ID of the template that the repository operates on. The template is used when creating new instances of the given item type.
- A <prefix> parameter element containing an arbitrary but fixed prefix, which is used as input to the IDGenerator along with external ID, to ensure the outcome is a unique GUID ID, which can be used as a unique item ID. For more information see section 3.9 How to Implement a Custom ID Generator.

There is a special kind of repository types, whose names start with "product", that doesn't actually store data in a separate repository, but augment the main product entity with references to the separate repositories. An example is productManufacturerRepository that has the responsibility of managing the references between the product item and the related manufacturer items. The configuration for productManufacturerRepository is shown below.

Instead of a template and path element, it has a couple of <param> elements, specifying the name of the field on the product item that holds the references (e.g. item IDs) to the related manufacturers. As these types of repositories needs to generate the right item IDs, they need to know the same prefix as was used in the configuration for the <manufacturerRepository>.

```xml
        <productManufacturerRepository
type="Sitecore.Commerce.Data.Products.ProductFieldRepository, Sitecore.Commerce"
singleInstance="true">
        <param desc="productFieldName">Manufacturer</param>
        <param desc="productPrefix">Product_</param>
        <path ref="paths/manufacturers" />
        <prefix>Manufacturer_</prefix>
        </productManufacturerRepository>
```

Note
Apart from being used in the pipelines to store entities in Sitecore, the repositories can be used to obtain an object instance of the given type by providing an ID.

All repositories provides a method that takes an ID as input and returns an instance of the given entity type:

```
public virtual TEntity Get(string entityKey)
```

Retrieving a specific product looks like this:

```
var product = this.productRepository.Get("external id");
```
ID Mapping,

3.1.2   2-way synchronization

The synchronization provided with Connect is designed to work in both directions. However, the most common scenario is to synchronize only one way, from the external commerce system to Sitecore content.

The logic that determines whether an entity is updated in the ECS, CMS, or both is based on a Direction parameter and the configured strategy. Each synchronization method takes an optional Direction parameter. If not provided the default direction value is Direction.Inbound, which means the product data is taken from the ECS and imported into CMS. The possible Direction values are:

- Direction.Inbound, e.g. one-way synchronization from ECS to CMS. This is the default value.

- Direction.Outbound, e.g. one-way synchronization from CMS->ECS.

- Direction.Both, e.g. 2-way synchronization based on the configured synchronization strategy

The default synchronization strategy is based on timestamps for when the entity was last updated and the last one (newest date and time) wins, meaning if the entity was last updated in the external system, then it gets overwritten in Sitecore and vice versa. Only when specifying 2-way synchronize with Direction.Both will the synchronization strategy be evaluated to determine which way data flows. The strategy is executed per product and all its constituent entities. For more information on customizing the synchronization strategy, see section 3.8 How to Create a Custom Synchronization Strategy

3.1.3   Pipeline pattern

Each type of item that makes of the product domain model is managed individually by following the divide and conquer strategy. As with all other service layers in Connect, the logic is implemented using pipelines. It means that there are one or more pipelines associated with every product entity in the model, where the entity can be product, manufacturer, division, classification etc.

Connect uses a pattern similar to the Bridge Design Pattern for the processors in the pipelines for each type of entity. The product domain model serves as the data abstraction that hides its implementation in the ECS as well as in Sitecore. Each entity is read from both the ECS and Sitecore.

A comparison of the values between identical instances is executed and the result is written back to both the ECS and Sitecore. This means that each pipeline has the same pattern of processors for each entity, where the entity can be product, manufacturer, division, or classification.

The two-way synchronization takes place in the following order:

1. **The entity is read from the ECS.**

   Naming convention: "Read[TypeOfEntity]FromSC"

2. **The entity is read entity from the CMS.**

Naming convention: "Read[TypeOfEntity]FromECS"

3. **The entities are compared and the differences are resolved.**

   Naming convention: "Resolve[TypeOfEntity]Changes"

4. **The results are written to the ECS.**

   Naming convention: "Save[TypeOfEntity]FromECS"

5. **The results are written to the CMS.**

   Naming convention: "Save[T7ypeOfEntity]FromSC"

When implementing integration with an external system, it is processors number 1 and 4, which are relevant to implement. The others come with Connect. There needs to be a custom version of processor number 3, but a base processor is provided which provides most of the logic needed.

Depending on the value of the Direction parameter, some of the previously listed processors skip execution. For example, if the Direction parameter is set to inbound (ECS->CMS), there is no need to read the entity from CMS or write it back to the ECS.

The following snippet shows the default configuration for synchronizing the main product item (a.k.a. ProductEntity). The basic pattern handles the cases of creating and updating. For product, an additional processor is injected to delete a product if it no longer exists in the external commerce system and must be removed from content.

```xml
<commerce.synchronizeProducts.synchronizeProductEntity>
  <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ReadProductFromSitecore,
Sitecore.Commerce">
    <param desc="productRepository" ref="productRepository" />
  </processor>
  <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ReadExternalCommerceSystemProduct, Sitecore.Commerce" />
  <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ResolveProductChanges,
Sitecore.Commerce">
    <param desc="synchronizationStrategy" ref="synchronizationStrategy" />
  </processor>
  <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.DeleteProductFromSitecore,
Sitecore.Commerce">
    <param desc="productRepository" ref="productRepository" />
    <param ref="entityFactory" />
  </processor>
  <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.SaveProductToExternalCommerceSystem, Sitecore.Commerce" />
  <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.SaveProductToSitecore,
Sitecore.Commerce" >
    <param desc="productRepository" ref="productRepository" />
    <param ref="entityFactory" />
  </processor>
</commerce.synchronizeProducts.synchronizeProductEntity>
```
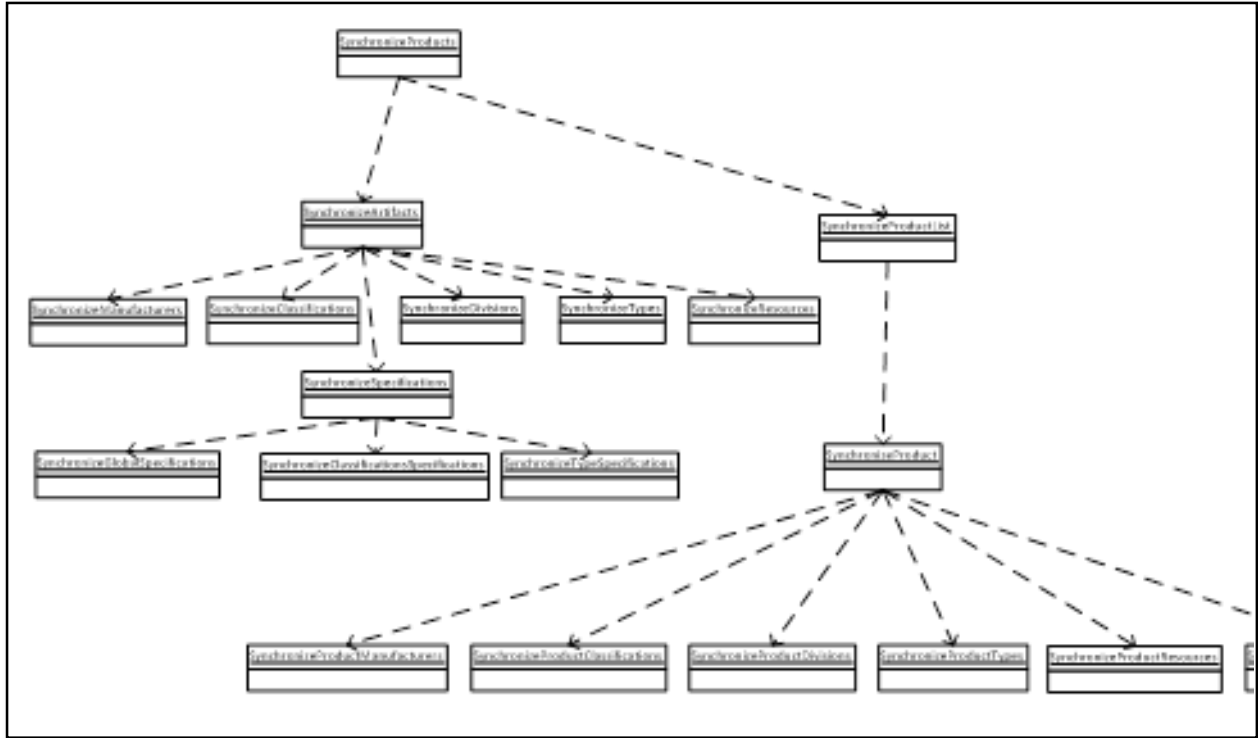
Figure 1 illustrates how the main pipeline SynchronizeProducts are executing other pipelines internally to do a full synchronization.

Figure 1: Call hierarchy between pipelines

Pipelines and naming convention

Pipelines are generally split into two types:

- Pipelines that operate on related product repositories, separate from the actual product repository.
  These separate repositories are used as references from the product repository. There are repositories for Manufacturers, Classifications, Types, Divisions, Resources and specifications.
  The pipeline names are prefixed with "Synchronize". An example is `SynchronizeManufacturers`, which is responsible for synchronizing all manufacturers.

- Pipelines that operate on individual products and synchronize references from products to the separate repositories.
  The pipeline names are prefixed with "SynchronizeProduct", having the word product added to signal that they are dealing with individual products as opposed to entire repositories. An example is `SynchronizeProductManufacturers`, which is responsible for synchronizing the connections/references between a specific product and its related manufacturers stored in the separate Manufacturers repository.

Processors that begins with the word Run are responsible for calling a separate pipeline and transfer the needed parameters. An example is `RunSynchronizeManufacturers,` which executes the `SynchronizeManufacturers` pipeline.

3.1.4   Integrating with Connect

Integrating products with Sitecore Commerce Connect via product synchronization requires:

3.2 Potentially customizing the product domain model, although the default model will cover most scenarios and carry the needed information for presentation purposes. For more information on customizing the domain model, see section 3.4, The Object Domain Model

The object diagram below visualizes the object domain model. For more information, see the Sitecore Commerce Connect Developer Guide.

There is a one to one correspondence between the product item templates and the objects.
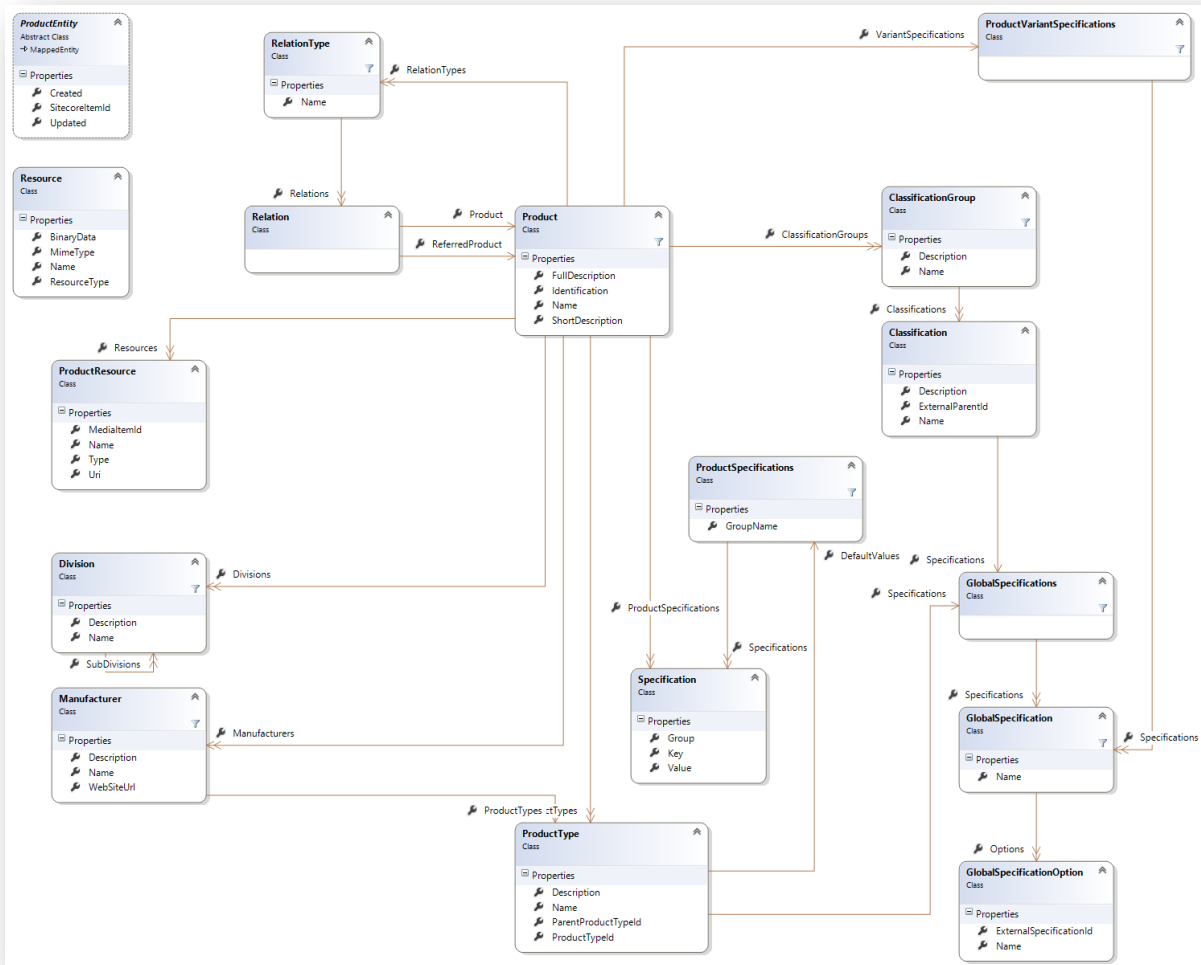
4. How to Implement a Custom Product Entity. The domain model consists of a list of Sitecore product template and corresponding object types.
5. Creating a custom processor for each pipeline that reads data externally and stores it in an instance of the corresponding domain model object type, for which the pipeline is responsible.

6. If the synchronization needs to go both ways, two additional processor must be created for each pipeline
    o A processor which stores the product data externally. The product data will be given in a an instance of the corresponding domain model object type, for which the pipeline is responsible
    o A processor, which resolves the differences and determines the resulting output. For more information, see section 3.8 How to Create a Custom Synchronization Strategy.

Note
When reading data in the external system and populating an instance of a domain model entity (step #2), it's important to provide unique names for entities of the same type so that it will result in unique item names in Sitecore. If not, it cannot be guaranteed that the items can be accessed. An example is resources, where two images with identical names for the same product, will result in only one of them being shown on the website, because Sitecore always picks the first item with a given name

3.2.1   Repository design pattern

Each Connect processor that writes data to Sitecore content is based on the Repository design pattern and has an associated configuration entry in the Sitecore.Commerce.Products.config file.

In the following snippet, the default configuration for the ManufacturerRepository is displayed.

```
      <manufacturerRepository type="Sitecore.Commerce.Data.Products.ManufacturerRepository,
Sitecore.Commerce" singleInstance="true">
        <path ref="paths/manufacturers" />
        <template>{8ECDC0A6-3A85-4F89-8F49-8A53AA75595E}</template>
        <prefix>Manufacturer_</prefix>
      </manufacturerRepository>
```

All repository configuration have the following in common

* A name and a type attribute that refers to the implementation. The name is the element name and the naming scheme is: [entity name in singular] + "Repository", like "manufacturerRepository".
* A <path> parameter element which refers to the main <paths> element, to specify where the root of the repository is located.
* A <template> parameter element that contains the ID of the template that the repository operates on. The template is used when creating new instances of the given item type.
* A <prefix> parameter element containing an arbitrary but fixed prefix, which is used as input to the IDGenerator along with external ID, to ensure the outcome is a unique GUID ID, which can be used as a unique item ID. For more information see section 3.9 How to Implement a Custom ID Generator.

There is a special kind of repository types, whose names start with "product", that doesn't actually store data in a separate repository, but augment the main product entity with references to the separate repositories. An example is productManufacturerRepository that has the responsibility of managing the references between the product item and the related manufacturer items. The configuration for productManufacturerRepository is shown below.

Instead of a template and path element, it has a couple of <param> elements, specifying the name of the field on the product item that holds the references (e.g. item IDs) to the related manufacturers. As these types of repositories needs to generate the right item IDs, they need to know the same prefix as was used in the configuration for the <manufacturerRepository>.

```
            <productManufacturerRepository
type="Sitecore.Commerce.Data.Products.ProductFieldRepository, Sitecore.Commerce"
singleInstance="true">
            <param desc="productFieldName">Manufacturer</param>
            <param desc="productPrefix">Product_</param>
            <path ref="paths/manufacturers" />
            <prefix>Manufacturer_</prefix>
        </productManufacturerRepository>
```

Note
Apart from being used in the pipelines to store entities in Sitecore, the repositories can be used to obtain an object instance of the given type by providing an ID.

All repositories provides a method that takes an ID as input and returns an instance of the given entity type:

```
public virtual TEntity Get(string entityKey)
```

Retrieving a specific product looks like this:

```
var product = this.productRepository.Get("external id");
```

3.2.2   ID Mapping

By design, the remote product repository is always regarded as the main repository, which by default owns the products. That makes the ID of the products and artifacts in the external system the primary key.

In Sitecore, the IDs of the corresponding items for products and artifacts are generated by Connect instead of relying on the default Sitecore implementation that automatically generates a new GUID for each new item created.

By using a hash algorithm, it is possible to generate a direct mapping between the IDs coming from the external system and the item IDs in Sitecore. It has the following benefits:

- No need for mapping tables taking up space.
- It becomes very fast to get the ID of the corresponding item.
- There is no need for searching for the items in Sitecore if the external ID is provided.

The default implementation is based on the MD5 hash algorithm and has the following format:

```
Item.ID = MD5.ComputeHash(Prefix + ExternalID);
```

For more information on creating a custom ID mapping implementation, see section 3.9 How to Implement a Custom ID Generator

3.2.3   Indexing

When Connect is installed the default index is patched to exclude all product data and a separate product index is created, which contains extended product data.

While synchronizing all products or a list of products, the indexing is stopped. After synchronization finishes the indexes are rebuilt. This is done for performance reasons. The configuration snippet below shows the default configuration of the SynchronizeProductList pipeline containing the processors related to indexing.

```
<commerce.synchronizeProducts.synchronizeProductList>
```

```
            <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.PauseSearchIndexing,
Sitecore.Commerce" />
            <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.SynchronizeProductList,
Sitecore.Commerce" />
            <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.ResumeSearchIndexing,
Sitecore.Commerce" />
            <processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.RebuildSearchIndexes,
Sitecore.Commerce" />
        </commerce.synchronizeProducts.synchronizeProductList>
```

In the Sitecore.Commerce.Products.Config file, the setting ProductSynchronization.ProductIndexes contains a comma separated list of index names that are stopped and re-started during product synchronization

```
    <!--  PRODUCT INDEXES.
          The indexes used to store synchronized products.
          Can be stopped, resumed and rebuild automatically during product synchronization.
    -->
    <setting name="ProductSynchronization.ProductIndexes" value="sitecore_master_index,
commerce_products_master_index" />
```

Note

If additional indexes are created, the setting should be updated to include the name of the indexes. If not, indexing will continue during synchronization resulting in performance degradation

Note

If new or custom product templates are introduced, both the DefaultIndexConfiguration and the Product index configurations must be updated. For more information, see the following sections.

The default index

By design, the default master and web indexes are configured, not to include the items stored in the product repositories.. The default index configuration is patched with an Exclude section with an entry for each product template:

```
    <exclude hint="list:ExcludeTemplate">
    <ProductRepositoryTemplateId>{F599BF48-D6FE-40DC-9F78-
CF2D56BFB657}</ProductRepositoryTemplateId>
              <ProductTemplateId>{47D1A39E-3B4B-4428-A9F8-B446256C9581}</ProductTemplateId>
    …
    </exclude>
```

For the full configuration see the configuration file "Sitecore.Commerce.Products.Lucene.DefaultIndexConfiguration.config"

The product index

Connect comes with its own product index for both the master and the web database. The index serves several purposes:

- To separate content from data in two separate indexes
  The product index is used, when searching the product repository bucket from within the Content Editor. This is achieved by patching the getContextIndex pipeline:
  ```
  <contentSearch.getContextIndex>
  ```

```
<processor type="Sitecore.Commerce.Pipelines.ContentIndex.CustomIndex.FetchCustomIndex,
Sitecore.Commerce"
patch:before="processor[@type='Sitecore.ContentSearch.Pipelines.GetContextIndex.FetchIndex
, Sitecore.ContentSearch']"/>
</contentSearch.getContextIndex>
```

- To include extended product data.
  A pipeline "commerce.inventory.stockStatusForIndexing" reads inventory data per product from the external commerce system and populates the index. Computed fields are used to include the inventory data. The configuration can be found in file Sitecore.Commerce.Products.Lucene.Index.Common.config.
  Like depicted in the following table, the product index is extended with four fields:
    1. In-Stock
       Contains a list of locations where the product is in stock
    2. Out-Of-Stock
       Contains a list of locations where the product is out of stock
    3. Location
       Contains a list of locations where the product is orderable from
    4. Pre-Orderable
       Contains a boolean value indicating whether the product is pre-orderable or not

.

Product ID (not variant)	Size	Color	In-Stock	Out-Of-Stock	Location	Pre-orderable
Aw123-04	S, M, L, XL	R, B, G, O	Central Store, Store1, Store2	Store3	Central Store 1, Store 2, Store 3	Yes

The product index is defined in the file "Sitecore.Commerce.Products.Lucene.Index.Master.config". A similar configuration file is defined for the Web index.

```
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
    <contentSearch>
      <configuration type="Sitecore.ContentSearch.ContentSearchConfiguration,
Sitecore.ContentSearch">
        <indexes hint="list:AddIndex">
          <index id="commerce_products_master_index"
type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex, Sitecore.ContentSearch.LuceneProvider">
            <param desc="name">$(id)</param>
            <param desc="folder">$(id)</param>
            <!-- This initializes index property store. Id has to be set to the index id
-->
            <param desc="propertyStore" ref="contentSearch/databasePropertyStore"
param1="$(id)" />
            <configuration
ref="contentSearch/indexConfigurations/defaultLuceneIndexConfiguration"/>
            <strategies hint="list:AddStrategy">
              <!-- NOTE: order of these is controls the execution order -->
              <strategy ref="contentSearch/indexUpdateStrategies/syncMaster" />
            </strategies>
            <commitPolicyExecutor type="Sitecore.ContentSearch.CommitPolicyExecutor,
Sitecore.ContentSearch">
              <policies hint="list:AddCommitPolicy">
                <policy type="Sitecore.ContentSearch.TimeIntervalCommitPolicy,
Sitecore.ContentSearch" />
              </policies>
            </commitPolicyExecutor>
            <locations hint="list:AddCrawler">
```

```
                    <crawler type="Sitecore.Commerce.Search.ProductItemCrawler,
Sitecore.Commerce">
                        <Database>master</Database>
                        <Root>/sitecore/content/Product Repository</Root>
                    </crawler>
                </locations>
            </index>
        </indexes>
    </configuration>
  </contentSearch>
</sitecore>
</configuration>
```

Note
It is assumed that the product repository is located under the path "/sitecore/content/Product Repository".
If this is not the case, the <Root> element for the crawler must be updated to reflect the actual location

A custom crawler is used for the product index to include the items based on the product templates defined in the <includeTemplates> section of the configuration file "Sitecore.Commerce.Products.config". The list of product templates defined in this section is an exact match of the exclude templates section for the DefaultIndexConfiguration. If a custom product template is introduced, then the index configuration must be updated.

```
<includeTemplates>
        <ProductRepositoryTemplateId>{F599BF48-D6FE-40DC-9F78-
CF2D56BFB657}</ProductRepositoryTemplateId>
        <ProductTemplateId>{47D1A39E-3B4B-4428-A9F8-B446256C9581}</ProductTemplateId>
    …
</includeTemplates>
```

3.3 The Connect product data model

The rationale behind the architecture of the Connect product data model are:

- To have a product data model that fulfills the identified end-user scenarios.
- To have a single common product data model no matter which ECS is integrated with
    - for solution developers it will be the same model across different solutions.
    - for UI components developers, the components will be easier to build and maintain if the data model remains the same across external commerce systems

Product data is complex and it's important to realize that in Connect there is not a one-2-one mapping between a product and a single item in Sitecore - in the same way that product data in the ECS is not stored in a single SQL table.

Note
Data for a single product consists of multiple Sitecore items. In order not to confuse matters with CMS items, they are referred to as Entities in this document and in Connect.

In Connect, a product data model is defined, so that it's possible to:

- Provide a solid base for typical must-have e-com scenarios.

    The main reason for pulling in product data in to Sitecore is to augment it and present it on different shops and channels, e.g. media, web, mobile, print

    The product data must be normalized in order to form a good and strong foundation for building upon and to fulfill the scenarios

    Data for a product is stored as a composite structure. There is a main content item based on a template Product representing a product with only the shared generic fields like ID, Name, Description, Type etc. Below the product item is sub-tree structure with specifications, relations and resources. In addition, there are several related repositories that a product links to, like Manufacturer, product type etc.

- Avoid redundant product data

    The more redundant data, the more data needs to be synchronized, updated manually in SC and maintained

    If all data for a single product were forced to be stored in one Sitecore item, there would be a lot of redundant data stored amongst all the product items, which means far too much data is being stored and it becomes a nightmare to maintain in Sitecore.

    Instead separate repositories are used for shared data like manufacturer information, divisions and specification values and referred to by way of linking. It's similar to the way normalized product data would be stored in separate tables in a SQL databases and references with foreign keys.

- Minimize product data to synchronize

    By avoiding redundant data, the amount of data to synchronize will be minimized

Also, not all product data is needed from the ECS. Only product data needed to fulfill the scenarios described in the following sections should be synchronized and stored in Sitecore.

▪ Avoid most typical custom model implementation problems

By splitting product data into a composite product structure and place data into separate repositories the most typical implementation problems can be avoided:

- Flat model

  In a flat model a single item represents a product. Without composition of multiple items to make up product data, the model is forced to:

  ▪ Be simple, missing out on a lot of the needed information

  ▪ Have a lot of typically unused fields or use a large number of templates to cover all the different product types. Both is not best-practice.

  ▪ Have a lot of redundant data for similar products or variants

  ▪ Forcing data into custom field types or encoding data using some custom scheme

- Redundant data

  Without separate repositories for storing shared information, redundant information will is unavoidable.

- Templates overload.

  Using a separate template for each product type soon becomes a problem with a large number of different products

- Difficult to extend

  With a single item for a product and without composition of multiple items to make up product data, it becomes difficult to extend with custom data.

- Content Editing of redundant data is a real hassle

▪ Take advantage of CMS 7 features

By leveraging the new features in CMS 7, it's possible to use buckets and features like Linq based searches against custom product indexes, returning product objects through the new Hydration model. The Hydration model is similar to NHydrate and Nhibernate. For more information, see the Data Definition API Cookbook for Sitecore 7

▪ Enable e-com vendors to easily map product data

Instead of forcing product data into a single item in an artificial construct, the data is stored in a more natural way with a composite structure that will make it easier for ECS to integrate with, customize and extend.

3.3.1 Minimum product concepts

The following lists represent the minimum product concepts that need to be part of the product data model.

1. Main product data like IDs - EAN, SKU, IBSN etc - , name, Brand, Model, tags etc.

2. Multiple classification schemes aka multiple different categorization schemes

   a. Typically products are stored in categories in the e-commerce system according to type.

   b. For presentation, a different way of organizing products into categories is typically used.

3. Specifications and lookup values

   a. Specifications, lookup and default values are on three levels:

   b. Categories. A category is associated with a number of specifications that goes for all products in the same category. The specifications are the same regardless of the products and their manufacturer.

   c. Product type. Specifications that are specific to the given product type. These specifications are more closely related to the actual product and its manufacturer.

   d. Single product. Specifications that only applies to the specific product.

4. Product variant significant specifications

   a. It must be possible to configure which specifications that makes up the difference between all variants of a product, e.g. what specifications make up the variants.

5. Related products

   a. There are different types of relations between products:
      i. Variants of the same product
      ii. Accessories
      iii. Cross-sell, up-sell etc

6. Resources

   a. images and files

7. Manufacturers

   a. Manufacturer information

8. Divisions

   a. Divisions is a way to model an organization of divisions into a hierarchical structure, which are used to tag products, so that are marked part of those division.
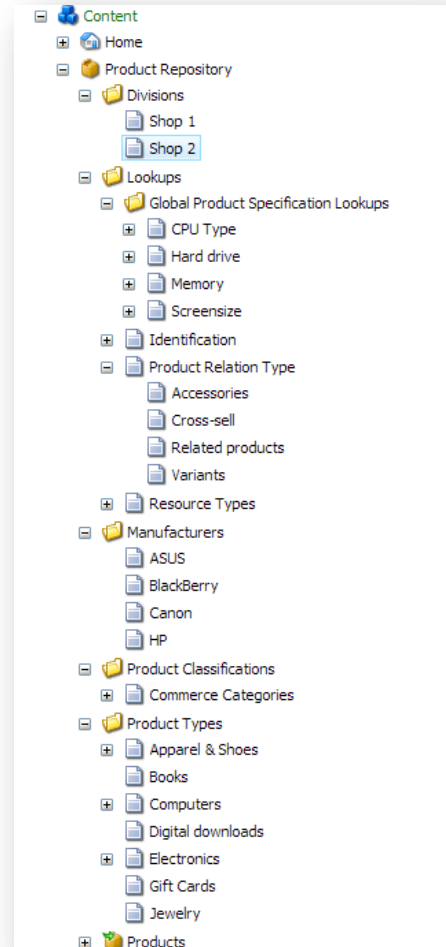
3.4   Item templates and structure

The following sections describe the templates making up the Connect product data model and the structure in which they are being used.

3.4.1   Item Templates used in the Product Data Model

In naming the templates for the product data model we have used the convention described in the following section

Rule of Thumb and Naming Conventions

For each concept in the model, product, specification, type etc. there's an entity template and a typically a corresponding folder template, e.g. each type of entity is kept in a folder. The naming convention is that the folder template name is in plural and the entity template name is in singular form. Example: Specification represents a specification entity and Specifications represent the folder in which the specifications are stored.

The product data model consists of data that are stored in separate repositories and data that is specific for a given product. For settings that relates to the specific product, the corresponding template name is prefixed with the word "Product", for example the generic specification template is called specification, whereas the product specific specification template is called ProductSpecification

Item templates

The item templates used in the product data model are displayed on the following screenshot.
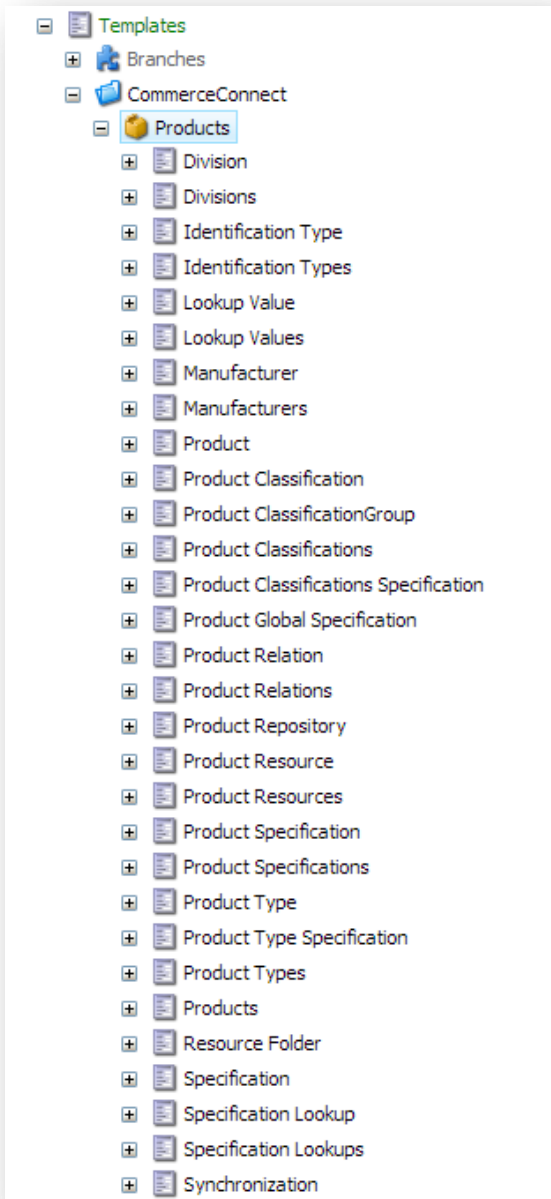
For every entity type, there's usually a folder template that holds the individual items. The following table describes the individual entity templates and list the corresponding folder template.

Entity template name	Folder template name	Description
Division	Divisions	A division represents a business unit and is used to tag products and thereby marking them part of the division. Filtering products by division will return the products that are associated with the division. Divisions are stored in an individual repository in

sitecore®

		a hierarchical manner. A division can have multiple sub-divisions. The Template Divisions is used as repository folder for the hierarchy of divisions stored within.
Identification Type	Identification Types	Multiple identifiers can be associated with any given product. Identification Type is an enumeration of different types of identification encoding schemes, e.g. a product has both an EAN and SKU number besides the internally used product code. Identification Types is used as repository folder for the identification types stored within.
Lookup Value	Lookup Values	A Lookup Value represents a key and value pair with the item name being the key and value stored in both short and long description. Each set of lookup values are stored in its own Lookup Values folder. Examples are Product Relation Types and Resource Types. Lookup Values is the folder template for lookup values and it has a single description field
Manufacturer	Manufacturers	The Manufacturer template is used to store the most essential information about a manufacturer, e.g. Name, description, website URL and Product URL macro. The template Manufacturers is used as a repository folder containing the manufacturers. The folder is configured as a bucket
Product	Product Repository	A product item represents the core data of a product and a point of reference to all related repositories: Manufacturers, Divisions, Types, Classifications Products are stored in a bucket and consists of multiple sub-items: Relations, Resources and Specifications
---	Product Artifacts	Product Artifacts is the folder template grouping together miscellaneous repositories relating to products like Manufacturers, Classifications, Divisions, Types and global lookup values
Product Classification Group	Product Classifications	A product classification Group represent a classification scheme. A lot of standards as well as custom classification schemes exists in the world today. Multiple different classification schemes and categorizations might be used concurrently in the

|		product data model, e.g. products could use both the categorization used in the external commerce system as well as <u>UNSPSC</u> classification scheme. UNSPSC being The United Nations Standard Products and Services Code, which is a hierarchical convention that is used to classify all products and services. Classifying products and services with a common coding scheme facilitates commerce between buyers and sellers and is becoming mandatory in the new era of electronic commerce A Product Classification Group contains a hierarchical structure of items based on template Product Classification. Product Classification Groups are stored in a list beneath a root folder based on template Product Classifications.
Product Classification	Product Classification Group	A Product Classification represent a category within one classification scheme (Product Classification Group) Multiple different classification schemes and categorizations might be used concurrently in the product data model. For further information see description for Product Classification Group. Product Classifications are structured in a hierarchical manner beneath a Product Classification Group.
Product Relation	Product Relations	A Product Relation represents a relation between the given product and other products in the repository
Product Resource	Product Resources	A product resource represents a media entity, e.g. a file (brochure), an image (Main image or alternate images). Resources are not always stored in Sitecore Media Library and can be represented by a URI. Resources are stored in its own folder based on template Product Resources under the product item
Product Specification	Product Specifications	A product specification holds the specification key and value or a reference to a value when based on a fixed set key-value pair table
Product Type	Product Types	Products are based on a type and inherits all the properties of the type. A product can only be of a single type.

		Types are stored in its own folder based on template Product Types organized in a hierarchical structure. Sub-types inherit properties of its ancestors. A type can contain definitions of Specifications, Specifications Default Values and Specification Options to limit the set of values from fixed set key-value pair tables for a given type.
Specification Lookup	Specifications	A Specification Lookup represents the key of a specification associated with a category or product type and the table of possible values.
Specification	Specifications	A Specification represents the key of a specification associated with a category or product type Specifications are stored in a Specifications folder.
Specification Option	Specification Options	A Specification Option is used to limit the possible values for a given product type. Specification Options are used in connection with Specification Lookups.
Specifications Default Values	Product Type	A folder for holding default values for specifications related to a type. The folder has no fields.

Branch templates

The product data model consists of the branch templates described in the following table

Branch template name	Description
Product	The branch template contains a composite tree structure that represents a product with default subfolders for related products, resources and specifications. The template is associated with the bucket that makes up the main product repository.
Product Repository	The branch installs the complete product repository including all the repositories related to products. Repositories like Divisions, Manufacturers, Product Types, Classifications, and lookups

The expanded branch templates are displayed on the following screenshot.
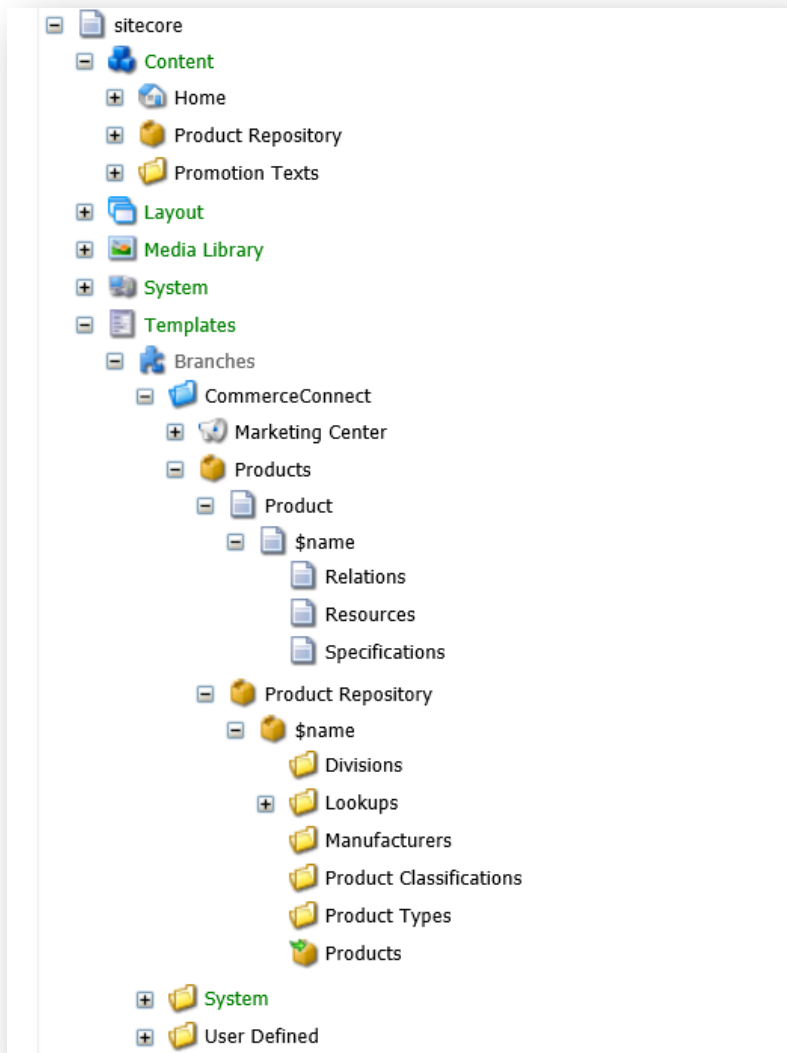
3.4.2   Main product data in one product repository bucket

In order to store a large number of products, in a single product repository, a bucket is used. One product consists of a main product item and a sub-tree of items containing values specific to the given product and references to other repositories.
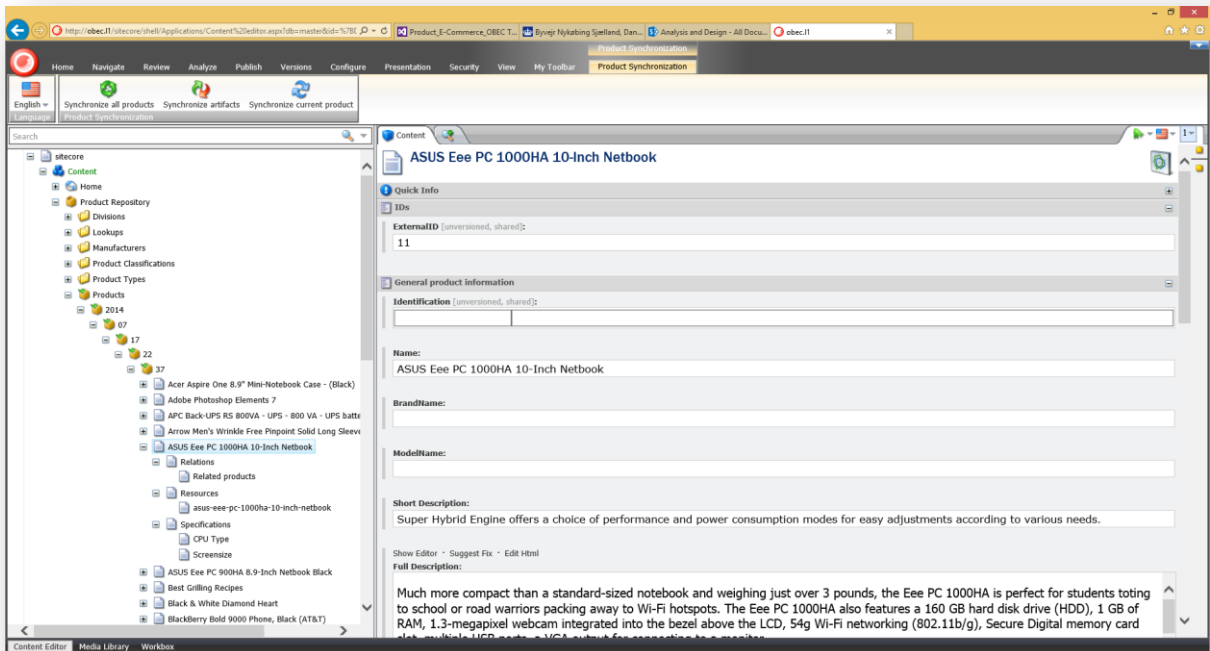
The product template only contains the most essential information that applies to all products.

Product Variants

Generally a variant is considered a product in Connect, so there's no distinction between a product and a variant of a product. They are stored in the same way.

To save space and avoid duplication of data, the concept of a product family can be used. In a product family there is one master product, which has all the default product data stored. The related product variants refer to the master product and only contains values that differ from the master product.

A Product type can be regarded as the master product as it can hold default specifications etc. Product variants must be assigned a product type and only the specifications that differ, needs to be set for the variant.

3.4.3   Product relations, resources and specifications

Each product has a composite structure organized in a sub-tree for storing relations, resources and specifications that apply to the particular product; see screenshot.
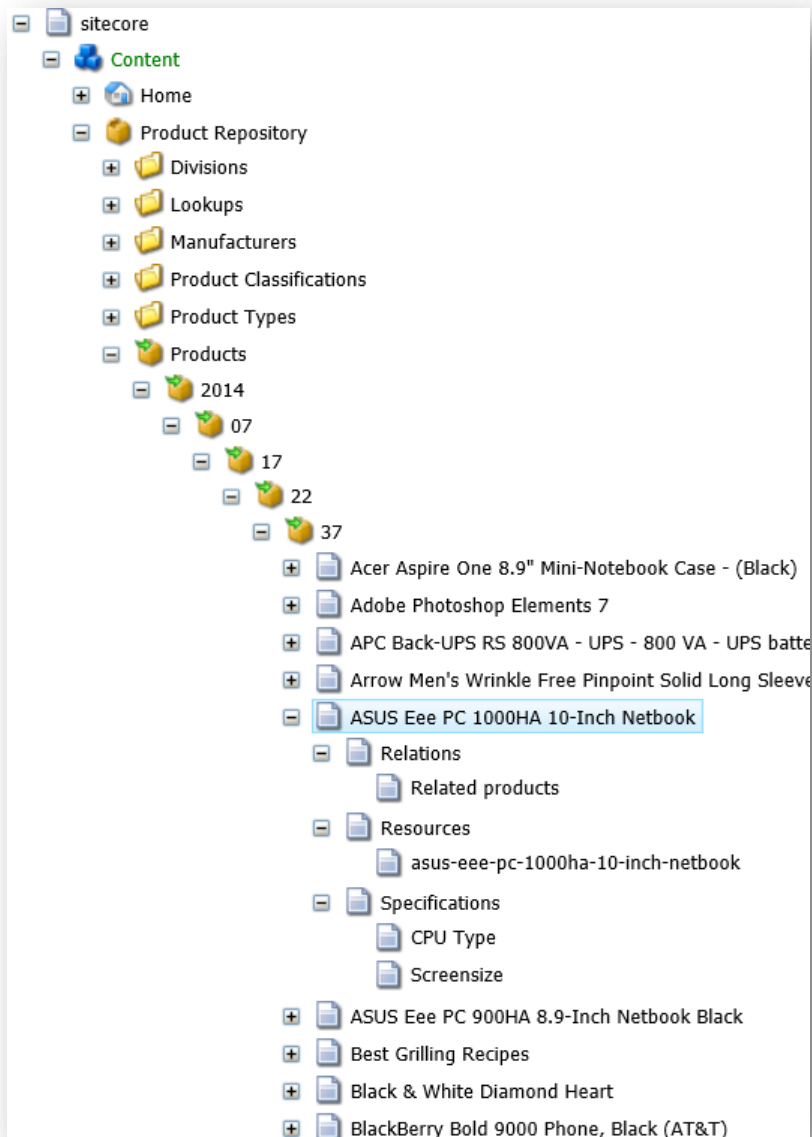
3.4.4 Specifications

Specifications for products are stored in several different places in the product data model:

1. Global specifications
   Specifications that are global across the entire product repository are stored in the global specifications folder under /Product Repository/Lookups/Global Product Specification Lookups (relative to the product repository).
   These specifications are typically stored as lookup tables where a key and a set of pre-defined values are defined.

2. Classification specifications
    For each classification scheme there's typically a list of specifications associated with each category. These are the specifications that make it possible to compare all products present within a category made by different manufacturers.
    It's also the specifications normally used for navigated or facetted search

3. Type specifications
    There's a close relationship between products and its type and hence there are 3 pieces of specification information available on types:
    a. Specifications in form of keys or keys + values (lookup tables)
    b. Specification options that narrows down the lookup table options for sub-types
    c. Specification default values that contains values for specification and all products of the given type will have those values given
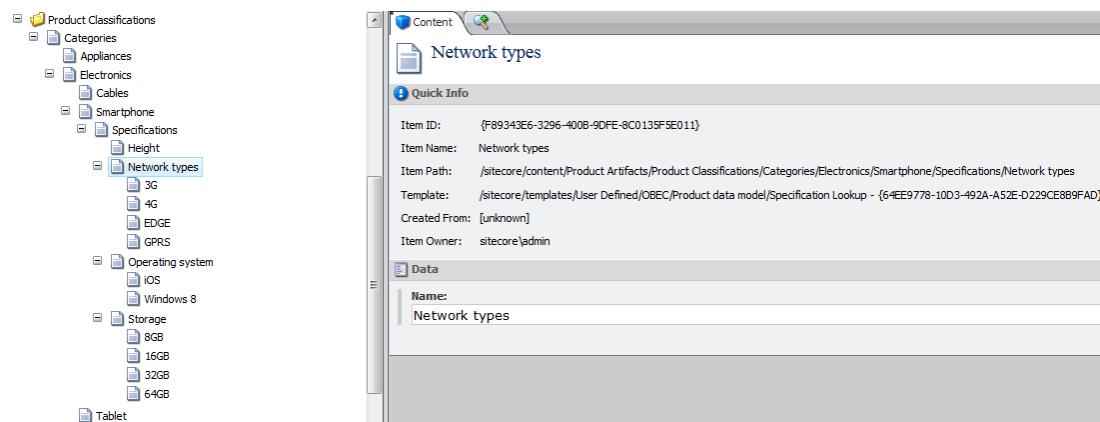
4. Product specifications
    Specifications that are unique to specific products can be stored under the product itself. Typically most specifications will be on the product type.
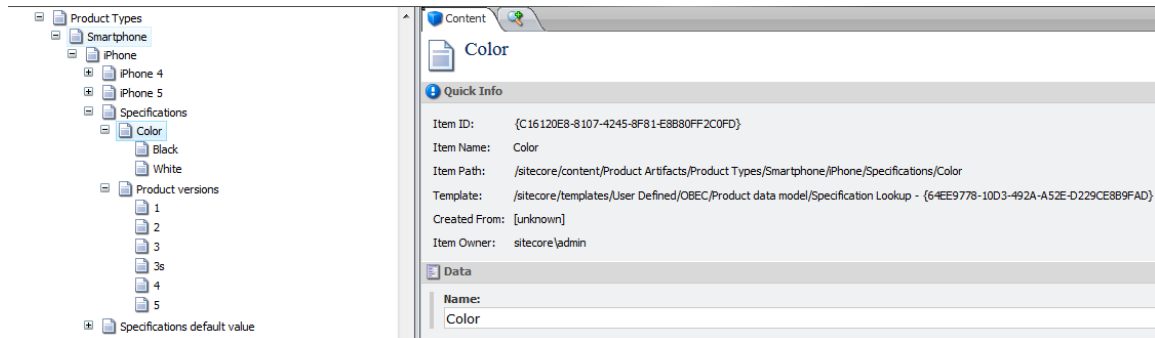
Specification

A specification represents an attribute belonging to a category or a type. A specification can be a single key or a key and a table with a set of fixed values.
For each specification there's typically a specification value. The value is stored either on the type as default value or directly on the product itself.

- Specifications can be defined on a category, meaning all products with the category assigned have the specifications and should have corresponding specification values stored. On the screenshot below the specifications for a category /Electronics/Smartphone is shown with the specifications Height, Network types, Operating System and Storage. 3 of which also represents tables of fixed lookup values.

- Specifications can be defined on the type, meaning all products of that type have the specifications as attributes and should have corresponding values stored.
  On the screenshot below the specifications for type /Smartphone/iPhone is shown with the specifications Color and Product Version, both of which also represents tables of fixed lookup values.

Specification values

Specification values represent values for a specification and can be stored in the following two places.

- Specification values on product
  Specifications are stored beneath the product item in a folder named Specifications. The folder is the repository for key-value pairs based on template Product Specification

- Specifications on product type
  Specifications are stored beneath a product type item in a folder named Specifications Default Value. The folder is the repository for key-value pairs based on template Product Specification

3.5   The Object Domain Model

The object diagram below visualizes the object domain model. For more information, see the Sitecore Commerce Connect Developer Guide.

There is a one to one correspondence between the product item templates and the objects.
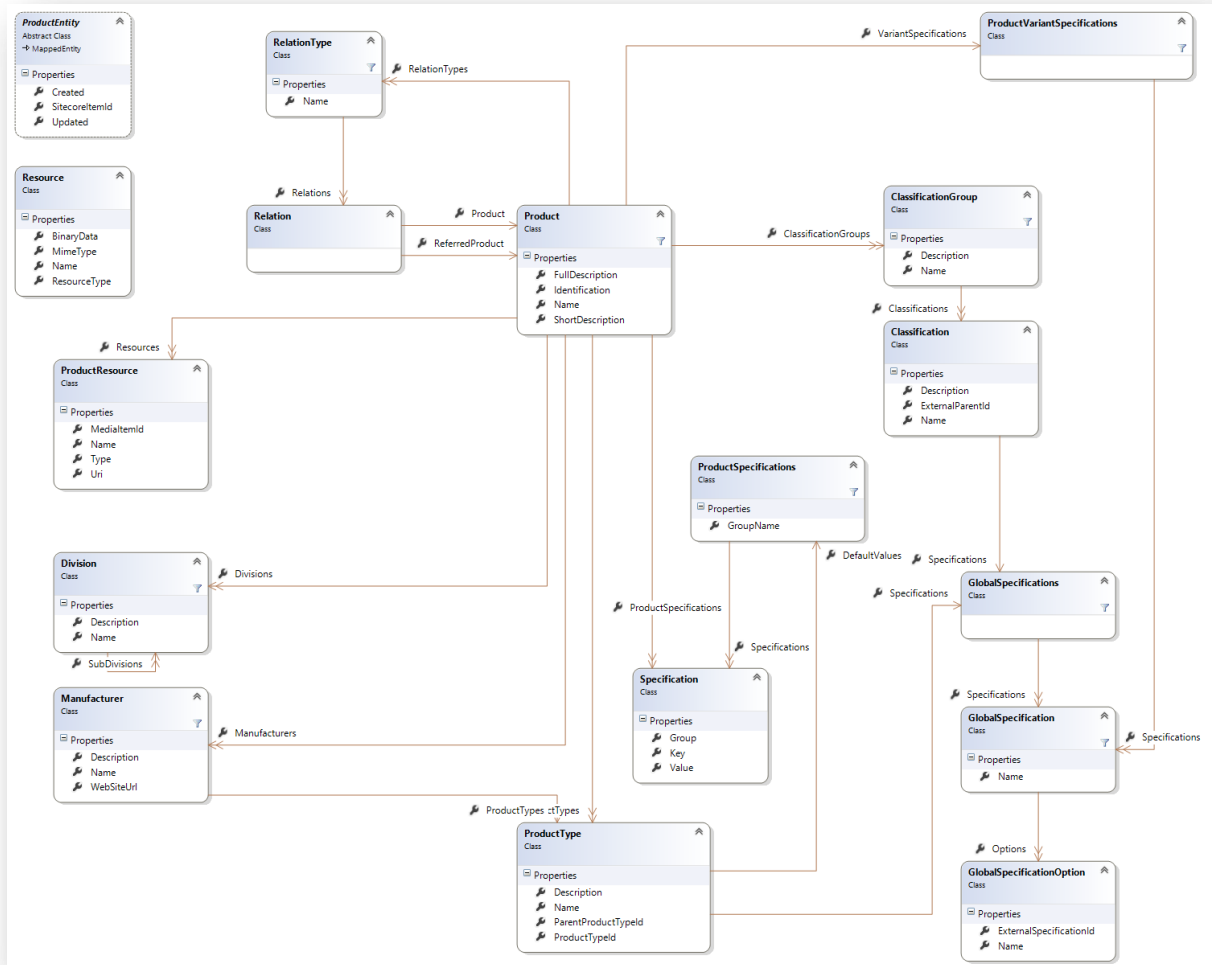
3.6 How to Implement a Custom Product Entity

When deciding whether to add a field to the main product template and corresponding object or add it to a sub-item, you should ask the simple question:

Does the field apply to all products or not?

If not, then it doesn't belong on the product template and corresponding product class.

The following steps are needed in order to extend the main product entity. The same procedure must be followed for other product entities.

- Create a custom template that inherits from the default Product template (/sitecore/templates/CommerceConnect/Products/Product) and extends it with further fields
- Create a custom Product class, that inherits from the default `Sitecore.Commerce.Entities.Products.Product` class and extends it with further properties
- Create a custom ProductRepository class, which inherits from the default `Sitecore.Commerce.Data.Products.ProductRepository`.
  Override the following two methods to save and load the extended properties. Make sure to also call the base implementation of these methods:
    - `protected override void UpdateEntityItem(Item entityItem, Product entity)`
    - `protected override void PopulateEntity(Item entityItem, Product entity)`
- Update the Sitecore.Commerce.Product.config file:
    - Update the ProductRepository element:
        - by replacing the value of the attribute type with the full type name of the custom product repository class, see snippet below with type value in italic.
        - by replacing the template ID set in the sub-element <template>. See snippet below with template value in italic

```
        <productRepository type="Sitecore.Commerce.Data.Products.ProductRepository,
Sitecore.Commerce" singleInstance="true">
            <template>{47D1A39E-3B4B-4428-A9F8-B446256C9581}</template>
```

    - Update the GUID of the ProductTemplateID in the IncludeTemplates section. For more information, see the section 3.2.3 The product index
    - Update the type attribute of the Product entity entry in the Commerce.Entities section of the Sitecore.CommerceProduct.config

```
<commerce.Entities>
<Product type="Sitecore.Commerce.Entities.Products.Product, Sitecore.Commerce" />
```

- Update the GUID of the ProductTemplateID in the ExncludeTemplates section of the Sitecore.Commerce.Products.LuceneDefaultIndexConfiguration.config file. For more information, see the section 3.2.3 The default index.
- In case 2-way synchronizing is used, then:
    - Create a custom ResolveProductChanges, that inherits from the default `Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ResolveProductChanges` class. For more information, see section

New properties which refer to items in related repositories, like Manufacturers, should load a collection of the given type and populate the values. Some internal helper methods, like PopulateEntityFieldCollections, are provided. Below is an example of how the Manufacturers collection is populated while loading the product entity.

```
        entity.Manufacturers = this.PopulateEntityFieldCollections(
```

```
                entityItem,
                "Manufacturer",
                typeof(Manufacturer),
                new Dictionary<string, string> { { "ExternalId", "ExternalID" }, { "Name",
"Name" }, { "Description", "Description" } })
            .Cast<Manufacturer>().ToList();
```

The actual manufacturer type should be created by calling the Create method on the type Sitecore.Commerce.Entities.EntityFactory, but this is left out to simplify the example.

The PopulateEntityFieldCollections method has the following signature:

```
        /// <summary>
        /// Fills the entity collections.
        /// </summary>
        /// <param name="entityItem">The entity item.</param>
        /// <param name="fieldName">Name of the field.</param>
        /// <param name="collectionMemberType">Type of the collection member.</param>
        /// <param name="properties">The properties.</param>
        /// <returns>
        /// The collection of the product entities.
        /// </returns>
        private IEnumerable<ProductEntity> PopulateEntityFieldCollections(Item entityItem,
string fieldName, Type collectionMemberType, IDictionary<string, string> properties)
```

3.7 How to Create a Custom ResolveChangesProcessor

For 2-way synchronization to work, a processor must be present, which compares the two entities originating from Sitecore and the external system respectively.

A base processor `Sitecore.Commerce.Pipelines.Products.ResolveChangesProcessor` does most of the work and resolves the configured SynchronizationStrategy to use for comparison.

The responsibility of the processor is to read two product entities from Sitecore and the external system respectively, compare and resolve the changes and return the resulting entity along with an indication of where the result must be saved.

Note
The current ResolveChangesProcessor implementation only saves one collection of resulting entities to be saved, which means the same instances will be saved to both Sitecore and the external system in case the direction is set to both. A custom version could save different versions to two separate collections to be saved in the two systems respectively.

Note
The two processors responsible for reading the product entities from Sitecore and the external system respectively, must write the result to two distinct and different pipeline arguments, so that they do not interfere.

To create a custom processor for a given entity type the following steps are needed:

1. Create a new class which inherites from ResolveChangesProcessor
   Leave the constructor empty, but make sure to call the base constructor
2. Override the GetSitecoreEntities method. The method must read the stored Sitecore entities and return an enumerable collection of objects of the given type. Naming convention for the PipelineArgs collection is to prefix the type name with the word "Sitecore" as the key. Example: "SitecoreManufacturers"
3. Override the GetExternalCommerceSystemEntities method. The method must read the stored external entities and return an enumerable collection of objects of the given type. Naming convention for the PipelineArgs is to simply use the type name as the key. Example: "Manufacturers"
4. Override the SaveEntities method. The method must save the resulting entities to the PipelineArgs collection. Naming convention is to use the type name as the key. Example: "Manufacturers"

The implementation of the ResolveManufacturerChanges is shown in the code snippet below.

```
public class ResolveManufacturersChanges : ResolveChangesProcessor
  {
    /// <summary>
    /// Initializes an instance of the <see cref="ResolveManufacturersChanges" /> class.
    /// </summary>
    /// <param name="synchronizationStrategy">The synchronization strategy.</param>
    public ResolveManufacturersChanges([NotNull] ISynchronizationStrategy
synchronizationStrategy) : base(synchronizationStrategy)
    {
    }

    /// <summary>
    /// Gets entities stored in Sitecore.
    /// </summary>
```

```
            /// <param name="args">The arguments.</param>
            /// <returns>Sitecore entities.</returns>
            protected override IEnumerable<ProductEntity> GetSitecoreEntities(ServicePipelineArgs
args)
            {
              return args.Request.Properties["SitecoreManufacturers"] as
IEnumerable<ProductEntity> ?? Enumerable.Empty<Manufacturer>();
            }

            /// <summary>
            /// Gets entities stored in external commerce system.
            /// </summary>
            /// <param name="args">The arguments.</param>
            /// <returns>External commerce system entities.</returns>
            protected override IEnumerable<ProductEntity>
GetExternalCommerceSystemEntities(ServicePipelineArgs args)
            {
              return args.Request.Properties["Manufacturers"] as IEnumerable<ProductEntity> ??
Enumerable.Empty<Manufacturer>();
            }

            /// <summary>
            /// Saves the entities to the arguments.
            /// </summary>
            /// <param name="args">The arguments.</param>
            /// <param name="productEntities">The product entities.</param>
            protected override void SaveEntities(ServicePipelineArgs args,
IEnumerable<ProductEntity> productEntities)
            {
              args.Request.Properties["Manufacturers"] = productEntities.Cast<Manufacturer>();
            }
          }
```

3.8 How to Create a Custom Synchronization Strategy

In order to use a custom synchronization strategy it is necessary to perform the following two steps:

1. Create new custom strategy class and implement ISynchronizationStrategy interface.

This interface contains one Resolve method that receives direction of synchronization and base product entity from external system and Sitecore.

Resolve method should decide which system to update (ECS or Sitecore) and return the result.

```csharp
namespace Sitecore.Commerce.Products
{
  using Sitecore.Commerce.Entities.Products;

  /// <summary>
  /// The SynchronizationStrategy interface.
  /// </summary>
  public interface ISynchronizationStrategy
  {
    /// <summary>
    /// Resolves the specified direction.
    /// </summary>
    /// <param name="direction">The direction.</param>
    /// <param name="externalSystemEntity">The external system entity.</param>
    /// <param name="sitecoreEntity">The entity from content management system.</param>
    /// <returns>The place, where we decide if entity is updated.</returns>
    UpdateIn Resolve(Direction direction, ProductEntity externalSystemEntity,
ProductEntity sitecoreEntity);
  }
}
```

2. Register custom synchronization strategy class in Sitecore.Commerce.Products.config.

To do this, change type attribute value of synchronizationStrategy element to custom synchronization strategy type.

```xml
<synchronizationStrategy
type="Sitecore.Commerce.Products.DateTimeSynchronizationStrategy, Sitecore.Commerce"
singleInstance="true" />
```

The default DateTime based strategy coming with Connect is the simplest possible strategy:

```
        /// <summary>
        /// The synchronization strategy based on updated date of entity in external system and
content management system.
        /// </summary>
        public class DateTimeSynchronizationStrategy : ISynchronizationStrategy
        {
          /// <summary>
          /// Resolves the specified direction.
          /// </summary>
          /// <param name="direction">The direction.</param>
          /// <param name="externalSystemEntity">The external system entity.</param>
          /// <param name="sitecoreEntity">The entity from content management system.</param>
          /// <returns>
          /// The place, where we decide if entity is updated.
          /// </returns>
          public UpdateIn Resolve(Direction direction, ProductEntity externalSystemEntity,
ProductEntity sitecoreEntity)
          {
            if (string.IsNullOrEmpty(sitecoreEntity.ExternalId) && (direction == Direction.Both
|| direction == Direction.Inbound))
            {
              return UpdateIn.Sitecore;
            }

            if (externalSystemEntity.Updated == sitecoreEntity.Updated)
            {
              return UpdateIn.None;
            }

            if (externalSystemEntity.Updated < sitecoreEntity.Updated)
            {
              if (direction == Direction.Both || direction == Direction.Outbound)
              {
                return UpdateIn.ExternalCommerceSystem;
              }
            }
            else
            {
              if (direction == Direction.Both || direction == Direction.Inbound)
              {
                return UpdateIn.Sitecore;
              }
            }

            return UpdateIn.None;
          }
        }
```

3.9 How to Implement a Custom ID Generator

The responsibility of the ID generator is to take a unique external ID in form of a string and return a unique ID which can be used as an item ID, e.g. GUID. The default implementation is based on the MD5 hash algorithm.

Note
Sitecore ID's must be unique and since the output from the IDGenerator is used as a Sitecore ID, it is important to always prefix the unique external ID with an arbitrary but fixed string, in order to avoid collision. For example, the IDs for manufacturers in the external system repository might have overlap with the IDs for the products, even though they are unique within their own range. The input to IDGenerator must therefore be in the following format: "Manufacturer " + ManufacturerID and "Products " + ProductID respectively.

In order to use a custom ID generator, it is necessary to perform the following two steps:

1. Create new ID Generator class and implement IIdGenerator interface.
   The interface has one StringToID method that accepts two parameters:
   a. a string containing the value of the external ID of the given entity
   b. a string containing a unique prefix to avoid collision between identical values used for different entities.
   The method returns a GUID as result, which is used to assign to the corresponding item in Sitecore representing the external entity.

```
/// <summary>
/// Defines interface for id generator.
/// </summary>
public interface IIdGenerator
{
  /// <summary>
  /// String to Sitecore ID.
  /// </summary>
  /// <param name="value">The value.</param>
  /// <param name="prefix">The prefix.</param>
  /// <returns>The generated ID</returns>
  [NotNull]
  ID StringToID([NotNull] string value, [NotNull] string prefix);
}
```

2. Register custom ID Generator class in Sitecore.Commerce.Products.config.
   To do this, change type attribute value of "idGenerator" element to the custom ID Generator type.

```
<idGenerator type="Sitecore.Commerce.Data.Products.Md5IdGenerator,
Sitecore.Commerce" singleInstance="true" />
```

The default implementation is based on the MD5 hash algorithm provided in .NET. The source code is listed below:

```
/// <summary>
/// Defines default implementation of id generator.
/// </summary>
public class Md5IdGenerator : IIdGenerator
{
  /// <summary>
  /// String to Sitecore ID.
```

```
      /// </summary>
      /// <param name="value">The value.</param>
      /// <param name="prefix">The prefix.</param>
      /// <returns>The generated ID</returns>
      public ID StringToID(string value, string prefix)
      {
        Assert.ArgumentNotNull(value, "value");
        Assert.ArgumentNotNull(prefix, "prefix");

        // Create a new instance of the MD5CryptoServiceProvider object.
        var md5Hasher = MD5.Create();

        // Convert the input string to a byte array and compute the hash.
        var data = md5Hasher.ComputeHash(Encoding.Default.GetBytes(prefix + value));
        return new ID(new Guid(data));
      }
    }
```

3.10 Performance tuning

For information on steps taken to improve performance, see the Sitecore Commerce Connect Overview document.

- Immediate or delayed bucket synchronization. When a new item is created in a bucket, it is immediately placed in the root folder. In order to move it into the right place, the bucket needs to be synchronized and there is a couple of ways it can be done. For more information on how to enable delayed bucketing, see section 3.11 Delayed Bucket Synchronization.

  - When synchronizing a single product, it can immediately be moved to the right place in the bucket. Calling `SynchronizeProduct` as a single operation is doing this by calling `BucketManager.MoveItemIntoBucket(entityItem, root);`

  - Doing a bulk synchronization by calling `SynchronizeProducts` or `SynchronizeProductList` can cause the creation of many new items that needs to be synchronized. When doing bulk synchronizing, it is faster to delay the bucket synchronization until all new product items have been processed. To further reduce the time spent synchronizing the products bucket, a temporary bucket is used for new product items. The temporary bucket is synchronized after all products have been processed and the bucket content is moved to the main bucket. That will eliminate the time spent touching all existing items in the bucket, which could be significant, e.g. adding a 1.000 new product items to a bucket with 1.000.000 product items, will touch 1.001.000 items to make sure they have not changed.

- Multi-threaded synchronization. A single thread is by default spawned to synchronize products, manufacturers, types, resources, divisions, and specifications in parallel. The threads are spawned for each repository being synchronized. The number of threads to use can be configured in the `Sitecore.Commerce.Products.config` file. The default is 1

  ```
  <setting name="ProductSynchronization.NumberOfThreads" value="8" />
  ```

Note
Due to issues in Sitecore CMS, using more than 1 thread can result in a SQL server deadlock situation, which is why the default configuration only specifies 1 thread.

- Disabling of indexing, events and caching. Triggering of item events as well as indexing is disabled while synchronizing in order not to waste resources on firing events or starting indexing before synchronization is done. Indexing is turned on after synchronization has finished. For more information, see section 3.2.3 Indexing.

- Reading product data ones and process it in multiple pipelines will reduce the number of calls between Sitecore and the external systems. All product entities in Connect are synchronized using its own pipeline, which naturally lends itself to reading the data from the external system in the individual pipelines. In that case, synchronizing a single product can amount to a fair amount of calls between the systems and each call takes time and resources. The design does not prevent product data to be read ones initially and passed on to the individual sub-pipelines for processing, reducing the number of calls between the systems.

- Resources can be located externally. Resources in Sitecore are stored as media items in the media library. Media items are binary blobs and can be rather large and time consuming to import into Sitecore and for that reason, resources can either be imported into Sitecore media library or simply referred to externally. If resources are imported, they are stored in a bucketed folder called Products under the media library. If not imported, they can be referred to by a URI

stored on the resource reference item. The default implementation of Connect supports both scenarios.

3.11 Delayed Bucket Synchronization

There's several ways to have products synchronized into the Bucket where the main product data is stored. It can be done:

- Immediately, which is the default and fastest approach.
- At the end of the entire Commerce Connect product synchronization. This approach is also implemented in Commerce Connect and can be activated by enabling the config file `Sitecore.Commerce.Products.DelayedSyncProductRepository.config.disabled` This is done by removing the postfix ".disabled"