The Commerce Connect integration guide

sitecore®
Own the experience™

Table of Contents

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 two components:

- Connect Core Framework contains the abstract service layers and performs engagement activities—such as tracking page events and goals, acting on conditional rendering rules, and following up on engagement automation plans, actions, and conditions.
- Connect Connectors hook into pipelines and integrate with external commerce systems. These connectors are not a part of the core framework, but are needed because Connect cannot run as a standalone system. Developers can use Connect Connectors to integrate Sitecore XP with one or more commerce systems.

The Commerce Connect integration API incorporates customer tracking by triggering goals and page events and uses engagement automation plans for following up on customer interaction. In addition, Commerce Connect comes with e-commerce specific rendering rule conditions for acting on customer interactions, cart content, and orders placed, and so on.

For a general introduction and overview of the components in Connect, see the Commerce Connect Components topics.

> **NOTE**
> If you are a developer who creates 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 Developer's Guide.

2. Integrating with Commerce Connect

You can use Connect Connectors to link to pipelines and integrate with external commerce systems. The typical connector consists of a number of custom processors inserted in Commerce Connect defined pipelines, which work with the ECS, either directly or through a web service.

2.1. Integration service layers

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.

2.1.1. A customizable domain model

Each service layer contains a set of entity classes that reflect the domain model. Domain model objects are used when operating with APIs. APIs accept the objects as part of the input parameters and return objects.

The domain model has been kept to a minimum because all vendors, to some degree, store different information and one model is not for everyone. Nonetheless, the domain models include enough information, domain objects, and parameters to cover common scenarios that are used in all shops.

It is 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 acceptable to leave them as is.

With product synchronization, in addition to the domain model objects, there is also a corresponding item domain model matching the entity classes. The entities can be customized by changing the configuration section.

The domain models are customizable so that:

- All domain model objects can be inherited and extended with custom properties.
- All nested objects can be inherited and extended with custom properties.
- All service methods keep the existing defined signature, even when used with customized domain objects.

We recommend that you create an abstraction layer on top of the ECS that extracts and manages the information to be exchanged with Commerce Connect. This approach is similar to the Bridge design pattern used in computer science and makes it easier to continuously manage the integration as both Commerce Connect and the ECS evolve. It also makes it easier to exchange information if the Commerce Connect domain model and the ECS abstraction layer objects have corresponding and similar object signatures.

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

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 need to be read from the ECS so that it can be presented to the visitor on the user interface. Shopping cart content, customer account information, and shipping information must be parsed over to the ECS so that an order can be created.

Default service layers can be customized or substituted.

Each method on the service layers accepts a single Request object and returns a single Result object. You can customize both the Request and Result objects individually for each method for maximum flexibility. The service layer interface remains the same, even when the domain model objects are customers, in addition to the parameters going in and the returned results.

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

For example:

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

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

The generic version of the same method:

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

Service methods

If possible, use the following naming conventions for all methods on a service provider:

- CreateEntityName (for example, CreateCart)
- GetEntityName (for example, GetCart)
- DeleteEntityName (for example, DeleteCart)
- UpdateEntityName (for example, UpdateCart)

If possible, use the following naming conventions for all methods that manipulate related items on an entity:

- AddRelatedEntityName (for example, AddLineItem)
- RemoveRelatedEntityName (for example, RemoveLineItem)
- UpdateRelatedEntityName (for example, UpdateLineItem)

Request parameters

A service method takes a single request object as a parameter and this request object must inherit from a `ServiceProviderRequest`. When you use a single request object instead of multiple parameters, 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 exposes 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 following example:

```
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, instead, you can use the property bag on the request to pass down any properties you want:

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

Result objects

Result objects generally mirror request objects, with the difference that they inherit from `ServiceProviderResult` and have a collection for system messages. If possible, always return system messages instead of exceptions.

You can set messages on a result using the following pattern:

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

2.1.3. Service method pipelines

Each service method launches a pipeline with the same name. As part of the initial pipeline being executed, one or more additional or shared pipelines can be called and executed. For example, `SaveCart` or `SynchronizeProductArtifacts`.

In Sitecore, the default pipeline arguments contain `Request` and `Result` properties, which have Properties of type `dictionary`, and 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 in 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 to 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
SynchronizeClassifications	SitecoreClassificationGroups	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: ReadExternalCommerceSystemClassifications Reader processors: • ResolveClassificationsChanges • SaveProductClassificationsToSitecore
SynchronizeClassificationsSpecifications	ProductClassificationGroups	Holds the product classification groups to be synchronized. Writer processor: ReadExternalCommerceSystemClassificationsSpecifications Reader processor: SaveClassificationsSpecificationsToSitecore
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: ResolveManufacturersChanges

Pipeline	Property name	Custom data description

		`ReadExternalCommerceSystemManufacturers`
		
		Reader processor:
		
		`ResolveDivisionsChanges`

`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:

Pipeline	Property name	Custom data description
		`ResolveTypesChanges`
	`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`

`SynchronizeProducts`	`ExternalCommerceSystemProductIds`	Holds the product IDs in the external commerce systems to be synchronized. Writer processor:

Pipeline	Property name	Custom data description

		`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`

`SynchronizeTypeSpecifications`	`SpecificationCollection`	Holds the specification collection to be synchronized.
		
		Writer processor:
		
		`ReadSitecoreTypeSpecifications`
		
		Reader processor:
		
		`SaveTypeSpecificationsToExternalCommerceSystem`

2.1.4. Passing data between pipeline components

All components in a pipeline operate independently without knowledge of what other components have done. However, there are occasions where information must be passed between components to avoid repeating the same action over and over.

In these situations, use the `RequestContext` property of the base request object. This is 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 that is to 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 is used to indicate if the initial request passed down to the pipeline was executed successfully. We recommend that in addition to setting the *Success* property to false, you add a failure message to the `SystemMessages` collection.

2.1.7. Configuration

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

- Cart Service Layer - `/App_Config/Include/Sitecore.Commerce.Carts.Config`
- Catalogs Service Layer - `/App_Config/Include/Sitecore.Commerce.Catalogs.Config`
- Customers and Users Service Layer - `/App_Config/Include/Sitecore.Commerce.Customers.Config`
- Gift Cards - `/App_Config/Include/Sitecore.Commerce.GiftCards.Config`
- Globalization - `/App_Config/Include/Sitecore.Commerce.Globalization.Config`
- Inventory Service Layer - `/App_Config/Include/Sitecore.Commerce.Inventory.Config`
- Loyalty Programs - `/App_Config/Include/Sitecore.Commerce.LoyaltyPrograms.Config`
- Orders Service Layer - `/App_Config/Include/Sitecore.Commerce.Orders.Config`
- Payments Service Layer - `/App_Config/Include/Sitecore.Commerce.Payments.Config`
- Pricing Service Layer - `/App_Config/Include/Sitecore.Commerce.Prices.Config`
- Product Synchronization Service Layer - `/App_Config/Include/Sitecore.Commerce.Products.Config`
- Shipping Service Layer - `/App_Config/Include/Sitecore.Commerce.Shipping.Config`
- Wish Lists Service Layer - `/App_Config/Include/Sitecore.Commerce.WishLists.Config`

The `Sitecore.Commerce.Products.DelayedSyncProductRepository.config.disable` file is an additional configuration file that can be enabled if the synchronization of products into the Bucket occurs at the end of Commerce Connect synchronization instead of immediately.

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

- `EntityFactory`
- `EaPlanProvider`
- `Contact Factory`
- `ItemClassificationService`
- `CommereContext`

2.1.8. Use a custom factory

You can customize all entities used in Commerce Connect 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 you prefer another factory, Dependency Injection (DI), or Inversion of Control (IOC) implementation, the default implementation can be replaced.

To use a custom factory:

1. Create a new custom factory class and implement the `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 the custom EntityFactory class in the `Sitecore.Commerce.config` file. To do this, change the type attribute value of the `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 looks up the actual type to instantiate in the configuration. Each service layer has its own section called `commerce.Entities`. For example, the following is the default entities for Carts:

```
<!--  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>
```

3. To use the custom entity, create a new Entity class.
4. Register the custom Entity class in the `commerce.Entities` configuration section by changing the attribute value of the entityFactory element to the custom `EntityFactory` type.

2.1.9. The EaPlanProvider class

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

2.1.10. The ContactFactory class

The `ContactFactory` 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 `ContactFactory` class and overriding the `GetContact` method.

The following is an example of how the default instance works. When you have the id of your user from the ECS, you identify the `Tracker.Current.Contact` with that id (using the `Tracker.Current.Session.Identify()`method),and from that point on this id is returned by the `ContactFactory` class. If no id is available from the external user then the id created by Sitecore Analytics is used instead.

```
public virtual string GetContact()
{
if (Tracker.Current == null)
{
if (HttpContext.Current != null &&
HttpContext.Current.User != null)
{
return HttpContext.Current.User.Identity.Name;
}

return System.Threading.Thread.CurrentPrincipal.Identity.Name;
}
var user = Tracker.Current.Contact.Identifiers.Identifier;
if (string.IsNullOrEmpty(user))
{
  user = Sitecore.Data.ID.Parse(Tracker.Current.Contact.ContactId).ToString();
}
return user;
}
```

2.1.11. The ItemClassificationService class

The `ItemClassificationService` class 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.

2.1.12. The CommerceContext class

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.

2.1.13. Service provider

Each service layer has its own interface that you can customize. These providers contain the service methods for interacting with the appropriate subsystem. All service providers must inherit from the `ServiceProvider` class and we recommend that you 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 requires a new service method, consider extending the service provider and adding the new method instead of creating a new service provider. The following lists the various subsystems and their service providers:

- 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`

2.1.14. Custom Commerce Connect cart and orders entities

If you extend any of the Commerce Server or Commerce Connect types, you can create your own equivalent pipeline processor and insert it after `RegisterDataModelExtensions`. Do not replace the `RegisterDataModelExtensions` pipeline.

You must register any entity that is synced into xDB (EAPs or Page Events) in the `MongoDbObjectMapper`. Inside the `Initialize` pipeline, there is a processor called `RegisterDataModelExtensions` that registers all of the Commerce Server extensions to Commerce Connect types.

Example of a pipeline processor that registers a custom type:

```
namespace MyNamespace.Pipelines
{
    using Sitecore.Analytics.Data.DataAccess.MongoDb;
    using Sitecore.Commerce.Connect.CommerceServer.Inventory.Models;
    using Sitecore.Commerce.Connect.CommerceServer.Orders.Models;
    using Sitecore.Pipelines;
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;

    /// <summary>
    /// Pipeline processor that registers data  model extensions with the BSON
class map.
    /// </summary>
    public class MyDataModelExtensions
    {
        ///  <summary>
        /// Processes the pipeline arguments.
       ///  </summary>
       ///
      <param name="args">The pipeline  arguments.</param>
      public virtual void Process(PipelineArgs  args)
      {

MongoDbObjectMapper.Instance.RegisterModelExtension<MyCustomType>();
      }
   }
}
```

2.2. Service layer specifics

Each service layer in Commerce Connect follows the same design pattern. The following sections describe the specific properties and configuration options unique to each service layer.

2.2.1. Cart service layer

Working with an external commerce system (ECS)

Cart integration can be done in four different ways:

- The cart is only passed to the external commerce system (ECS) 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, for example, in Commerce Connect.
  In this case, the cart related pipelines only contain the default Commerce Connect provided processors. This is very easy to set up 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.
- The cart is only saved in the ECS after each change (`OnSave`).
  This option is the default option and will minimize the number of calls to the external commerce system and improve performance. It will, however, be more difficult for the external commerce system to act upon updates made to the cart in Sitecore, such as making changes to cart lines when products are added to the cart. For example:

    - 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.
- All cart actions are forwarded to the ECS (`AddLine`, `UpdateCart`, and so on). This option provides the most flexibility for advanced scenarios as explained previously, but it also makes more calls to the external commerce system, decreasing performance.
- Cart data is only persisted in the ECS. In this case, you must remove the Commerce Connect specific processors from the pipelines mentioned in parentheses `LoadCartFromEAState` (LoadCart), `SaveCartToEAState` (SaveCart), `FindCartInEAState` & `RunResumeCart` (CreateOrResumeCart), `DeleteCartFromEaState` (deleteCart), and `BuildQuery` + `ExecuteQuery` (GetCarts).

Configuration

You can find all configuration for the cart subsystem 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.

To extend any of the default entities, you create a new class that inherits from the appropriate type, and then patch the appropriate entity under <commerce.Entities> in the `Sitecore.Commerce.Carts.config` file.

> **NOTE**
> For more information, see the Developer's 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 external commerce system (ECS) or implement functionality that the destination ECS might now support.

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

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` file with the new full class name.

Commerce Connect ships with two sample repositories called `EaStateSqlBasedCartRepository` and `EaStateCartRepository`, these are only sample repositories and you must not use them in a production scenario.

Abandoned Cart Marketing Automation campaign

The plan is provided as a branch template. Use only one instance per shop. You can customize the default plan with more states as needed.

To make the plan work, Sitecore Commerce provides the following two specific conditions and one action:

- Condition: Has Empty Cart?
  The condition retrieves the cart of the current visitor and checks if it is empty or not, for example, if there are any cart lines in it. By default, this only works with one cart per user.
- Condition: Has Provided E-mail?
  The condition retrieves an email message 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 the Customer Engagement Platform (CEP), to send out the notification email message informing the user about the abandoned cart and encouraging them to return and complete the purchase.

2.2.2. Catalog

Configuration

You can find all configuration for the catalog subsystem in the `Sitecore.Commerce.Catalog.config` file. Here you will find all details for the entities, pipelines, and repositories used by the catalog subsystem.

Entities

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

You can extend the following entity defined in the Connect system for catalog:

- `SortDirection` - an extendable enum type to represent sorting directions.

To extend any default entity, you create a new class that inherits from the relevant type, and then patch the appropriate entity under <commerce.Entities> in the `Sitecore.Commerce.Catalog.config` file.

Pipelines

There are five pipelines for the Catalog service and all raise a page event based on user action:

- The `productSorting` pipeline - called when a user performs a sort on a list of products.
- The `facetApplied` pipeline - called when a user tries to facet on search results.
- The `visitedProductDetailsPage` pipeline - called when a user visits a product detail page.
- The `visitedCategoryPage` pipeline - called when a user visits a category page.
- The `searchInitiated` pipeline - called when a user performs a keyword search.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.3. Customers and users

Both users and customers are consumers of your external commerce system (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` entities represent two aspects of the customer, whereas 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.

There are many different ways to use Connect with an ECS for customers and users:

- To pass customer and user information between the ECS and Sitecore.
- To set and/or get customer information during checkout.
- To register accounts for new users.
- To authenticate (for example, login or logout registered users).
- To enter a user into an engagement automation (EA) plan when creating a user account and trigger goals when logging in.

Configuration

You can find all configuration for the customer subsystem 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.

> **NOTE**
> 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.

You can use the following five entities defined in the Connect system for customers and users to extend functionality to suit your needs:

- `CommerceCustomer` - is always extended to include custom information particular to the external commerce system (ECS).
- `CommerceUser` - is always extended to include custom information particular to the external commerce system (ECS).
- `CustomerParty` - represents the type and 0-to-many relationship between the customer and a list of parties, where parties are of type Party.
- `CustomerPartyType` - indicates the type of relationship between the customer and party. The class is introduced as an extensible enum. To extend and customize the options, Connect has two party types: `AccountingParty` and `BuyerParty`.
- `Party` - a shared entity between the carts service layer and the customer and users service layer. This entity stores party information, for example, address information.
  To extend any of these default entities, you create a new class that inherits from the relevant type and then patch the relevant entity under commerce.Entities in the `Sitecore.Commerce.Customers.config` file.

Pipelines

There are numerous pipelines for customers and users that provide most basic functionality.

For example:

- `CreateCustomer` and `CreateUser` pipelines - used to create customers and users.
- `UpdateCustomer` and `UpdateUser` pipelines - used to update customers and users.
- `DeleteCustomer` and `DeleteUser` pipelines - used to delete customers and users.
- `AddCustomers` and `AddUsers` pipelines - used to associate customers to users.
- `AddParties` pipelines - used to add party information.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.4. Globalization

Configuration

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

Entities

The globalization service does not have any custom entities.

Pipeline

The globalization service has a single pipeline called `cultureChosen` that can be called when a site visitor changes the active culture on the site.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.5. Orders service layer

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

Configuration

You can find all configuration for the order subsystem 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 the cart, with the exception of the `Order` and `OrderHeader` entities.

The Order entity simply extends Cart and adds an `OrderId` property, and the `OrderHeader` entity extends from CartBase that simple serves as a class with some basic information about an order.

To extend any of the default entities, you create a new class that inherits from the appropriate type, and then patch the appropriate entity under `<commerce.Entities>` in the `Sitecore.Commerce.Orders.config` file.

Pipelines

The order layer ships with the following pipelines: `submitVisitorOrder`, `getVisitorOrder`, `getVisitorOrders`, `visitorCancelOrder`, `reorder`, `getAvailableCountries`, `getAvailableRegions`, `SynchronizeOfflineOrders`, and `orderStatusChanged`.

By default, these pipelines trigger a goal, except for the `submitVisitorOrder` pipeline, which adds the order to an Engagement Automation Plan, and the `reorder` pipeline, which calls the relevant cart pipelines to add the reordered items to the customers cart. Each of these pipelines must have a relevant processor that can communicate with an external commerce system (ECS).

> **NOTE**
> For more information, see the Developer's Guide.

New Order Placed Engagement Automation Plan

The plan is provided as a branch template and multiple instances can be created. Use 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.

2.2.6. Gift cards

Configuration

You can find all configuration for the gift card subsystem in the `Sitecore.Commerce.GiftCards.config` file. Here you will find all details for the entities, pipelines, and repositories used by the gift card system.

Entities

The default gift card entities for Commerce Connect only assume some of the basic gift card information that will be used across all commerce systems. It is expected that you will need to extend these entities.

You can use the following two entities defined in the Connect system for gift cards to extend functionality to fulfill your requirements:

- `GiftCard` - represents a gift card owned by a customer and contains information such as balance and original amount.
- `GiftCardPaymentInfo` - enables the use of a gift card as a payment method by extending the PaymentInfo class and adding a property to store the amount to charge.

To extend any of these default entities, you can create a new class that inherits from the relevant type, and then patch the relevant entity under commerce.Entities in the `Sitecore.Commerce.GiftCards.config` file.

Pipeline

There is a single pipeline in the gift card implementation that enables most basic functionality:

- The `getGiftCard` pipeline - used, for example, to retrieve a gift card object.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.7. Inventory service layer

The inventory service layer provides read-only integration with inventory / stock information from an external commerce system (ECS). However, you can extend this service layer to support read-write integration if desired.

Configuration

You can find all configuration for the inventory service layer n the `Sitecore.Commerce.Inventory.config` file. Here you will find all details for the entities, pipelines, and repositories used by the inventory system.

> **NOTE**
> We highly recommend that you do not modify this file when adding your ECS connector components to the pipelines, overriding entity definitions, and so on. 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 want to support updating stock information through the inventory system, you must extend the system with your own pipelines and service provider methods.

Pipelines

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

- **Runtime Integration**:
  - commerce.inventory.getStockInformation
  - commerce.inventory.getPreOrderableInformation
  - commerce.inventory.getBackOrderableInformation
  - commerce.inventory.getStockLocations
  - commerce.inventory.getProductStockLocations
- **Search Integration**:
  - commerce.inventory.stockStatusForIndexing
- **Event Raising**:
  - commerce.inventory.visitedProductStockStatus
  - commerce.inventory.productsAreBackInStock
  - commerce.carts.addCartLines
- **Products Back In Stock Engagement Plan**:
  - commerce.inventory.visitorSignUpForStockNotification
  - commerce.inventory.removeVisitorFromStockNotification
  - 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
- commerce.inventory.getStockLocations
- commerce.inventory.getProductStockLocations

Extending the other pipelines is optional.

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 must 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 `staticreadonly` field that represents the `Downloadable` value (that is, `public static StockStatus Downloadable = new EcsStockStatus(5, “Downloadable);)`

The InventoryProductBuilder

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

The InventoryAutomaionProvider

The `InventoryAutomationProvider` class is a helper class used by the conditions and 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` class 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. Create one instance per shop. You can customize the default plan with different or more states as is needed. Its purpose is to notify customers by email message 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 updates the automation plan state data, and removes the back-in-stock notification requests that are 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 an email message to the customers when a product they are interested is back in stock. Customize this action 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 are 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` class to access automation state data and perform notification comparisons. So, customizing the conditions and actions directly is not necessary. Instead, you can update the `InventoryAutomationProvider` class to extend any functionality needed in this automation plan.

Extend the InventoryProduct entity

The `InventoryProduct` entity is used to uniquely identify a product/stock information in the external commerce system (ECS). If the default `InventoryProduct` entity is not sufficient to identify stock information, you must extend this entity, as well as a few provider classes in the inventory system.

To extend the `InventoryProduct` entity:

1. Create an `EcsInventoryProduct` class that derives from `InventoryProduct` entity that contains the information required to identify stock information in your ECS.
2. Create an `EcsCommerceContext` class that derives from the `CommerceContextBase` class 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 the `InventoryProductBuilder`, and override all methods of the base class to properly handle your `EcsInventoryProduct`. In particular, you must use the new `EcsCommerceContext` inside `CreateInventoryProduct`() to populate the additional properties of your `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 must 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`.

2.2.8. Loyalty cards

Configuration

You can find all configuration for the loyalty card subsystem in the `Sitecore.Commerce.LoyaltyPrograms.config` file. Here you will find all details for the entities, pipelines, and repositories used by the loyalty card system.

Entities

The default loyalty card entities for Commerce Connect only assume some of the basic loyalty card information will be used across all commerce systems. It is expected that you will need to extend these entities.

You can extend the following eleven entities , which are defined in the Connect system for loyalty cards:

- `LoyaltyCard`- represents the LoyaltyCard of a user, and contains information such as card number, associated programs, and reward points.
- `LoyaltyCardPaymentInfo` - extends the `PaymentInfo` class to allow a loyalty card to be used to pay for all or part of an order.
- `LoyaltyCardTier` - describes a tier within a loyalty program, contains information such as valid from and to dates.
- `LoyaltyCardTransaction` - describes a points transaction on a loyalty card, contains information such as points awarded and the type of points awarded.
- `LoyaltyProgram` - describes a program that a loyalty card is part of; contains information such as any tiers that the program might have.
- `LoyaltyProgramStatus` - describes a program that a loyalty card is part of; contains information such as any tiers that the program might have.
- `LoyaltyProgramSummary` - contains basic information about a program such as name, description, shop name, and id.
- `LoyaltyRewardPoint` - represents meta data around points added to a card; contains information such as description, point type, and issued points.
- `LoyaltyRewardPointEntryType` - used as extensible enumeration class for describing different point types. The default types are None, Earn, and Redeem.
- `LoyaltyTier` - describes a tier within a loyalty program; contains information such as the tier id and level.
- `PointBasedLoyaltyTier` - extends the Loyalty tier class to describe a points-based tier; contains a property to indicate the amount of points required to reach the tier.

- `RewardPointType` - indicates what can or cannot be done with a collection of points. The default values are `Redeemable` and `NotRedeemable`.

To extend any of these default entities, you create a new class that inherits from the relevant type, and then patch the relevant entity under <commerce.Entities> in the `Sitecore.Commerce.LoyaltyPrograms.config` file.

Pipelines

There are seven pipelines for loyalty cards that allow most basic functionality.

For example:

- The `GetLoyaltyCards`, `GetLoyaltyCardTransactions`, and `getLoyaltyProgramStatus` pipelines - used to get loyalty card information.
- The `getLoyaltyPrograms` and `getLoyaltyProgram` pipelines - used to get program information.
- The `joinLoyaltyProgram` pipeline - used to join a customer to a loyalty program.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.9. Payments

Configuration

You can find all configuration for the payments subsystem in the `Sitecore.Commerce.Payments.config` file. Here you will find all details for the entities, pipelines, and repositories used by the payments system.

Entities

The default payment entities for Commerce Connect only assumes some of the basic payment information that will be used across all commerce systems; it is expected that you will need to extend these entities.

You can extend the following three entities that are defined in the Connect system for payments:

- `PaymentOption` - represents detailed information about a payment option (also known as a payment category); contains information such as description and `PaymentOptionType`.
- `PaymentMethod` - represents information about a payment method, where a payment method is an implementation of a payment option, for example, if the payment option is PayCard then a payment method for that would be Visa or Mastercard.
- `PaymentOptionType` - an extensible enum class used to contain the different types of payment options; the default choices are None, PayCard, PayLoyaltyCard, and PayGiftCard.

To extend any of these default entities you create a new class that inherits from the appropriate type, and then patch the appropriate entity under `commerce.Entities` in the `Sitecore.Commerce.Payments.config` file.

- `PaymentLookup` - holds information required to lookup payment prices for a collection of line items.
- `CardType` - used to represent a payment card type supported by the merchant website or payment service.

- `PurchaseLevel` - used to represent the level of card processing that will be performed during a payment service transaction.
- `TransactionType` - used to represent the type of transaction being performed with a payment service.

Pipelines

There are several pipelines for payments that allow most basic functionality.

For example:

- The `getPaymentOptions` pipeline - used to retrieve payment options.
- The `getPaymentMethods` pipeline - used to retrieve of payment methods.
- The `getPricesForPayments` pipeline - used to get the cost of using a particular payment option and method.
- The `getPaymentServiceUrl` pipleline - used to retrieve the payment service payment acceptance page.
- The `getPaymentServiceActionResult` pipeline - used to retrieve the result of a payment service transaction.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.10. Shipments

Configuration

You can find all configuration for the shipping subsystem in the `Sitecore.Commerce.Shipping.config` file. Here you will find all details for the entities, pipelines, and repositories used by the shipping system.

Entities

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

You can extend the following four entities that are defined in the Connect system for shipping:

- `ShippingOption` - represents detailed information about a shipping option (also known as a shipping category); contains information such as description and ShippingOptionType.
- `ShippingMethod` - represents information of a shipping method, where a shipping method is an implementation of a shipping option, for example, if the payment option is ShipToAddress then a shipping method for that would be FedEx or UPS.
- `ShippingOptionType` - extensible enum class used to contain the different types of shipping options, the default choices are None, ShipToAddress, PickupFromStore, and ElectronicDelivery.
- `LineShippingOption` - represents the selected shipping option for each line item, this enables the support of split shipments.

To extend any of these default entities, you create a new class that inherits from the appropriate type, and then patch the appropriate entity under <commerce.Entities> in the `Sitecore.Commerce.Shipping.config` file.

Pipelines

There are two pipelines for shipments that allow most basic functionality.

For example:

- The `getShippingOptions pipeline` - used to retrieve shipping options.
- The `getShippingMethods pipeline` - used to retrieve shipping methods.
- The `getShippingMethod pipleline` - used to retrieve shipping method details.
- The `getPricesForShipments pipeline` - used to get the cost of using a particular shipping option and method.

> **NOTE**
> For more information, see the Developer's Guide.

2.2.11. Wish lists

Configuration

You can find all configuration for the wish list subsystem in the `Sitecore.Commerce.WishLists.config` file. Here you will find all details for the entities, pipelines, and repositories used by the wish list system.

Entities

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

You can extend the following three entities defined in the Connect system for the wish list:

- `WishList` - represents a wish list and associated line items.
- `WishListHeader` - represents a summary of a wish list; contains information such as name, customerId, and isFavorite.
- `WishListLine` - represents a line item in a wish line; contains information such as quantity and assigned product.

To extend any of these default entities, you create a new class that inherits from the appropriate type, and then patch the appropriate entity under `commerce.Entities` in the `Sitecore.Commerce.WishLists.config` file.

Pipelines

There are numerous pipelines for wish lists that allow most basic functionality.

For example:

- The `getWishList` and `getWishLists` pipelines - used to retrieve wish lists.
- The `createWishList`, `deleteWishList` and `updateWishList` pipelines - used to create, delete, and update wish lists.
- The `addLinesToWishList`, `updateWishListLines`, and `removeWishListLines` pipeline - used to manipulate line items.

- The `emailWishList` and `printWishList` pipelines - used to extend functionality.

> **NOTE**
> For more information, see the Developer's Guide.

3. Product synchronization

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

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.

3.1. The basics of product synchronization

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

- Synchronize All Products
  The `SynchronizeProducts` service method synchronizes all products and related product repositories (that is, artifacts) that need 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, the `SynchronizeProductList` method is called, specifying the list of products to synchronize.
  Before calling the `SynchronizeProductList` method, all the related product repositories are synchronized.
- Synchronize Product List
  The `SynchronizeProductList` service method accepts a list of product IDs, which it iterates over and calls the `SynchronizeProduct` method.
  No related product repositories are synchronized as part of this, but are assumed to be up-to-date.
- Synchronize Product
  The `SynchronizeProduct` service method 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-to-date.
- Synchronize Artifacts
  The related product repositories such as Manufacturers, Product Types, Classifications (categories), and global specifications are referred to as product artifacts.
  The `SynchronizeArtifacts` service method 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, such as the EventDisabler, SecurityDisabler, and so on.

> **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. This 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.

3.1.1. Two-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 (ECS) to Sitecore content.

The logic that determines whether an entity is updated in the ECS, the content management system (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 the CMS. The possible direction values are:

- `Direction.Inbound` - one-way synchronization from the ECS to the CMS. This is the default value.
- `Direction.Outbound` - one-way synchronization from the CMS to the ECS.
- `Direction.Both` - two-way synchronization based on the configured synchronization strategy.

The default synchronization strategy is based on time stamps 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 two-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.

3.1.2. 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 and so on.

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 external commerce system (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 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 that 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 that 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 products, an additional processor is injected to delete a product if it no longer exists in the external commerce system and must be removed from the content.

```
<commerce.synchronizeProducts.synchronizeProductEntity>
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ReadProduct
FromSitecore, Sitecore.Commerce">
<param desc="productRepository" ref="productRepository" />
</processor>
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ReadExterna
lCommerceSystemProduct, Sitecore.Commerce" />
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ResolveProd
uctChanges, Sitecore.Commerce">
<param desc="synchronizationStrategy" ref="synchronizationStrategy" />
</processor>
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.DeleteProdu
ctFromSitecore, Sitecore.Commerce">
<param desc="productRepository" ref="productRepository" />
<param ref="entityFactory" />
</processor>
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.SaveProduct
ToExternalCommerceSystem, Sitecore.Commerce" />
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.SaveProduct
ToSitecore, Sitecore.Commerce" >
<param desc="productRepository" ref="productRepository" />
<param ref="entityFactory" />
</processor>
</commerce.synchronizeProducts.synchronizeProductEntity>
```

Figure 1 illustrates how the main `SynchronizeProducts` pipeline executes other pipelines internally to do a full synchronization.

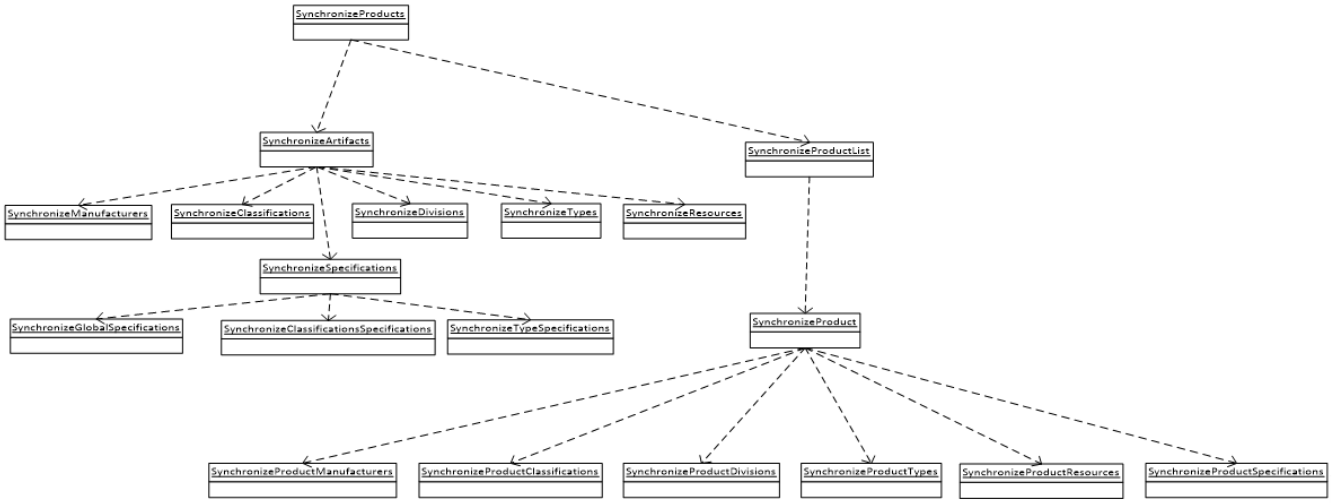


Figure 1: Call hierarchy between pipelines

Pipelines and naming conventions

Pipelines are generally split into two types:

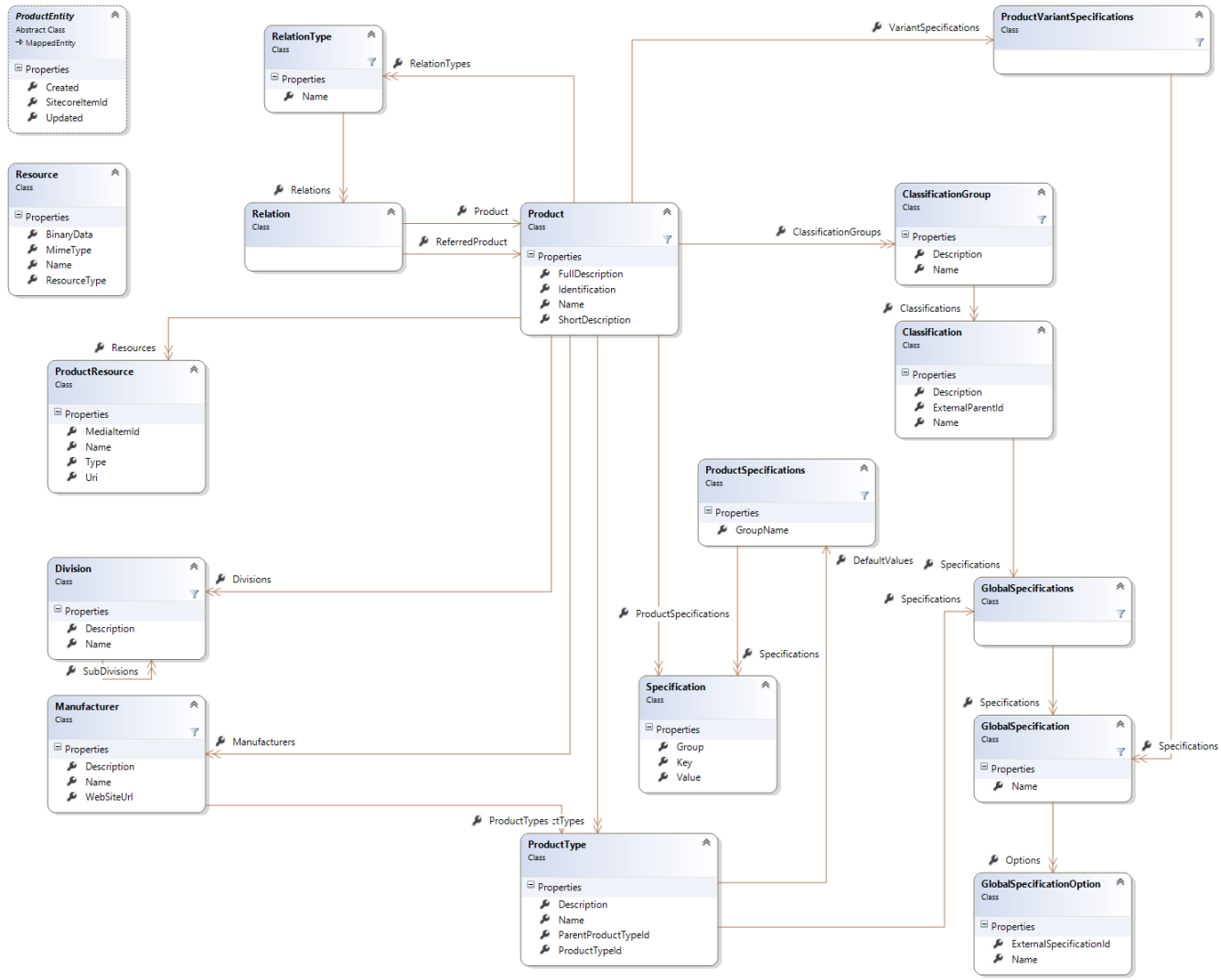
- Pipelines that operate on related product repositories, separate from the actual product repository. These separate repositories are used as references from the product repository. There are repositories for Manufacturers, Classifications, Types, Divisions, Resources and specifications. The pipeline names are prefixed with `Synchronize`. For example, the `SynchronizeManufacturers` pipeline, 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. For example, the `SynchronizeProductManufacturers` pipeline, which is responsible for synchronizing the connections/references between a specific product and its related manufacturers stored in the separate Manufacturers repository.

Processors that begin with the word `Run` are responsible for calling a separate pipeline and transferring the needed parameters. For example, the `RunSynchronizeManufacturers` processor that executes the `SynchronizeManufacturers` pipeline.

3.1.3. Integrating with Connect

When you integrate products with Sitecore Commerce Connect using product synchronization you might have to customize 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 The Object Domain Model.

There is a one-to-one relationship between the product item templates and the objects.



Implement a custom product entity

The domain model consists of a list of Sitecore product templates and the corresponding object type.

To implement a custom product entity:

1. Create 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.
2. If synchronization needs to go both ways, create the following two additional processors for each pipeline:
    - A processor that stores the product data externally. The product data will be given in an instance of the corresponding domain model object type, for which the pipeline is responsible.
    - A processor that resolves the differences and determines the resulting output. For more information, see Create a custom synchronization strategy.

NOTE

When reading data in the external system and populating an instance of a domain model entity, you must use unique names for entities of the same type so that there will be unique item names in Sitecore.

Without unique names, there is no guarantee 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 selects the first item with a given name

3.1.4. 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 code 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 configurations 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, for example, `manufacturerRepository`.
- A <path> parameter element that refers to the main <paths> element and specifies 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 that is used as input to the IDGenerator along with an external ID to ensure the outcome is a unique GUID ID (which can be used as a unique item ID). For more information, see Implement a Custom ID Generator.

There are special repository types with names that start with “product”. These repository types do not store data in a separate repository but instead augment the main product entity with references to the separate repositories. For example, the `productManufacturerRepository` repository has the responsibility of managing the references between the product item and the related manufacturer items. The configuration for the `productManufacturerRepository` is shown in the following code snippet.

Instead of a template and path element, the code snippet uses <param> elements to specify the name of the field on the product item that holds the references (for example, item IDs) to the related manufacturers. As these types of repositories need to generate the right item IDs, they need to know the same prefix 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 provide 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");
```

3.1.5. 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, which automatically generates a new GUID for each new item created.

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 with the following benefits:

- No need for mapping tables, which take up space.
- Retrieving the ID of the corresponding item is very quick.
- No need to search for the items in Sitecore if the external ID is provided.

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

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

You can also create a custom ID mapping implementation.

3.1.6. Indexing

When Connect is installed the default index is patched to exclude all product data and a separate product index is created that 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.PauseSearchIn
dexing, Sitecore.Commerce" />
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.SynchronizePr
oductList, Sitecore.Commerce" />
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.ResumeSearchI
ndexing, Sitecore.Commerce" />
<processor
type="Sitecore.Commerce.Pipelines.Products.SynchronizeProductList.RebuildSearch
Indexes, 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 are used to store synchronized products.
Can be stopped, resumed and rebuilt automatically during product
synchronization.
-->
<setting name="ProductSynchronization.ProductIndexes"
value="sitecore_master_index, commerce_products_master_index" />
```

> **NOTE**
> If new or custom product templates are introduced, both the DefaultIndexConfiguration and the Product index configurations must be updated.

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

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>
```

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.GetContextI
ndex.FetchIndex,
Sitecore.ContentSearch']"/>
</contentSearch.getContextIndex>
```

- To include extended product data.
  A `commerce.inventory.stockStatusForIndexing` pipeline 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 the `Sitecore.Commerce.Products.Lucene.Index.Common.config` file. The following table contains an example of the product index extended with four fields:
  - In-Stock—contains a list of locations where the product is in stock.
  - Out of Stock—contains a list of locations where the product is out of stock.
  - Location—contains a list of locations where the product is orderable from.
  - 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	

The product index is defined in the `Sitecore.Commerce.Products.Lucene.Index.Master.config` file. 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 the 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 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 following 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 `Sitecore.Commerce.Products.config` configuration file. 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>
```

3.2. The Connect product data model

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

- To have a product data model that fulfills the identified end-user scenarios.
- To have a single common product data model regardless of the external commerce system (ECS) used.
- To have the same model across different solutions making it easier for solution developers.
- To have components that are easy to build and maintain for UI component developers because the data model remains the same across external commerce systems.

Product data is complex and in Connect there is not a one-to-one mapping between a single 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 designed to:

- Provide a solid base for typical e-commerce scenarios.
  The main reason for pulling product data into Sitecore is to augment it and present it on different shops and channels, for example, media, web, mobile, print.
  The product data must be normalized to ensure it is valid and provides a sound foundation to build on and to enable scenarios to be fulfilled.

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

- Avoid redundant product data
  Having too much redundant data (for example, storing all data for a single product in one Sitecore item) results in the need to manually synchronize and update data in Sitecore and this becomes difficult to maintain.
  Instead separate repositories are used for shared data such as manufacturer information, divisions, and specifications and referred to by way of linking. This is similar to the way normalized product data is stored in separate tables in an SQL databases and references with foreign keys.
- Minimize product data synchronization
  By avoiding redundant data, the amount of data to synchronize is minimized. Also, not all product data is needed from the ECS. Only product data needed to fulfill the scenarios described in the following sections are synchronized and stored in Sitecore.
- Avoid typical custom model implementation issues

By dividing product data into a composite product structure and placing data into separate repositories, the most typical implementation problems can be avoided. Problems such as:

- Flat model
  In a flat model, a single item represents a product. Without composition of multiple items to make up product data, the model will:
  - Be simple, missing out on a lot of the needed information.
  - Have a lot of unused fields or use a large number of templates to cover all the different product types. Both are not best-practice.
  - Have a lot of redundant data for similar products or variants.
  - Force data into custom field types or encode data using custom schemes.
- Redundant data
  Without separate repositories for storing shared information, redundant information is unavoidable.
- Too many templates
  Using a separate template for each product type soon becomes unmanageable 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 time consuming

By leveraging the new features in CMS 7, it is possible to use buckets and features such as 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 latest version of the Data Definition API Cookbook for Sitecore.

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 the ECS to be integrated with, customized and extended.

3.2.1. Minimum product concepts

This topic lists the minimum product concepts that must be part of the product data model:

- Main product data such as IDs - EAN, SKU, IBSN, and so on, name, brand, model, tags, and so on.
- Multiple classification schemes (that is, multiple different categorization schemes):
  - Typically products are stored in categories in the e-commerce system according to type.
  - For presentation, a different way of organizing products into categories is typically used.
- Specifications and lookup values on three levels:
  - 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.
  - Product type. Specifications that are specific to the given product type. These specifications are more closely related to the actual product and its manufacturer.
  - Single product. Specifications that only apply to the specific product.
- Product variant significant specifications:
  - It must be possible to configure which specifications that make up the difference between all variants of a product, for example, what specifications make up the variants.
- Related products - there are different types of relations between products:
  - Variants of the same product.
  - Accessories.
  - Cross-sell, up-sell, and so on.
- Resources - images and files.
- Manufacturer information.
- Divisions - 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.

3.3. 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.

3.3.1. 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.

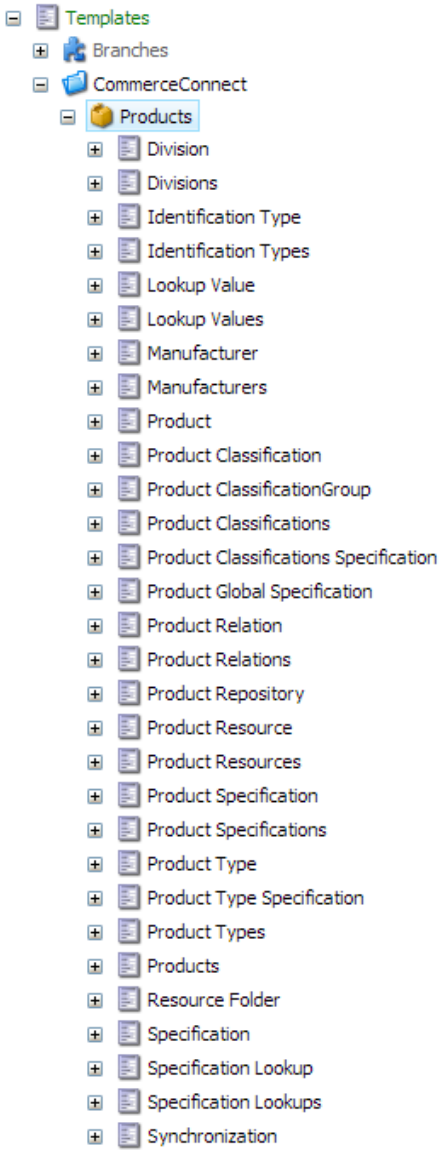
Naming conventions

For each concept in the model, product, specification, type, and so on, there is an entity template and, typically, a corresponding folder template (for example, each type of entity is kept in a folder). The naming convention is that the folder template name is plural and the entity template name is singular. For example: *Specification* represents a specification entity and *Specifications* represents the folder in which the specifications are stored.

The product data model consists of data that is stored in separate repositories and data that is specific to a given product. For settings that relate 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 in the following screenshot.



For every entity type, there is usually a folder template that holds the individual items. The following table describes the individual entity templates and lists 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 a hierarchical manner. A division can have multiple sub-divisions. The Template Divisions folder is used as a 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, for example, a product has both an EAN and SKU number besides the internally used product code. Identification Types is used as a 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 the 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, for example, 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 subitems: Relations, Resources, and Specifications.
---	Product Artifacts	Product Artifacts is the folder template grouping together miscellaneous repositories relating to products such as Manufacturers, Classifications, Divisions, Types, and global lookup values.
Product Classification Group	Product Classifications	A product classification Group represents a classification scheme. A lot of standards as well as custom classification schemes exist in the world today. Multiple different classification schemes and categorizations might be used concurrently in the product data model, for example, products could use both the categorization used in the external commerce system as well as the UNSPSC classification scheme. UNSPSC is 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 the previous description for the Product Classification Group.

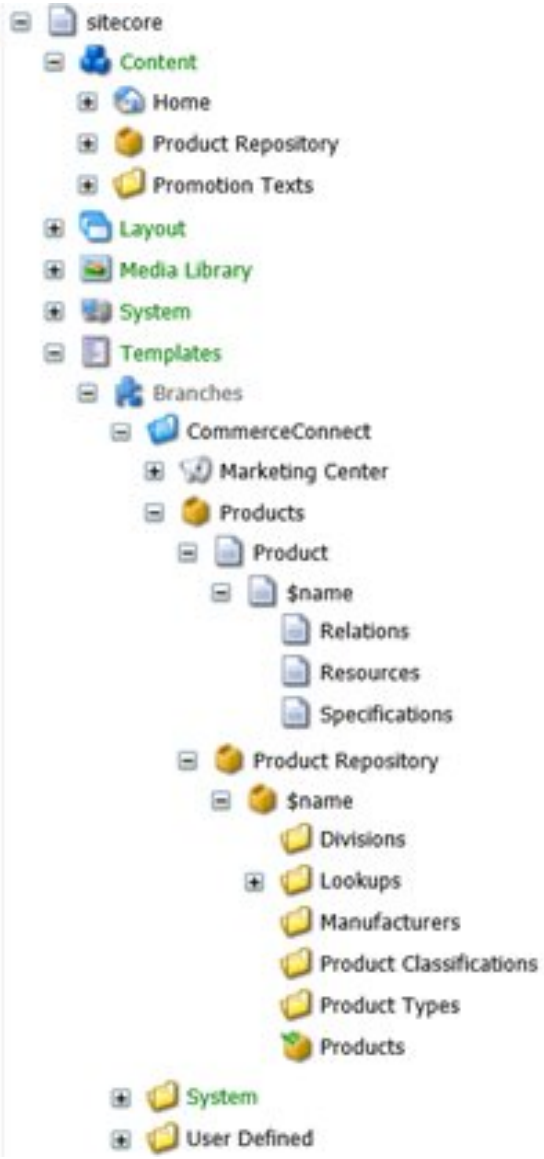
Entity template name	Folder template name	Description
		Product Classifications are structured in a hierarchical manner beneath a Product Classification Group.
Product Relation	Product Relations	A Product Relation represents a relation between the given product and other products in the repository.
Product Resource	Product Resources	A product resource represents a media entity, for example, a file (brochure), an image (main image or alternate images). Resources are not always stored in the Sitecore Media Library and can be represented by a URI. Resources are stored in its own folder based on the 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 the Product Types template organized in a hierarchical structure. Subtypes 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:

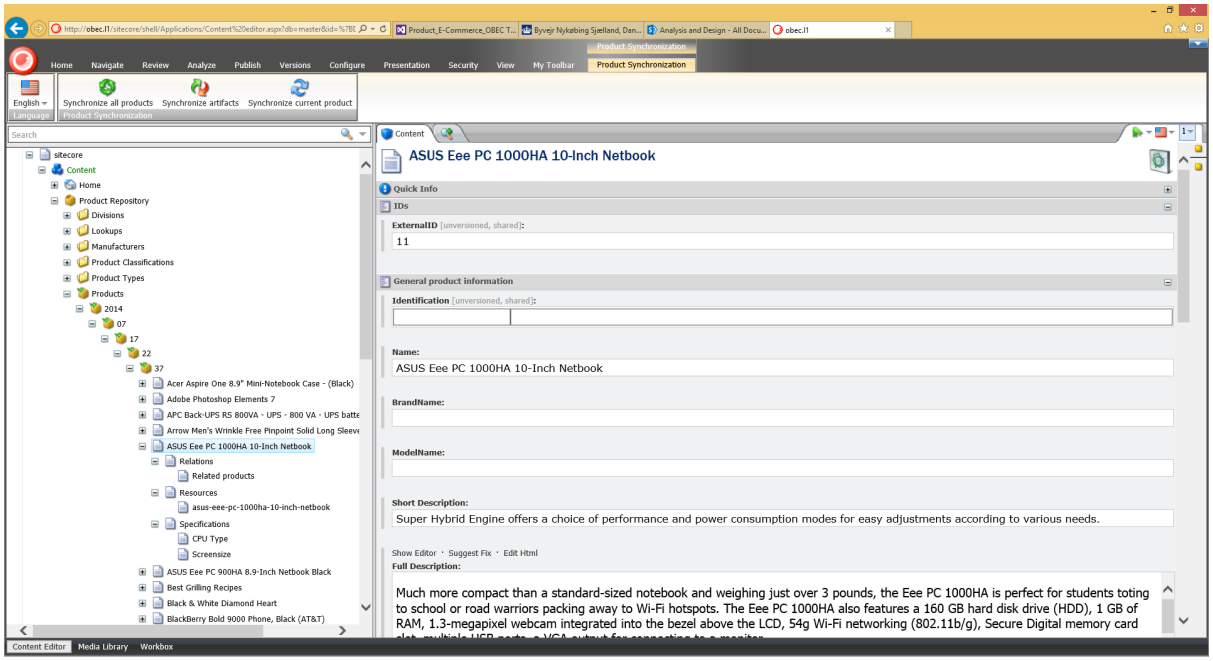
Branch template name	Description
Product	The branch template contains a composite tree structure that represents a product with default subfolders for related products, resources and specifications. The template is associated with the bucket that makes up the main product repository.
Product Repository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

The expanded branch templates are displayed in the following screenshot.



3.3.2. Main product data in one product repository bucket

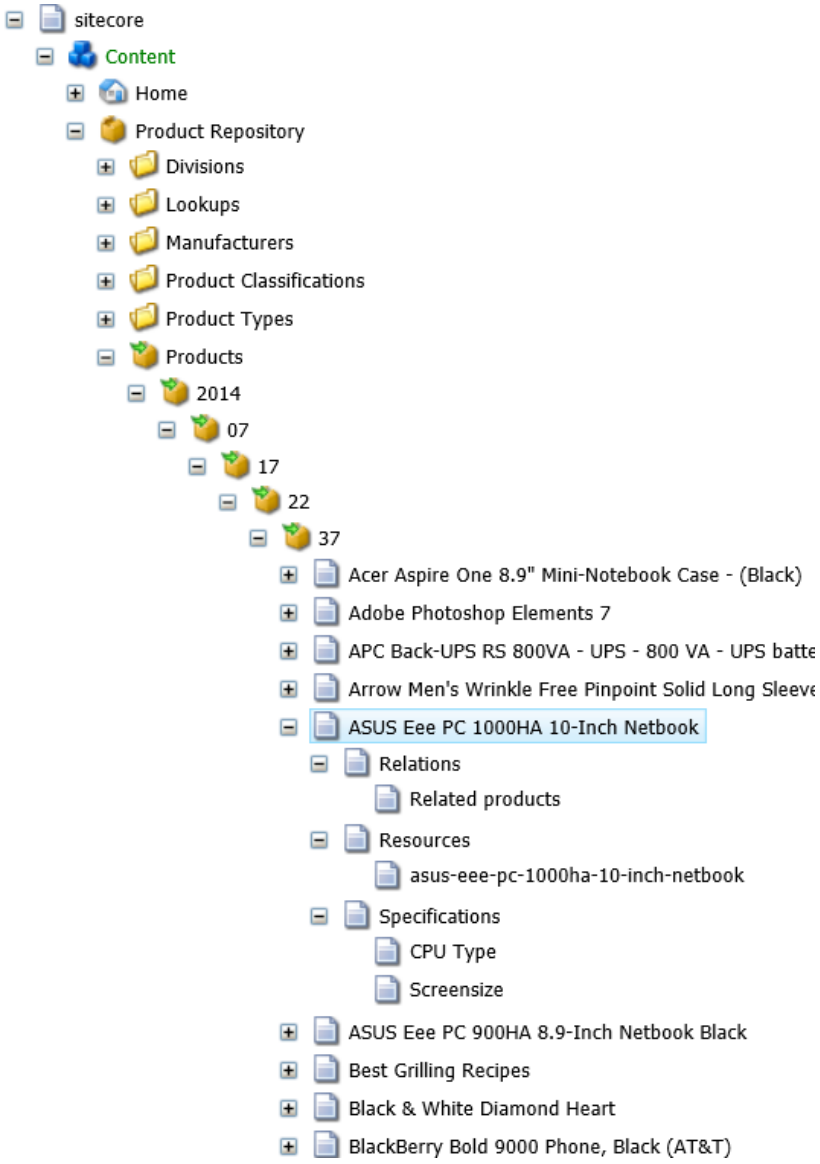
To store a large number of products in a single product repository, a bucket is used. One product consists of a main product item and a sub-tree of items containing values specific to the given product and references to other repositories.



The product template only contains the most essential information that applies to all products.

3.3.3. Product relationships, resources, and specifications

Each product has a composite structure organized in a subtree for storing relationships, resources, and specifications that apply to the particular product:



3.3.4. Product variants

Generally a variant is considered a product in Connect, so there is 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, you can use the concept of a product family. 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 contain values that differ from the master product.

A Product type can be regarded as the master product as it can hold default specifications. and so on. You must assign product variants a product type and only the specifications that differ need to be set for the variant.

3.3.5. Specifications

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

- 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 predefined values are defined.

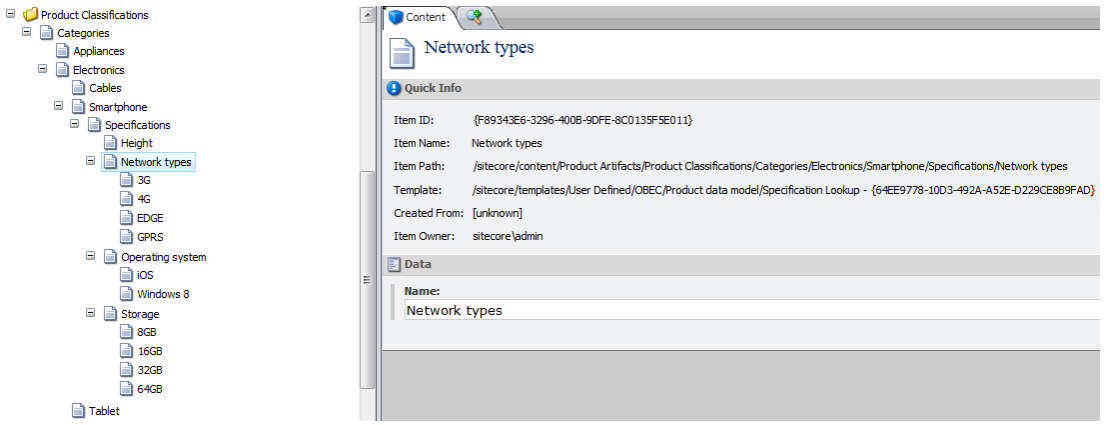
- Classification specifications
  For each classification scheme there is typically a list of specifications associated with each category. These are the specifications that make it possible to compare all products present within a category made by different manufacturers.
  These are also the specifications normally used for navigated or faceted search.
- Type specifications
  There is a close relationship between a product and its type and therefore there are three pieces of specification information available on types:
  - Specifications in the form of keys or keys + values (lookup tables).
  - Specification options that narrow down the lookup table options for subtypes.
  - Specification default values that contain values for specification. All products of the given type will have those values.
- 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.

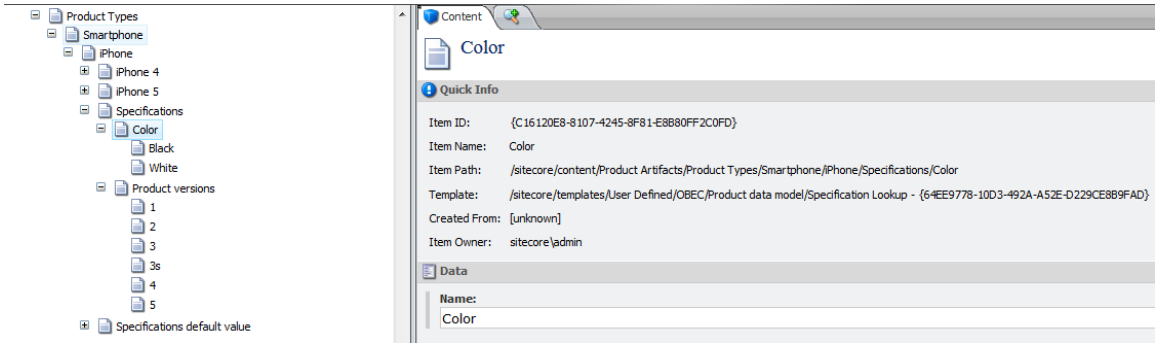
For each specification there's typically a specification value. The value is stored either on the type as default value or directly on the product itself.

- 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.



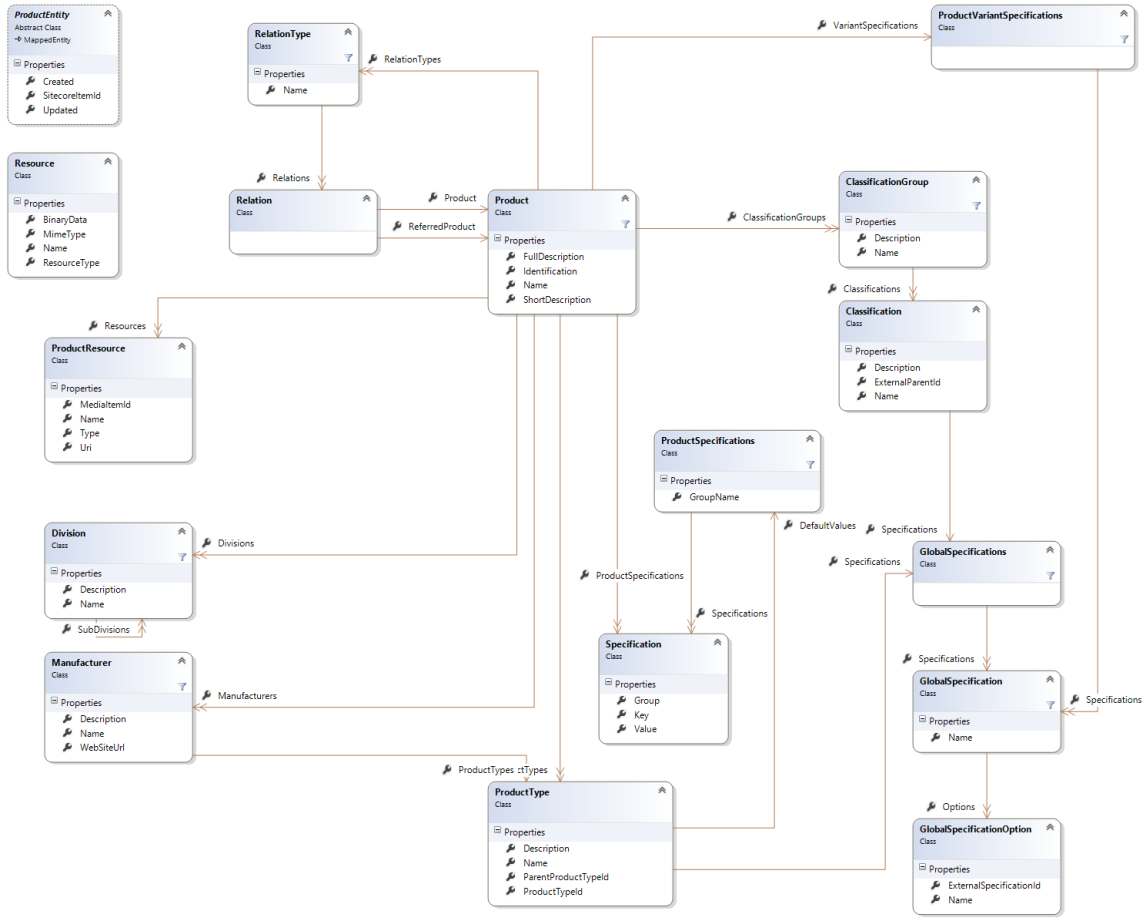
Specification values

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 the *Product Specification* template.
- 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 the *Product Specification* template.

3.4. The object domain model

The following object diagram visualizes the object domain model. There is a one-to-one correspondence between the product item templates and the objects.



NOTE

For more information, see the Developer's Guide.

3.5. Implement a custom product entity

When deciding whether to add a field to the main product template and corresponding object or whether to add it to a subitem, ask yourself whether the field applies to all products.

To extend the main product entities:

1. Create a custom template that inherits from the default product template (`/sitecore/templates/CommerceConnect/Products/Product`) and extends it with further fields.
2. Create a custom product class that inherits from the default `Sitecore.Commerce.Entities.Products.Product` class and extends it with further properties.
3. Create a custom `ProductRepository` class that inherits from the default `Sitecore.Commerce.Data.Products.ProductRepository` class.
4. Override the following two methods to save and load the extended properties. Call the base implementation of these methods:
   - `protected override void UpdateEntityItem(Item entityItem, Product entity)`

- `protected override void PopulateEntity(Item entityItem, Product entity)`

5. To update the `ProductRepository` element in the `Sitecore.Commerce.Product.config` file:

   - Replace the value of the attribute type with the full type name of the custom product repository class. See the following snippet with the type value in italics.
   - Replace the template ID set in the subelement template. See the following snippet with the template value in italics.

```
<productRepository
type="Sitecore.Commerce.Data.Products.ProductRepository,
Sitecore.Commerce" singleInstance="true">

<template>{47D1A39E-3B4B-4428-A9F8-B446256C9581}</template>
```

   - In the `IncludeTemplates` section, update the GUID of the `ProductTemplateID`. For more information, see Product Index.
   - In the `Commerce.Entities` section of the `Sitecore.CommerceProduct.config` file, update the type attribute of the Product entity entry:

```
<commerce.Entities>

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

   - In the `ExcludeTemplates` section of the `Sitecore.Commerce.Products.LuceneDefaultIndexConfiguration.config` file, update the GUID of the ProductTemplateID.
     For more information, see Default Index.
   - If two-way synchronizing is used, create a custom `ResolveProductChanges` class that inherits from the default `Sitecore.Commerce.Pipelines.Products.SynchronizeProductEntity.ResolveProductChanges` class.

New properties that refer to items in related repositories, such as `Manufacturers`, load a collection of the given type and populate the values. Some internal helper methods, such as `PopulateEntityFieldCollections`, are provided. The following 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)
```

Follow the same procedure for other product entities.

3.6. Create a custom processor

For two-way synchronization to work, a processor must be present that 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 are saved to both Sitecore and the external system if 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:

1. Create a new class that inherits from the `ResolveChangesProcessor` processor.
   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. The naming convention for the `PipelineArgs` collection is to prefix the type name with the word Sitecore as the key. For 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. The naming convention for the `PipelineArgs` is to simply use the type name as the key. For example, `Manufacturers`.
4. Override the `SaveEntities` method.
   The method must save the resulting entities to the `PipelineArgs` collection. The naming convention is to use the type name as the key. For example, `Manufacturers`.

The implementation of the `ResolveManufacturerChanges` processor is shown in the following code snippet:

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

3.7. Create a custom synchronization strategy

To use a custom synchronization strategy:

1. Create a new custom strategy class and implement the `ISynchronizationStrategy` interface.

This interface contains one Resolve method that receives the direction of synchronization and a base product entity from the external system and Sitecore.
The Resolve method decides which system to update (theECS or Sitecore) and returns the result.

```
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 a custom synchronization strategy class in the `Sitecore.Commerce.Products.config` file.
   To do this, change the type attribute value of the `synchronizationStrategy` element to custom synchronization strategy type.

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

The default DateTime based strategy that comes 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;
}
}
```

3.8. Implement a custom ID generator

The ID generator takes a unique external ID in the form of a string and returns a unique ID, which can be used as an item ID (for example, GUID). The default implementation is based on the MD5 hash algorithm.

> **NOTE**
> Sitecore IDs must be unique and because the output from the ID generator 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 the ID generator must therefore be in the following format: Manufacturer + ManufacturerID and Products + ProductID respectively.

To use a custom ID generator:

1. Create a new ID Generator class and implement the `IIdGenerator` interface.
   The interface has one `StringToID` method that accepts two parameters:
   - A string containing the value of the external ID of the given entity.
   - 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 the custom ID Generator class in the `Sitecore.Commerce.Products.config` file.
   To do this, change type attribute value of the `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));
}
}
```

3.9. Performance improvements

Performance improvements in Sitecore Commerce Connect include:

- 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, you must synchronize the bucket. You can do this by:
  - By synchronizing a single product, which can immediately be moved to the right place in the bucket. This is done by calling `SynchronizeProduct` as a single operation from `BucketManager.MoveItemIntoBucket(entityItem, root);`
  - By doing a bulk synchronization. This is done by calling the `SynchronizeProducts` or `SynchronizeProductList`, which can, however, result in the creation of many new items that must 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. This will eliminate the time spent touching all existing items in the bucket, which could be significant, for example, adding 1000 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.

  For more information on how to enable delayed bucketing, see Delayed bucket synchronization.

- Multithreaded synchronization. A single thread is, by default, created to synchronize products, manufacturers, types, resources, divisions, and specifications in parallel. The threads are created for each repository being synchronized. You can configure the number of threads to use 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 indexing, events, and caching. Triggering item events as well as indexing is disabled while synchronizing in order to conserve resources and avoid indexing before synchronization is complete. Indexing is turned on after synchronization has finished. For more information, see Indexing.
- Reading product data once and processing it in multiple pipelines reduces the number of calls between Sitecore and the external systems. All product entities in Connect are synchronized using their own pipelines, which naturally lends itself to reading the data from the external system in the individual pipelines. In this 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 once initially and passed on to the individual subpipelines 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 this reason, you can either import resources into the Sitecore media library or simply refer to them 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.

3.10. Delayed bucket synchronization

You can synchronize products into the bucket where the main product data is stored in the following two ways:

- Immediately - this is the default.
- At the end of the entire Commerce Connect product synchronization - to activate this approach, remove the suffix `.disabled` from the `Sitecore.Commerce.Products.DelayedSyncProductRepository.config.disabled` config file.