sitecore®

Sitecore Commerce Connect 7.5

The Commerce Connect Developer's Guide

A Developer's Guide for using Commerce Connect to build E-commerce solutions

Table of Contents

Chapter 1

Introduction

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

Note
In the following, Connect is used as an abbreviation for Sitecore Commerce Connect and ECS is used for External Commerce System.

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

This guide describes the API and configuration of Connect for frontend developers who create Sitecore solutions and are looking for information about how to use Connect

If you are a developer looking for information about how to integrate Connect with an external commerce systems, see the Connect Integration Guide

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

- **Chapter 2 — Using Connect to build e-commerce solutions**
  This chapter describes how to use the Connect API as a solution developer.

Chapter 2

Using Connect to build e-commerce solutions

This chapter describes the Connect API which consists of a number of abstract service layers. Each section in this chapter describes a service layer, its relevant classes and how to use it.

This chapter contains the following sections:

- Carts

- Pricing

- Order

- Inventory

- Customer

- Product Sync

- Connect Configuration

2.1 Carts

2.1.1 Cart Domain Model

This chapter describes the domain model that represents the cart and its constituent Data Transfer Objects (DTO). The domain model is used when a Sitecore developer needs to interact with the cart and transfer cart data back and forth to the ECS. All business logic for carts is implemented in the service layer API, which must be used to manipulate carts.

Some examples are of cart domain model usage:

- To collect information from visitors
- To display information in renderings
- To use information for personalization
- To pass cart information to the external commerce system
- To return cart total from pricing service layer

Note: Because cart has strong references to Price and Total they are included in the domain model, but methods for calculating total and getting prices are part of the Pricing integration service layer.

Note: The class diagram below shows the domain model. All the Cart classes, prefixed with Cart, to the left, are described in this document, whereas the Price, Total and TaxTotal classes to the right are described in the Pricing document and is manipulated by the pricing API as well as the Party, ShippingInfo and PaymentInfo are part of the Customer, Shipping and Payment service layers respectively. For more information see the respective service layer documents for more information.
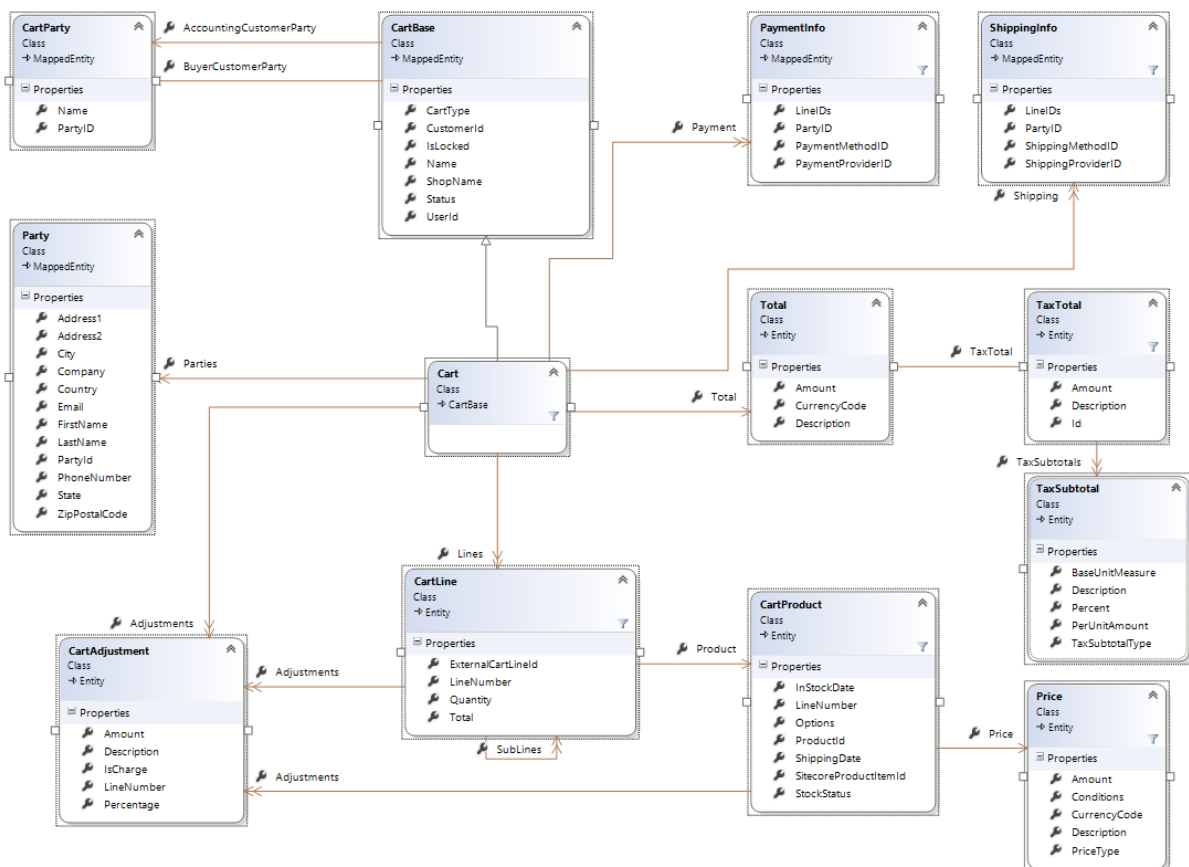
Note: The domain model consists of classes that make up the contracts with the external system. The contracts are defined as classes instead of interfaces to allow the model to be easily extended later if needed. This follow the best practice guidelines defined in the book Framework Design Guidelines.

Default implementation of the contracts are delivered as part of Connect. If an actual Connect provider with an external commerce system contains more functionality than provided by default, the implementation can be replaced. All instantiation of actual classes will be handled using the Factory design pattern.

IReadOnlyCollections and service API

Most collections in the domain model are specified as IReadOnlyCollections. This is done to avoid developers modifying these collections directly. All manipulation must be done through calling of the service API.

Cart Keys

A cart has two unique composite keys:

- CartId (aka. ExternalCartId), ShopName
  Some methods like LoadCart use this key to locate a cart.
- UserId, CustomerId, CartName, ShopName
  Some methods like GetCarts and CreateCart uses this key

Class: Cart Base

The Cart Base represents a summary of the main cart data and is used as return values from GetCarts and CreateOrResumeCart in case of multiple matching carts.

The Cart class is inherited from Cart Base. For more information, see class Cart below.

Name	Type	Description
Name	String	Can be used to name a cart. The property is optional to use, but can be used in solutions where a single visitor/user can have multiple carts. In order to differentiate between the carts, a string value must be provided.
UserId	String	The identifier for the user of the cart, can be: • the contact Id or Identifier provided by Sitecore • A user id provided by Sitecore • A user id provided by ECS
CustomerId	String	The identifier for the customer of the cart. The customer cannot manipulate the cart directly, but only indirectly through the user. For more information, see the Connect roles document:
ShopName	String	The shop name where the cart belongs. Is used to enable multi-shop support. Can in implementation correspond to website name if there is a one-2-one mapping between website and shop. All API methods take ShopName as parameter.
IsLocked	Boolean	Indicates if the cart is locked or not A cart is typically locked during part of the checkout process, especially after initiation of payment

CartType	String	Contains type values for different usage of cart. Example values are: Regular, Gift card, Wish list, Recurring-cart. In default implementation only "Regular" will be used and tested Can be extended later and customized in the actual Connect provider implementation.
CartStatus	String	A status code: InProcess, Abandoned. Can be extended later and customized in the actual Connect provider implementation. The value is updated in the Abandoned Cart EA plan, if a cart is resumed from abandoned state

Class: Cart

The Cart is responsible for representing the collection of products a customer is planning to buy. It is used when the content of the cart needs to be displayed or when Sitecore needs to determine the content of the cart for any other purpose (such as personalization).

The Cart class inherits its base cart data from Cart Base. For more information, see class Cart Base Above.

Name	Type	Description
CartLines	IReadOnlyCollection<CartLine>	Contains the cart lines.
Adjustments	IReadOnlyCollection<Adjustment>	Collection of adjustments that describe any discrepancies between the total and the sum of the prices of the individual cart line items. Examples are coupon codes (manually triggered), and shipping surcharges (automatically triggered)
CartTotal	Total	Represents the total value for the products in the cart.

Class: Cart Line

The Cart Line represents a line in the shopping cart. It represents something that a visitor has added to his cart, along with the quantity of the item that was added to the cart. It also represents the position of the line relative to other lines in the cart (for controlling the order the lines appear when the lines are displayed).

Name	Type	Description
ExternalCartLineId	String	Unique identifier for the cart line in the commerce system. Will be empty when adding cart lines and can be set by the ECS. Can be specified when removing cart lines Can be specified when updating cart lines
Quantity	Unsigned integer	The quantity of the specific cart product in the cart.

Product	CartProduct	The cart line product. The Cart Product object contains the reference to the actual product
LineNumber	Unsigned integer	The position of the cart line in the cart. Can be specified when removing cart lines Can be specified when updating cart lines Can be empty when adding cart lines
Adjustments	IReadOnlyCollection<Adjustment>	Collection of adjustments that describe any discrepancies between the total and the cart item price. The actual content of the adjustments are provided by the ECS.
Total	Total	Represents the total for the product represented by the cart product (sub-item).This value will be the same as the total on the cart product (sub-item) multiplied by the quantity including list of Adjustments. **Read-only**. Is reserved for Pricing Service Layer. Should not be supplied when adding, removing or updating cart lines
SubLines	IReadOnlyCollection<CartLine>	Recursive sub-lines For example, if the cart item is a bundled product, the sub-lines are the individual products that make up the bundle. Another example is services like insurance that is added to a product

Class: Cart Product

The Shopping Cart Product represents a product in a Shopping Cart Line, or a sub-item of a Shopping Cart Product in case of bundling or insurance etc.

Name	Type	Description
ProductId	String	Unique identifier for the product in the external commerce system.
SitecoreProductItemId	ID	Shortcut for referencing the product item in Sitecore directly
Options	IReadOnlyCollection<Option>	Represents visitor-specified product options (such as engraving=Adam)
LineNumber	Unsigned integer	Gets the position of the cart item in the cart.
Adjustments	IReadOnlyCollection<Adjustment>	Collection of adjustments that describe any discrepancies between the total and the price of the cart item's product.
Price	Price	Contains the product price. Can be supplied when adding or updating cart lines to cart, but can also be set by Pricing service layer during call to GetCartTotal

Class: Cart Adjustment

A cart adjustment describes a charge or a discount given on a Cart, Cart Line or Cart Line Product.

Examples are:

- Coupon codes (manually triggered) on the Cart or the Cart Line
- Shipping surcharges on the Cart or Cart Line (automatically triggered)

The adjustment can be a fixed amount or a percentage, but not both at the same time. If both an amount and percentage is applied, then two adjustments must be given and the LineNumber specifies the order in which they are applied

Name	Type	Description
Amount	Decimal	Amount that needs to be added or subtracted to sub-total. Only Amount or Percentage (see below) can be used at any one time
Description	String	A description of the charge or discount applied. Description could include a reference to coupon code, if that's the scenario
IsCharge	Boolean	Indicates it is a charge if true, otherwise discount
Percentage	Float	A percentage that needs to be added or subtracted. Only Amount or Percentage can be used at any one time
LineNumber	Unsigned integer	If multiple adjustments are given the LineNumber specifies the order in which the charge or discount is applied. The information stored in adjustments is used for presentation, showing calculation order and not for actually calculation to the cart.

Class: Cart Option

Option represents a configurable or variable value that is specific to an item in the cart. An example of an option is engraving on a personalized product. When the "engraving" service is added to the cart, the text to engrave is also required. The Option represents this.

If an option has an additional charge, the option should be represented by a separate Cart Line Product. The option is only used to describe variables or settings on a specific Cart Line Product, not to handle adjustments or pricing changes.

Name	Type	Description
OptionId	String	An internal ID for the option, typically provided by Sitecore when the user adds an option to the cart line
Description	String	An optional description referring to the type of option. For example, Engraving, Name on t-shirt etc.
Value	String	The actual custom value

2.1.2   Cart Service Provider

Service providers are wrapper objects designed to make it easier to interact with Connect pipelines. The providers implement no logic other than calling Connect pipelines. All of the business logic is implemented in the pipeline processors.

For each method in the provider there is a corresponding Request and Result object used, ex. GetCarts takes a GetCartsRequest object and returns a GetCartsResult object. In some cases the response objects are re-used when returning the same data.

Customized versions of the default request and result arguments can be used by calling the overloaded generics based versions of the methods.

The Cart Service Provider contains the following methods for interacting with cart data.

GetCarts

GetCarts is used to query Cart data against the external commerce system and **doesn't** return a collection of Carts, but a collection of CartBase objects that only contains the summary of the main cart data.

Name:	GetCarts
Description:	Gets the carts that match the specified criteria. Calls the pipeline "GetCarts"
Usage:	Called when a list of carts is needed.
	Examples include:
	• Getting the carts for a specific visitor across all visitors
	• Getting one of the carts for the current visitor in a multi-cart solution
	• Getting the carts that have not been used within a period of time, for example, abandoned
Signature:	GetCartsResult GetCarts(GetCartsRequest request)
Input:	
	UserId – Optional - The ids of the users whose carts should be retrieved. If no value is specified, the user IDs are not considered when retrieving carts.
	CustomerId – Optional – The ids of the customers whose carts should be retrieved. If no value is specified the customer IDs are not considered when retrieving carts.
	CartName – Optional - The names of the carts that should be retrieved. If no value is specified, the cart names are not considered when retrieving carts.
	CartStatus – Optional – The status of carts that should be retrieved. Examples include "Active" and "Abandoned". If no value is specified, the cart statuses are not considered when retrieving carts. This could be used in a B2B scenario when you want to display a list of available carts to a user but only carts that are not locked.
	IsLocked – Optional – If provided it will mean the search also will filter on whether the cart is locked or not
	ShopName – Optional. Name of shop to search for carts in

Output:	
	IEnumerable<CartBase> – A collection of CartBase objects
	The lists represent the carts that match the criteria specified in the request.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create request to get the carts.
var request = new GetCartsRequest("MyStore")
{
    UserIds = new Collection<string> { "John" },
    CustomerIds = new Collection<string> { "JohnCustomerId" },
    Names = new Collection<string> { "JohnsName" },
    Statuses = new Collection<string> { "InProcess" },
    IsLocked = false
};

// Call service provider and receive the result.
var result = cartServiceProvider.GetCarts(request);
```

CreateOrResumeCart

Name:	**CreateOrResumeCart**
Description:	Initiate the creation of a shopping cart and in the process: • Tries to load persisted, potentially abandoned cart, if present • Trigger event in DMS • Enter user in Engagement Automation plan with ID of shopping cart.
Usage:	Called when a shopping cart is needed upon visitor arrival to shop.
Signature:	CreateOrResumeCartResult CreateCart(CreateOrResumeCartRequest request)
Input:	All four input parameters are used to search and match against existing carts for the current visitor, but only two of them are mandatory.
	UserId – Mandatory
	CustomerId – Optional
	CartName – Optional
	ShopName – Mandatory
Output:	
	Cart – A Cart object instance which represent the shopping cart. In case multiple carts already exists for the current visitor and it is undecided which one to return, then no cart is returned
	IEnumerable<CartBase> - In case multiple carts already exists for the current visitor and it is undecided which one to return, then a list of CartBase objects are returned
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create the request.
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Bred");

// Call the service provider to get the cart
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

// Read the id of the returned cart
var id = cart.ExternalId;
```

```
// Create request to remove the cart.
var deleteCartRequest = new DeleteCartRequest(cart);

// Call the service provider and receive the result.
cartServiceProvider.DeleteCart(deleteCartRequest);
```

LoadCart

Name:	LoadCart
Description:	Gets the cart with given Cart ID on the specified shop. Calls the pipeline "LoadCart". This method returns the full cart object with all cartlines associated.
Usage:	Called when a specific cart is needed
Signature:	LoadCartResult LoadCart(LoadCartRequest request)
Input:	
	CartId – required
	ShopName – required
Output:	
	Cart – A cart object instance which represent the shopping cart that matches the criteria specified in the request.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create the request
var request = new LoadCartRequest("Autohaus", "Bred");

// call the service provider to get the cart
var cart = cartServiceProvider.LoadCart(request).Cart;
```

SaveCart

Name:	SaveCart
Description:	Saves the specified cart in the external system if supported as well as in Sitecore EA state. Calls the pipeline "SaveCart". Called from other service layer methods implicitly, but not called explicitly
Usage:	Called when a specific cart needs to be persisted. The method should be executed after any operation that modified the cart resulting in a change of cart. It's executed implicitly when update cart, adding, deleting or updating cart lines as well as locking and un-locking the cart.
Signature:	SaveCartResult SaveCart(SaveCartRequest request)
Input:	
	Cart – required
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create cart and lock it.
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Mark");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

// add a cartline to the cart
var cartLine1 = new CartLine
{
    Quantity = 1,
    Product = new CartProduct
    {
        ProductId = "Audi Q10",
```

```
                Price = new Price(55, "USD")
        }
};

var cartLines = new Collection<CartLine> { cartLine1 };
var addCartLinesRequest = new AddCartLinesRequest(cart, cartLines);
cart = cartServiceProvider.AddCartLines(addCartLinesRequest).Cart;

var saveCartRequest = new SaveCartRequest(cart);
var result = cartServiceProvider.SaveCart(saveCartRequest);
```

AddCartLines

Name:	**AddCartLines**
Description:	Responsibility is to add lines to cart.
Usage:	Called when a list of cart lines is about to be added to the shopping cart. UI wise when the user clicks the Add-To-Cart button.
Signature:	`AddCartLinesResult AddCartLines(AddCartLinesRequest request)`
Input:	
	Cart – Required - The cart must be unmodified. Any changes made to cart instance will be disregarded. Only the cart Id and ShopName are considered for retrieving and modifying the cart.
	IEnumerable<CartLine> CartLines – Required - A collection of cart lines to add
Output:	
	Cart - Cart object that represent the updated cart in the external system.
	SystemMessages - Collection of messages from the external system

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Prepare parameters for getting cart for visitor ID Ivan in shop Autohaus
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Ivan");

// Get a cart, new or existing
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

// Create cart line with subline to add to the cart
var cartLines = new ReadOnlyCollection<CartLine>(new Collection<CartLine>
{
    new CartLine
    {
        Product = new CartProduct
        {
            ProductId = "Audi",
            Price = new Price(10000, "USD") },
    Quantity = 1,
    SubLines = new Collection<CartLine>
            {
    new CartLine
            {
                Product = new CartProduct
                {
                    ProductId = "Winter Tyres",
                    Price = new Price(100, "USD")
                },
                Quantity = 4
            },
    new CartLine
            {
                Product = new CartProduct
                {
                    ProductId = "Summer Tyres",
                    Price = new Price(80, "USD")
                },
                Quantity = 4
```

```
                    }
                }
            }
        }
    }
);

// Create request with prefix and prefix lines
var request = new AddCartLinesRequest(cart, cartLines);

// Add prefix lines into prefix
var result = cartServiceProvider.AddCartLines(request);
var resultCart = result.Cart;
```

RemoveCartLines

Name:	RemoveCartLines
Description:	Responsibility is to remove lines from cart.
Usage:	Called when one or more cart lines are about to be removed from the shopping cart. UI wise when the user updates the cart by removing one or more lines.
Signature:	RemoveCartLinesResult RemoveCartLines(RemoveCartLinesRequest request)
Input:	
	Cart - Required. The cart must be unmodified. Any changes made to cart instance will be disregarded. Only the cart Id and ShopName are considered for retrieving and modifying the cart.
	IEnumerable<CartLine> CartLines – Required - A collection of cart lines to remove. ExternalCartLineId, LineNumber or object reference can be used to identify the line(s) to be removed. The default Connect based implementation removes lines by object reference. Typically
Output:	
	Cart - Cart object that represent the updated cart in the external system.
	SystemMessages - Collection of messages from the external system

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create cart with "Audi Q10", "BMW X7" and "Citroen C3"
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "John");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

var cartLine1 = new CartLine
{
    Quantity = 1,
    Product = new CartProduct
    {
        ProductId = "Audi Q10",
        Price = new Price(55, "USD")
    }
};

var cartLine2 = new CartLine
{
    Quantity = 2,
    Product = new CartProduct
    {
        ProductId = "BMW X7",
        Price = new Price(10, "USD")
    }
};

var cartLine3 = new CartLine
{
    Quantity = 1,
    Product = new CartProduct
```

```
            {
                ProductId = "Citroen C3",
                Price = new Price(25, "USD")
            }
        };

        var cartLines = new Collection<CartLine>
        {
            cartLine1, cartLine2, cartLine3
        };

        var addCartLinesRequest = new AddCartLinesRequest(cart, cartLines);
        cart = cartServiceProvider.AddCartLines(addCartLinesRequest).Cart;

        // Create request to remove cart line "BMW X7".
        var request = new RemoveCartLinesRequest(cart, cart.Lines.Where(l =>
l.Product.ProductId == "BMW X7").ToArray());

        // Call service provider and receive the result.
        var result = cartServiceProvider.RemoveCartLines(request);
```

UpdateCartLines

Name:	**UpdateCartLines**
Description:	Responsibility is to update lines on cart.
Usage:	Occurs when a shopping cart is about to be updated referring to lines already in the cart. UI wise it is when the user updates the cart regarding a specific product. Most typically it is when
	• The quantity is changed
	• A service is added like insurance or shipping
	• Promotion code is added for a given product
	• Product is replaced with another variant
Signature:	UpdateCartLinesResult UpdateCartLines(UpdateCartLinesRequest request)
Input:	
	Cart - Required. The cart must be unmodified. Any changes made to cart instance will be disregarded. Only the cart Id and ShopName are considered for retrieving and modifying the cart.
	IEnumerable<CartLine> CartLines – Required - A collection of cart lines to update on cart
Output:	
	Cart – Cart object that represent the updated cart in the external system.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
        var cartServiceProvider = new CartServiceProvider();

        // Create cart with "Audi Q10", "BMW X7" and "Citroen C3"
        var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "John");
        var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

        var cartLine1 = new CartLine
        {
            Quantity = 1,
            Product = new CartProduct
            {
                ProductId = "Audi Q10",
                Price = new Price(55, "USD")
            }
        };

        var cartLine2 = new CartLine
        {
```

```
        Quantity = 2,
        Product = new CartProduct
        {
            ProductId = "BMW X7",
            Price = new Price(10, "USD")
        }
    };

    var cartLines = new Collection<CartLine> { cartLine1, cartLine2 };

    var addCartLinesRequest = new AddCartLinesRequest(cart, cartLines);
    cart = cartServiceProvider.AddCartLines(addCartLinesRequest).Cart;

    var bmw = cart.Lines.First(i => i.Product.ProductId == "BMW X7");
    bmw.Product.Price = new Price(110000, "USD");
    bmw.Quantity = 3;

    // Create request to update cart lines.
    var updateCartLinesRequest = new UpdateCartLinesRequest(cart, new Collection<CartLine>
{ bmw });

    // Call service provider and receive the result.
    var result = cartServiceProvider.UpdateCartLines(updateCartLinesRequest);
```

DeleteCart

Name:	**DeleteCart**
Description:	Responsibility is to delete a cart permanently: • The cart is deleted. • Trigger event in DMS telling the cart is deleted.
Usage:	Must be called when a cart needs to be deleted. UI wise this could be • When the user has gone through the B2C scenario of paying and an order has been created and registered. • After a cart has been in abandoned state for a given time and the visitor is not expected to return.
Signature	`DeleteCartResult DeleteCart(DeleteCartRequest request)`
Input:	
	Cart - Required. The cart must be unmodified. Any changes made to cart instance will be disregarded. Only the cart Id and ShopName are considered for retrieving and modifying the cart.
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var cartServiceProvider = new CartServiceProvider();

    // Create the request.
    var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Bred");

    // Call the service provider to get the cart
    var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

    // Read the id of the returned cart
    var id = cart.ExternalId;

    // Create request to remove the cart.
    var deleteCartRequest = new DeleteCartRequest(cart);

    // Call the service provider and receive the result.
    cartServiceProvider.DeleteCart(deleteCartRequest);
```

UpdateCart

Name:	UpdateCart
Description:	Responsibility is to pass an updated cart to the external commerce system Trigger event in DMS telling the cart is being updated.
Usage:	The method should be executed after any operation that modifies the cart, typically when Adjustments have been added, removed or modified.
Signature:	`UpdateCartResult UpdateCart(UpdateCartRequest request)`
Input:	
	Cart - Required - The cart to be updated. The cart must be unmodified. Any changes made to cart instance will be disregarded. Only the cart Id and ShopName are considered for retrieving and modifying the cart.
	Cart Base – An instance of Cart Base containing the changes to be made to the cart Typically the only properties allowed to be modified would be: UserId, CustomerId, CartName and potentially ShopName. Whether IsLocked and CartStatus will be considered, depends on business logic in the external commerce system. Null values will not be considered, but blank values will
Output:	
	Cart – Cart object that represent the updated cart in the external system.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create prefix.
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Peter");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

// Create the instance of the CartBase with properties that should be changed in
existing prefix.
var changes = new CartBase
{
    CustomerId = "Customer Peter",
    Name = "Peter's Cart",
    ShopName = "Autohaus"
};

// Create request to update the prefix.
var updateCartRequest = new UpdateCartRequest(cart, changes);

// Call service provider with prepared request and receive the result.
var result = cartServiceProvider.UpdateCart(updateCartRequest);
```

LockCart

Name:	LockCart
Description:	Responsibility is to set the cart in a locked state where it is ready to be committed to an order but before any optional payment transaction is performed: • Set cart to locked and save it. • Trigger event in DMS telling the cart is in locked state. When cart is in locked state it indicates two things: • It's *not* possible to modify the shopping cart content using the other service layer methods • It's easy to identify the carts in locked state in order to compare and confirm with payment transactions whether there are carts that have not

	been finalized due to some unforeseen incident in the checkout process. There is a corresponding UnlockCart method. If the cart is locked when LockCart is called, the pipeline is aborted and nothing happens.
Usage:	Is typically executed during the checkout process, just before any payment transaction is about to be executed and before turning the cart into an order. UI wise its triggered when a user in the checkout flow has selected "confirm" and in a B2C scenario is going to pay and the order is created.
Signature:	`LockCartResult LockCart(LockCartRequest request)`
Input:	
	Cart - Required
Output:	
	Cart – Cart object that represent the updated cart in the external system.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create a sample cart.
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Jho");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

// Create request to lock this cart.
var lockCartRequest = new LockCartRequest(cart);

// Call service provider and receive the result.
var result = cartServiceProvider.LockCart(lockCartRequest);
```

UnlockCart

Name:	**UpdateCartLines**
Description:	Responsibility is to set the cart in an unlocked state: • Set cart to unlocked and save it. • Trigger event in DMS telling the cart is in locked state. By default a cart is in unlocked state and can be edited There is a corresponding LockCart method, which sets the state to locked. If the cart is not locked when UnlockCart is called, the pipeline is aborted and nothing happens.
Usage:	Is typically called if user returns to cart and decides to modify the content after starting the checkout process and payment transaction, but its implementation specifies how it should be handled.
Signature:	`UnlockCartResult UnlockCart(UnlockCartRequest request)`
Input:	
	Cart – Required
Output:	
	Cart – Cart object that represent the updated cart in the external system.
	SystemMessages - Collection of messages from the external system. This is how error conditions can be reported.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

// Create cart and lock it.
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "Mark");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;
var lockCartRequest = new LockCartRequest(cart);
cart = cartServiceProvider.LockCart(lockCartRequest).Cart;
```

```
// Create request to unlock this cart.
var unlockCartRequest = new UnlockCartRequest(cart);

// Call service provider and receive the result.
var result = cartServiceProvider.UnlockCart(unlockCartRequest);
```

MergeCart

Name:	MergeCart
Description:	Responsibility is to merge two specified carts: • Both carts must have identical shop names. • Both carts must have different ExternalIDs. • Cart1 and Cart2 cart line items are merged and returned
Usage:	Is typically called when a user logs in and the system notices an anonymous cart exists.
Signature:	CartResult MergeCart([NotNull] MergeCartRequest request)
Input:	
	UserCart – Required **AnonymousCart - Required**
Output:	
	Cart – Cart object representing the merged user cart.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

var userCart = new Cart
{
    ExternalId = "0",
    ShopName = "first shop",
    Lines  = new List<CartLine>
    {
        new CartLine
        {
            Quantity = 1,
            Product = new CartProduct
            {
                ProductId = "Audi Q10",
                Price = new Price(55, "USD")
            }
        }
    }
};

var anonymousCart = new Cart
{
    ExternalId = "1",
    ShopName = "first shop",
    Lines  = new List<CartLine>
    {
        new CartLine
        {
            Quantity = 1,
            Product = new CartProduct
            {
                ProductId = "BMW M5",
                Price = new Price(75, "USD")
            }
        }
    }
};

var request = new MergeCartRequest(userCart, anonymousCart);

var mergeredCart = cartServiceProvider.MergeCart(request).Cart;
```

```
cartServiceProvider.DeleteCart(new DeleteCartRequest(anonymousCart));
```

AddParties

Name:	AddParties
Description:	Responsibility is to add parties to a cart
Usage:	Is typically called when adding party information to a cart
Signature:	AddPartiesResult AddParties([NotNull] AddPartiesRequest request)
Input:	
	Cart – Required **Parties -- Required**
Output:	
	Parties – The read only list of all parties associated with this cart after the add

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "John");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

var partyList = new List<Party>
{
    new Party
    {
        PartyId = "123", FirstName = "Joe", LastName = "Smith",
        Address1 = "123 Street", City = "Ottawa",
        State = "Ontario", Country = "Canada"
    },
    new Party
    {
        PartyId = "456", FirstName = "Jane", LastName = "Smith",
        Address1 = "234 Street", City = "Toronto",
        State = "Ontario", Country = "Canada"
    }
};

var addPartiesRequest = new AddPartiesRequest(cart, partyList);

var addPartiesResult = cartServiceProvider.AddParties(addPartiesRequest);
```

RemoveParties

Name:	RemoveParties
Description:	Responsibility is to remove parties from a cart
Usage:	Is typically called when removing party information
Signature:	RemovePartiesResult RemoveParties([NotNull] RemovePartiesRequest request)
Input:	
	Cart – Required **Parties – Required –** The list of parties to remove from the cart
Output:	
	Parties – The read only list of all parties associated with this cart after the remove

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();
```

```
var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "John");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

var partyList = new List<Party>
{
    new Party
    {
        PartyId = "123", FirstName = "Joe", LastName = "Smith",
        Address1 = "123 Street", City = "Ottawa",
        State = "Ontario", Country = "Canada"
    },
    new Party
    {
        PartyId = "456", FirstName = "Jane", LastName = "Smith",
        Address1 = "234 Street", City = "Toronto",
        State = "Ontario", Country = "Canada"
    }
};

var addPartiesRequest = new AddPartiesRequest(cart, partyList);
var addPartiesResult = cartServiceProvider.AddParties(addPartiesRequest);

var removePartiesRequest = new RemovePartiesRequest(cart, new List<Party>
{
    partyList[0]
});
var removePartiesResult = cartServiceProvider.RemoveParties(removePartiesRequest);
```

UpdateParties

Name:	**UpdateParties**
Description:	Responsibility is to update a list of parties within a cart
Usage:	Is typically called when parties need to be updated
Signature:	`UpdatePartiesResult UpdateParties([NotNull] UpdatePartiesRequest request)`
Input:	
	Cart – Required **Parties – Required –** The list of parties to update in the cart
Output:	
	Parties – The read only list of all parties associated with this cart after the update

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();

var createCartRequest = new CreateOrResumeCartRequest("Autohaus", "John");
var cart = cartServiceProvider.CreateOrResumeCart(createCartRequest).Cart;

var party1 = new Party
{
    PartyId = "123",
    FirstName = "Joe",
    LastName = "Smith",
    Address1 = "123 Street",
    City = "Ottawa",
    State = "Ontario",
    Country = "Canada"
};

var party2 = new Party
{
    PartyId = "456",
    FirstName = "Jane",
    LastName = "Smith",
    Address1 = "234 Street",
    City = "Toronto",
    State = "Ontario",
```

```
        Country = "Canada"
};

var partyList = new List<Party> { party1, party2 };

var addPartiesRequest = new AddPartiesRequest(cart, partyList);
var addPartiesResult = cartServiceProvider.AddParties(addPartiesRequest);

party1.Address1 = "678 Road";
party1.City = "London";

var updatePartiesRequest = new UpdatePartiesRequest(cart, new List<Party> { party1 });
var removePartiesResult = cartServiceProvider.UpdateParties(updatePartiesRequest);
```

AddPaymentInfo

Name:	**AddPaymentInfo**
Description:	Responsibility is to add payment information to a cart
Usage:	Is typically called during a checkout flow to add the payment info for processing of an order.
Signature:	AddPaymentInfoResult AddPaymentInfo([NotNull] AddPaymentInfoRequest request)
Input:	
	Cart – Required **Payments – Required –** a list of payment info to be added to the cart
Output:	
	Payments – The read only list of payments associated with the cart after the add

Usage Example:

```
var cartService = new CartServiceProvider();
var cart = cartService.CreateOrResumeCart(new
    CreateOrResumeCartRequest("MyShop", "Me")).Cart;

var paymentList = new List<PaymentInfo>
{
    new PaymentInfo() { ExternalId = "1", PaymentMethodID = "1"},
    new PaymentInfo() { ExternalId = "2", PaymentMethodID = "2"}
};

var addPaymentRequest = new AddPaymentInfoRequest(cart, paymentList);

var addPaymentResult = cartService.AddPaymentInfo(addPaymentRequest);
```

RemovePaymentInfo

Name:	**RemovePaymentInfo**
Description:	Responsibility is to remove payment information from a cart
Usage:	Is typically called when a user wants to change their payment information
Signature:	RemovePaymentInfoResult RemovePaymentInfo([NotNull] RemovePaymentInfoRequest request)
Input:	**Cart – Required** **Payments – Required –** a list of payment info to be removed from the cart
Output:	**Payments** – The read only list of payments associated with the cart after the remove

Usage Example:

```
var cartService = new CartServiceProvider();
```

```
var cart = cartService.CreateOrResumeCart(new
    CreateOrResumeCartRequest("MyShop", "Me")).Cart;

var paymentList = new List<PaymentInfo>
{
    new PaymentInfo() { ExternalId = "1", PaymentMethodID = "1"},
    new PaymentInfo() { ExternalId = "2", PaymentMethodID = "2"}
};

var addPaymentRequest = new AddPaymentInfoRequest(cart, paymentList);

var addPaymentResult = cartService.AddPaymentInfo(addPaymentRequest);

var removePaymentRequest = new RemovePaymentInfoRequest(cart, new
List<PaymentInfo> {paymentList[0]});
var removeResult = cartService.RemovePaymentInfo(removePaymentRequest);
```

AddShippingInfo

Name:	
Description:	Responsibility is to add shipping information to a cart
Usage:	Is typically called during a checkout flow to add the shipping info for processing of an order.
Signature:	AddShippingInfoResult AddShippingInfo([NotNull] AddShippingInfoRequest request)
Input:	
	Cart – Required **ShippingInfo – Required –** a list of shipping info to add to the cart
Output:	
	ShippingInfo – A read only list of shipping info associated with the cart after the add

Usage Example:

```
var cartService = new CartServiceProvider();
var cart = cartService.CreateOrResumeCart(new
    CreateOrResumeCartRequest("MyShop", "Me")).Cart;

var shippingList = new List<ShippingInfo>
{
    new ShippingInfo() { ExternalId = "1", ShippingMethodID = "1"},
    new ShippingInfo() { ExternalId = "2", ShippingMethodID = "2"}
};

var addRequest = new AddShippingInfoRequest(cart, shippingList);

var addResult = cartService.AddShippingInfo(addRequest);
```

RemoveShippingInfo

Name:	
Description:	Responsibility is to remove shipping information from a cart
Usage:	Is typically called during a checkout flow to remove the shipping info for processing of an order.
Signature:	RemoveShippingInfoResult RemoveShippingInfo([NotNull] RemoveShippingInfoRequest request)
Input:	
	Cart – Required **ShippingInfo – Required –** a list of shipping info to remove from the cart
Output:	

ShippingInfo – A read only list of shipping info associated with the cart after the remove

Usage Example:

```
var cartService = new CartServiceProvider();
var cart = cartService.CreateOrResumeCart(new
    CreateOrResumeCartRequest("MyShop", "Me")).Cart;

var shippingList = new List<ShippingInfo>
{
    new ShippingInfo() { ExternalId = "1", ShippingMethodID = "1"},
    new ShippingInfo() { ExternalId = "2", ShippingMethodID = "2"}
};

var addRequest = new AddShippingInfoRequest(cart, shippingList);

var addPaymentResult = cartService.AddShippingInfo(addRequest);

var removeRequest = new RemoveShippingInfoRequest(cart, new
List<ShippingInfo> {shippingList[0]});
var removeResult = cartService.RemoveShippingInfo(removeRequest);
```

2.1.3   Cart Pipelines

The integration and engagement logic used in the Cart API is implemented by pipelines that can be customized as needed. There is a pipeline for each method on the API. Some pipelines call other common pipelines like SaveCart and LoadCart. In some cases the logic is split into several sub-pipelines to handle if-than-else situations like used in CreateOrResumeCart.

GetCarts

Name:	GetCarts
Description:	The pipeline is responsible for performing a search against all carts and return a list of CartBase instances for carts found, matching the specified search criteria.
	The carts might be persisted in both the ECS and/or EA state.
	Depending on the location of the main cart repository, the processors configured for the pipeline will differ between different Connect provider implementations.
Usage:	Called via method GetCarts on the Connect API when searching for carts.
Args:	
	Request - Contains the search criteria: UserID, CustomerID, CartName, CartStatus, IsLocked, and ShopName. Is set prior to calling the pipeline.
	Response - Contains the cart objects. Is read after the pipeline is executed.
Processors:	**GetCartsFromEAState –**
	Responsibility - To execute query against carts stored in EA states across all users which matches the input parameters UserID, CustomerID, CartName, and ShopName.
	In default implementation, the processor is split into two, named:
	• BuildQuery.
	• ExecuteQuery. Queries the EA repository

Usage: The processor is generic for searching in EA state and returning lists of carts. *If the external commerce system supports persisting carts then the processor(s) should be replaced by the custom GetCarts processor querying against ECS, see below.*

Ownership: The processor is provided with Connect

Customization: If more search parameters are needed then the processor should be inherited and overwritten to search in

---

GetCarts –

Responsibility - To execute a query against carts stored in the external commerce system which matches the input parameters UserID, CustomerID, CartName, and ShopName.

Usage: If ECS supports persistence of carts, then this processor replaces the default Connect processor(s) querying against EA state

If not supported, EA state and GetCartsFromEAState processor(s) must be used instead

Ownership: ECS vendor

Customization: The processor is custom to the ECS. If more search parameters are needed then the processor should be extended to support that.

CreateOrResumeCart

Name:	**CreateOrResumeCart**
Description:	Initiates the creation of a shopping cart and in the process to: • Load persisted, potentially abandoned cart, if present and return that • Trigger event in DMS if new cart is created • Enter user in Engagement Automation plan with ID of new shopping cart The pipeline will call different pipelines depending on whether an existing cart is found and can be resumed or not.
Usage:	Called via method CreateOrResumeCart on the Connect API.
Args:	
	Request - Contains the essential cart parameters also used to search for existing cart. Is set prior to calling the pipeline.
	Response - Contains the cart objects or a list of CartBase instance in case multiple carts exists that matches the given parameters. Is read by the Cart Provider after the pipeline is called.

Processors:

FindCartInEAState –

Responsibility: To locate a cart in the current contact's / visitor's EA state which matches the input parameters UserID, CustomerID, CartName, and ShopName.

If a match is found then the Cart ID is set in custom pipeline argument CartID (args.CustomData["CartID"])

The processor is needed to retrieve the cart ID of the existing cart before LoadCart can be called

Usage: The processor is generic for searching in EA state and returning cart ID for a cart matching the given input parameters.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality, unless further query parameters are introduced

RunLoadCart –

Responsibility: Is to call pipeline LoadCart and load a cart with given ID. The ID is specified in parameter CartId.

If cart was not found, cart is null or empty in pipeline arguments

Usage: Mandatory. The processor is generic for calling pipeline LoadCart that loads a cart by CartID and can be used in other pipelines.

It's assumed that the CartID to be used for loading cart is stored in customer pipeline arg "CartID" (args.CustomData["CartID"])

Pipeline RunResumeCart is assuming that the cart to be resumed is loaded before it is executed

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality.

RunResumeCart –

Responsibility: Is to call pipeline ResumeCart in order to resume a loaded cart if possible.

It's expected that a cart is present in pipeline arguments.

If no cart is present, then the processor will not call pipeline ResumeCart

Usage: Mandatory. The processor is generic for calling pipeline ResumeCart

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality.

RunCreateCart –

Responsibility: Is to call pipeline CreateCart in order to create a new cart.

It's expected that no cart is present in pipeline arguments.

If a cart is present, then the processor will not call pipeline CreateCart

Usage: Mandatory. The processor is generic for calling pipeline CreateCart

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

CreateCart

Name:	CreateCart
Description:	Creates and saves a new shopping cart and in the process: • Trigger event in DMS for new cart • Enter visitor / contact into Engagement Automation plan with new shopping cart
Usage:	Called from pipeline CreateOrResumeCart as one of the branches when no cart exists
Args:	
	Request - Contains the same parameters as CreateOrResumeCart pipeline. Is carried over from CreateOrResumeCart pipeline.
	Response - Contains the cart objects. Is read by the Cart Provider after the pipeline is called.
Processors:	
	CreateCart – Responsibility: Is to create a new cart, initialize values from arguments and return it. If a cart already exists in pipeline arguments, it is ignored and will be overwritten. Usage: Optional. The default processor is generic for creating a domain model cart and initialize values, but it never calls the ECS. In some integration scenarios it will be relevant to call the ECS at this point in time to create a new cart. In that case this processor can be used as a base

class and extended, as it does initialize the cart domain object, or it can be replaced altogether.

In other integration scenarios the ECS will not be called at this point in time. Instead it will likely be called when manipulating cart lines or only when saving the cart. It depends on the system being integrated with.

Ownership: The processor is provided with Connect

Customization: It can be useful to use the processor as a base class and extended it by calling the ECS after initializing the cart domain model object,

AddVisitorToEAPlan –

Responsibility: Is to add the current visitor to an EA plan, in this case the Abandoned Cart EA plan.

The current implementation supports multiple shops. By using an eaPlanProvider the plan and state IDs can be retrieved. By default the eaPlanProvider tries to find an EA plan with a name prefixed with the site name. For example, the ECS could contain a site named Autohaus. According to naming convention, the EA plan must be named "Autohaus Abandoned Cart". If a different EA plan name is used, it can be configured on the site definition item.

Usage: Optional. The processor is generic for adding the visitor to an EA plan, but it expects to be in the CreateCart pipeline with the CreateCartRequest object in pipeline args.

The plan and state IDs are retrieved from the eaPlanProvider, which is specified in the processor parameter and instantiated via Sitecore Factory.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

RunSaveCart –

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality.

TriggerPageEventForResultCart –

Responsibility: Is to trigger a specified page event and register the cart values: ExternalCartId, UserId, CartName, CartStatus

The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event with the specified parameters from the cart stored in the args.Result argument.

The event to be triggered is specified in processor parameter PageEventName

The event text to be used is specified in processor parameter PageEventText

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered

ResumeCart

Name:	**CreateOrResumeCart**
Description:	Validates and resumes the cart specified in arguments and in the process: - Change the state to the initial state in abandoned cart Engagement (reboot) - Updates the cart state - Saves the cart - Trigger event in DMS
Usage:	Called via the Connect API from Sitecore.
Args:	
	Request - Contains the search criteria. Is set by the Cart Provider prior to calling the pipeline.
	Response - Contains the cart objects. Is read by the Cart Provider after the pipeline is called.
Processors:	
	CheckCanBeResumed – Responsibility: Is to check if given cart can be resumed. If visitor is not already in the abandoned cart EA plan the pipeline is aborted and if the visitor is already in the initial state, it is also aborted. Also sets three parameters in pipeline args that are used in the processors listed in parentheses: - CartSourceStateId (MoveVisitorToInitialState) - CartDestinationStateId (MoveVisitorToInitialState) - PreviousStateName (TriggerPageEventForResumedCart)

Usage: Mandatory. The processor checks if the visitor is already in the plan or not. If not, there is no cart to resume.

Ownership: The processor is provided with Connect

Customization: No obvious customization needed

ChangeCartStatus –

Responsibility: Is to update the cart status field with the value "InProcess" taken from constant CartStatus.InProcess

Usage: Optional. The processor is generic but the status value set on cart is fixed to InProcess.

Ownership: The processor is provided with Connect

Customization: The processor could be updated to take the state value from parameter instead of constant in code.

MoveVisitorToInitialState –

Responsibility: Is to move the current visitor between two states of an EA plan, in this case the Abandoned Cart EA plan.

Usage: Optional. The processor is generic for moving a visitor to initial state of EA plan, but it expects to be in the CreateCart pipeline with the CreateCartRequest object in pipeline args.

The source and destination state IDs are read from custom pipeline arguments:

- "CartSourceStateId"
- "CartDestinationStateId"

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

RunSaveCart –

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Mandatory. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality.

TriggerPageEventForResumedCart –

Responsibility: Is to trigger a specified page event when resuming the cart and register the cart values:

- ExternalCartId, UserId, CartName, CartStatus, StateName (containing the previous state that it was moved from)

The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is specific for resuming a cart and for triggering a page event with the specified parameters from the cart stored in the args.Result argument.

The value for previous state is found in pipeline args "PreviousState" and is inserted in the page event text along with the ExternalCartId

The event text to be used is specified in processor parameter PageEventText

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

LoadCart

Name:	**LoadCart**
Description:	Loads the cart that matches the specified criteria. For example, ID and ShopName.
	This pipeline is responsible for reading data for a specific cart that is managed by the commerce system. This pipeline reads the cart data from the commerce system and/or from Engagement Automation state.
Usage:	1. Called directly via the Connect API method LoadCart from Sitecore.
	2. Called indirectly via the Connect API methods CreateOrResumeCart
Args:	
	Request - Contains the criteria that determine which cart should be retrieved. Is set prior to calling the pipeline.
	Response - Contains the cart object after the pipeline is called.
Processors:	
	LoadCartFromEAState –

Responsibility:

Load existing cart from EA state with given CartID and in the shop specified with parameter ShopName

For performance reasons, the default implementation works as follows:

- First carts are loaded from current visitor / contact EA state data and filtered by the given parameters. If found, it is written to pipeline result args and execution stops.

- If not found, then all carts across all contacts are loaded and filtered by the given parameters. Searching across all contacts is an expensive operation.

- If cart is found in EA state, it is written to pipeline Result arg.

Usage: The processor is generic for loading a cart from EA state.

The processor might be used alone in pipeline LoadCart, if cart persistence is not available in the ECS or if handling of carts occurs in Sitecore alone.

It might also be left out of the pipeline if the ECS manage the cart repository alone.

Some ECS systems does not provide all the information specified in the cart domain model and a hybrid configuration might be used where the main data is read from the ECS and augmented with additional cart data stored in EA state by inserting an additional processor.

Abandoned carts might be purged from the ECS but still remain in EA state in Sitecore. In that case it might make sense to have both processors in the pipeline. Sitecore will then act as a backup storage for carts.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

LoadCart –

Responsibility:

Load existing cart from ECS with given CartID and in the shop specified with parameter ShopName

Usage: The processor is specific for loading a cart from ECS.

The processor might be used alone in pipeline LoadCart if cart persistence is available in the ECS. See also scenarios in description for LoadCartFromEAState processor.

Some ECS systems does not provide all the information specified in the cart domain model and a hybrid configuration might be used where the main data is read from the ECS and augmented with additional cart data stored in EA state by inserting an additional processor.

If used in combination with LoadCartFromEAState, even though the cart might already be loaded from EA state, it is important to check with the ECS and load cart from there to ensure the latest version is used.

If cart is found in external commerce system it can overwrite or merge with the cart already stored in pipeline Response arg. It's up to the Connect provider implementation with the ECS.

Ownership: The processor is provided by the ECS

Customization: Must be built specifically for the ECS.

SaveCart

Name:	SaveCart
Description:	Saves the cart both to the external commerce system and in Engagement Automation state.
Usage:	Called from other service layer methods, but rarely, if never called explicitly
Args:	
	Request - Contains the criteria that determine which cart should be retrieved. Is set prior to calling the pipeline.
	Response - Contains the cart object after the pipeline is called.
Processors:	
	SaveCart – Responsibilitty: Saves the given cart to the ECS Usage: The processor is specific for saving a cart to the ECS. The processor should not be used alone in pipeline SaveCart because the feature for resuming existing carts is depending on the cart being stored in EA state as well. The processor FindCartInEAState is looking up the cart in EA state in order to get the CartId for loading the cart with LoadCart from ECS: Some ECS systems does not provide all the information specified in the cart domain model and a hybrid configuration might be used where the main data is saved to the ECS and the additional cart data is stored in EA state by. In some ECS implementations the SaveCart pipeline is the first and only place the ECS system is provided a cart from Connect. Since the ECS is the primary repository for carts, it is assumed that the unique CartID key is provided by the ECS. Ownership: The processor is provided by the ECS Customization: Must be built specifically for the ECS.

SaveCartToEAState –

Responsibility:

Saves the cart to EA state

Usage: Mandatory. The processor is generic for saving a cart to EA state.

The processor might be used alone in pipeline LoadCart, if cart persistence is not available in the ECS or if handling of carts occurs in Sitecore alone.

The processor should be used in pipeline SaveCart because the feature for resuming existing carts is depending on the cart being stored in EA state as well. The processor FindCartInEAState is looking up the cart in EA state in order to get the CartId for loading the cart with LoadCart from ECS:

Some ECS systems does not provide all the information specified in the cart domain model and a hybrid configuration might be used where the main data is read from the ECS and augmented with additional cart data stored in EA state by inserting an additional processor.

Abandoned carts might be purched from the ECS but still remain in EA state in Sitecore. In that case it might make sense to have both processors in the pipeline. Sitecore will then act as a backup storage for carts.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

.

AddCartLines

Name:	**AddCartLines**
Description:	This pipeline is responsible for adding a new line to the shopping cart and recording a corresponding page event in DMS. This happens when a product is added to the cart.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the cart to be updated, along with what lines should be added to the cart.
	Is set prior to calling the pipeline.
	Response - Contains the updated cart object after the pipeline is called.
Processors:	
	CheckIfLocked –
	Responsibility: Checks if the cart is locked and abort the pipeline if so, returning SystemMessages to signal the locked state

Usage: Optional. The processor is generic for checking if cart is locked. The processor is and should be used in all pipelines that potentially modify the cart content

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

AddLinesToCart –

Responsibility: Adds the given lines to the provided cart.

Calling this method will always add the given lines to the existing collection of lines in the cart, even if lines already exist on the cart containing the same product. Alternatively UpdateLinesOnCart can be called if a line already exists with a product where the quantity simply should be adjusted.

Usage: Optional. The processor is generic for adding lines to cart, but potentially should be replaced by an ECS specific implementation.

Ownership: The processor is provided with Connect

Customization: The default implementation operates on the cart domain model in Sitecore only.

In most ECS integrations it will be relevant to inherit and overwrite or replace this implementation and call the ECS, so that the changes to cart are passed on and any business logic can be applied.

RunSaveCart -

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

TriggerPageEventsForCartLines –

Responsibility: Is to trigger a specified page event when adding lines to cart and register the values:

- Product ID, QTY, Price and Currency

The event to be triggered is passed in as parameter as well as the Page Event Text.

In this case the event is "Lines Added To Cart". The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event when modifying cart lines

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

RemoveCartLines

Name:	RemoveCartLines
Description:	Responsibility is to remove cart lines from cart
Usage:	Called from Sitecore.
Args:	
	Request - Contains the cart to be updated along with the cart lines to be removed
	Is set prior to calling the pipeline.
	Response - Contains the updated cart object after the pipeline is called.
Processors:	

CheckIfLocked –

Responsibility: Checks if the cart is locked and abort the pipeline if so, returning an SystemMessages to signal the locked state

Usage: Optional. The processor is generic for checking if cart is locked. The processor is and should be used in all pipelines that potentially modify the cart content

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

RemoveLinesFromCart –

Responsibility: Removes the given lines from the provided cart.

The list of lines to remove must be references directly to the lines in the CartLines collection on the cart. They will be removed by reference: cart.CartLines = cart.CartLines.Except(request.CartLines).ToList();

Usage: Optional. The processor is generic for removing lines from cart, but potentially should be replaced by an ECS specific implementation.

Ownership: The processor is provided with Connect

Customization: The default implementation operates on the cart domain model in Sitecore only.

In most ECS integrations it will be relevant to inherit and overwrite or replace this implementation and call the ECS, so that the changes to cart are passed on and any business logic can be applied.

SaveCart –

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

TriggerPageEventsForCartLines –

Responsibility: Is to trigger a specified page event when adding lines to cart and register the values:

- Product ID, QTY, Price and Currency

The event to be triggered is passed in as parameter as well as the Page Event Text. In this case the event is "Lines Removed From Cart". The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event when modifying cart lines

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

UpdateCartLines

Name:	UpdateCartLines
Description:	Responsibility is to update lines on cart
Usage:	Called from Sitecore.
Args:	
	Request - Contains the cart along with the cart lines to be updated
	Is set prior to calling the pipeline.
	Response - Contains the updated cart object after the pipeline is called.
Processors:	
	CheckIfLocked – Responsibility: Checks if the cart is locked and abort the pipeline if so, returning an SystemMessages to signal the locked state Usage: Optional. The processor is generic for checking if cart is locked. The processor is and should be used in all pipelines that potentially modify the cart content Ownership: The processor is provided with Connect Customization: No immediate need for overwriting the default functionality
	UpdateLinesOnCart – Responsibility: To update the given lines on the provided cart. The default implementation doesn't process the lines on the original cart, but simple return the cart that was passed in and that already had its Cartlines updated. Hence, the list of updated lines must be references to the CartLines collection on the cart. Usage: Optional. The processor is generic for updating lines on cart, but potentially should be replaced by an ECS specific implementation. Ownership: The processor is provided with Connect Customization: The default implementation operates on the cart domain model in Sitecore only. In most ECS integrations it will be relevant to inherit and overwrite or replace this implementation and call the ECS, so that the changes to cart are passed on and any business logic can be applied.

SaveCart –

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

TriggerPageEventsForCartLines –

Responsibility: Is to trigger a specified page event when adding lines to cart and register the values:

- Product ID, QTY, Price and Currency

The event to be triggered is passed in as parameter as well as the Page Event Text. In this case the event is "Lines Updated On Cart". The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event when modifying cart lines

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

DeleteCart

Name:	DeleteCart
Description:	Responsibility is to delete a cart permanently: • The cart is deleted. • Trigger event in DMS telling the cart is deleted.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the cart to be deleted Is set prior to calling the pipeline.
	Response – SystemMessages

Processors:

CheckIfLocked –

Responsibility: Checks if the cart is locked and abort the pipeline if so, returning an SystemMessages to signal the locked state

Usage: Optional. The processor is generic for checking if cart is locked. The processor is and should be used in all pipelines that potentially modify the cart content

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

DeleteCart –

Responsibility: Deletes and removes cart from storage in ECS

Usage: Mandatory

Ownership: The processor is provided by the ECS

Customization: Must be built specifically for the ECS.

DeleteCartFromEAState –

Responsibility: Locates and deletes cart from EA state

Usage: Optional. The processor is generic for deleting cart in EA state.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

TriggerPageEventForRequestCart –

Responsibility: Is to trigger a specified page event when deleting a cart and register the values:

- ExternalCartId, UserId, CartName, CartStatus

The event to be triggered is passed in as parameter as well as the Page Event Text. In this case the event is "Cart Deleted". The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event for processors that takes an argument based on parameter type CartRequest

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

UpdateCart

Name:	UpdateCart
Description:	Responsibility is to pass an updated cart to the external commerce system
	Trigger event in DMS telling the cart is being updated.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the cart and the data to be updated in Cart Base
	Is set prior to calling the pipeline.
	Response - Contains the updated cart object after the pipeline is called.
Processors:	
	CheckIfLocked – Responsibility: Checks if the cart is locked and abort the pipeline if so, returning an SystemMessages to signal the locked state Usage: Optional. The processor is generic for checking if cart is locked. The processor is and should be used in all pipelines that potentially modify the cart content Ownership: The processor is provided with Connect Customization: No immediate need for overwriting the default functionality
	UpdateCart – Responsibility: To update the cart with the updated values of the cart body (CartBase object only, not lines etc.). The default implementation will update all default properties on CartBase except "CustomerID"; "CartName", "ShopName". Everything including null and black values are included. Usage: Optional. The processor is generic for updating the cart body.

Ownership: The processor is provided with Connect

Customization: In case the default Connect domain model is customized, the processor should be overwritten to include the customized properties.

TriggerPageEventForRequestCartChanges –

Responsibility: Is to trigger a specified page event when deleting a cart and register the values:

- CustomerId, CartName, ShopName

The event to be triggered is passed in as parameter as well as the Page Event Text. In this case the event is "Cart Deleted". The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event for processors that takes an argument based on parameter type UpdateCartRequest

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

RunSaveCart –

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

LockCart

Name:	LockCart
Description:	Responsibility is to set the cart in a locked state and prevent any modifications
Usage:	Called from Sitecore.

Args:	
	Request - Contains the cart to be locked
	Is set prior to calling the pipeline.
	Response - Contains the cart object after the pipeline is called.
Processors:	
	LockCart –
	Responsibility: Is to set the cart to locked state (IsLocked = true)
	Usage: Optional. The processor is generic for locking a cart.
	Ownership: The processor is provided with Connect
	Customization: The default implementation does not call the ECS. It might be relevant to overwrite or replace the implementation to call the ECS when locking.
	RunSaveCart –
	Responsibility: Is to call pipeline SaveCart which saves the specified cart.
	Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.
	Ownership: The processor is provided with Connect
	Customization: No immediate need for overwriting the default functionality
	TriggerPageEventForRequestCart –
	Responsibility: Is to trigger a specified page event when deleting a cart and register the values:
	• ExternalCartId, UserId, CartName, CartStatus
	The event to be triggered is passed in as parameter as well as the Page Event Text. In this case the event is "CartLocked". The Page Event text is localized by looking up in Sitecore dictionary.
	Usage: Optional. The processor is generic for triggering a page event for processors that takes an argument based on parameter type CartRequest
	Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be differeent	

UnlockCart

Name:	UnlockCart
Description:	Responsibility is to set the cart in an unlocked state
Usage:	Called from Sitecore.
Args:	
	Request - Contains the cart to unlock Is set prior to calling the pipeline.
	Response - Contains the cart object after the pipeline is called.
Processors:	
	UnlockCart – Responsibility: Is to set the cart to not-locked state (IsLocked = false) Usage: Optional. The processor is generic for unlocking a cart. Ownership: The processor is provided with Connect Customization: The default implementation does not call the ECS. It might be relevant to overwrite or replace the implementation to call the ECS when unlocking.
	RunSaveCart – Responsibility: Is to call pipeline SaveCart which saves the specified cart. Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines. Ownership: The processor is provided with Connect Customization: No immediate need for overwriting the default functionality
	TriggerPageEventForRequestCart – Responsibility: Is to trigger a specified page event when deleting a cart and register the values: • ExternalCartId, UserId, CartName, CartStatus

The event to be triggered is passed in as parameter as well as the Page Event Text. In this case the event is "Cart Unlocked". The Page Event text is localized by looking up in Sitecore dictionary.

Usage: Optional. The processor is generic for triggering a page event for processors that takes an argument based on parameter type CartRequest

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from cart should be registered or the text formatting should be different

MergeCart

Name:	MergeCart
Description:	Responsibility is to merge a User cart with an Anonymous cart.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the User cart and the Anonymous cart.
	Response - Contains the merged User cart.
Processors:	

MergeCart –

Responsibility: is to merge the User and Anonymous carts.

Usage: Optional. The processor is generic for merging carts.

Ownership: The processor is provided with Connect

Customization: The default implementation does not call the ECS. It might be relevant to overwrite or replace the implementation if custom merging rules are required.

RunSaveCart –

Responsibility: Is to call pipeline SaveCart which saves the specified cart.

Usage: Optional, but should always be called when changes have been made to the cart, which is the case here. The processor is generic for calling pipeline SaveCart and can be used in other pipelines.

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

RunDeleteCart -

Responsibility: Is to call pipeline DeleteCart which deletes the specified cart, which in this case is the cart specified in parameter Anonymous Cart

Usage: Optional, depending on whether the result of the merge is to remove one of the carts. Otherwise DeleteCart can be call explicitly

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality

2.2 Pricing

2.2.1   The Pricing Domain Model
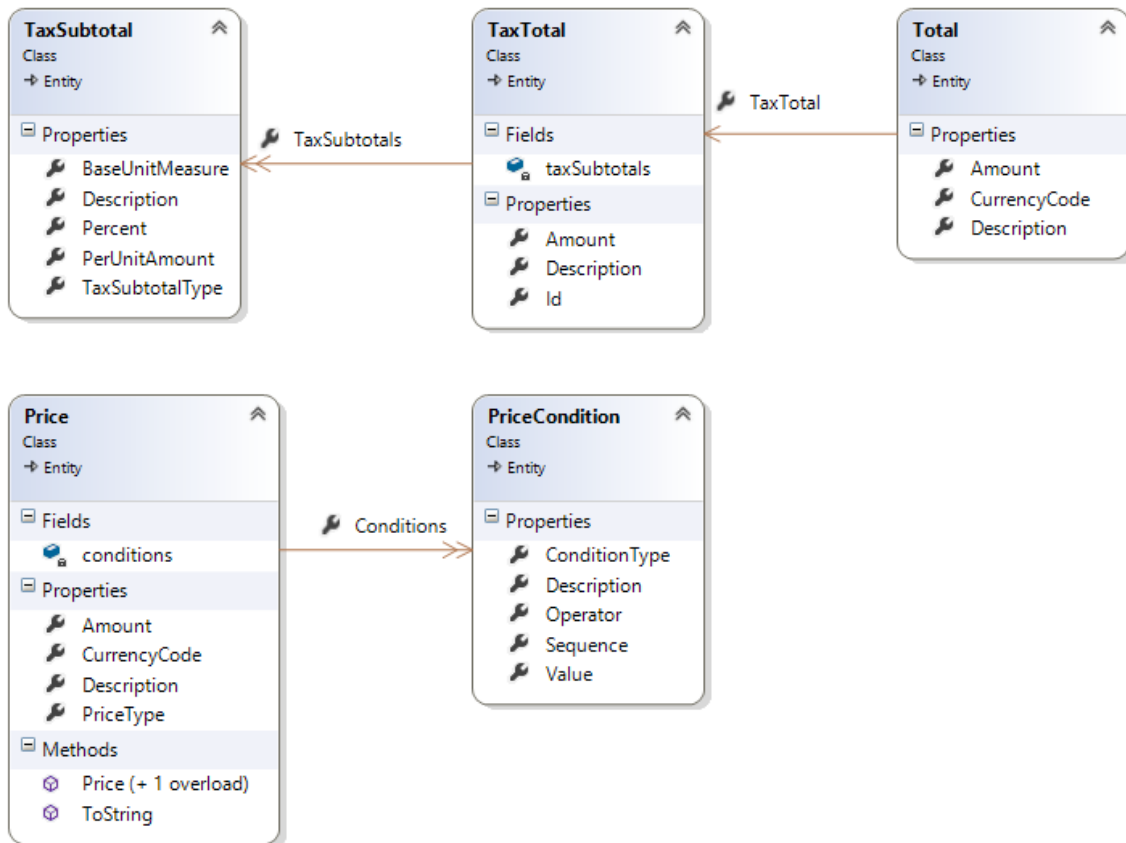
Note

The information passed in the service layer from Connect framework must be enough to meet the requirements of the external commerce system in order to do its business logic. The Connect framework doesn't perform business logic and therefore the information passed back from the external commerce system is for informational purposes.

Class: Price

Price represents the amount that a product costs. The price is used to determine what a customer has to pay for a single product, but the total cost is represented by the Total object, which takes additional information into consideration like tax, shipping etc.

Price is returned by the Pricing Service Provider as a part of the output from the GetProductPrices method. A product may have multiple prices and multiple prices might be returned from a single call. Hence a collection of Price objects is the output from the Pricing Service Provider when a single product is priced.

Name	Type	Description
PriceType	String	Examples are "List Price"(mandatory) and "Customer Price" (mandatory). Customer price means the price that the customer will have to pay taking all parameters into account.

| | | Other custom PriceTypes might be "sale price" and "break price".

There might be several prices for a single product with a given PriceType (e.g. break price), which is where the conditions are used to distinguish when the price is applicable. |
Description	String	Arbitrary text description for the price
Amount	Float	The price amount
CurrencyCode	String	Currency in which the price amount is given
Conditions	List<PriceCondition>	Used for break pricing and campaigns, where a specific price is only good when certain conditions are met (the customer has bought at least 5 products or the date is in the year 2013).

Class: Price Condition

Price Condition represents a condition that must be met in order for a price to apply. This interface must be extended for each commerce system with the actual ConditionTypes, operators and possible return values depends.

Note

The condition information is used for presentation and triggering engagement and not for calculation, so the value can be an arbitrary string.

Name	Type	Description
ConditionType	String	Examples are "quantity", "date", and "total"
Description	String	
Operator	String	Examples are "greater than", "equal to" and "between"
Sequence	Int	Represents the order in which the condition is evaluated.

For example, one condition may only apply to "shop A" and another condition may only apply to more than 5 items. If the customer is in "shop A" and has more than 5 items, which pricing should be used? This value determines this. Conditions apply in ascending order |
| Value | String | **Break prices**
Break prices depends on the quantity which is why the ConditionType will be Quantity.
In case of ConditionType Quantity the value will typically either be:
- a single integer with operator "greater than '>'"
- a range like "5-10" with operator "between"

Campaign prices
In case of ConditionType Date the value will typically either be: |

		• a single date with operator "greater than '>'" or "less than '<'" • a date range like "A - B" with operator "between"

Class: Total

Total represents the total price a customer will have to pay for a product, cart-line or entire cart at a specific point in time including charges, discounts, coupon codes, tax and shipping etc.

Total is the output from the Pricing Service Provider when a collection of products is priced as a unit (aka bundling). It provides a total price for the entire collection.

Name	Type	Description
Description	String	Arbitrary text
Amount	Float	Representing the total amount
CurrencyCode	String	A code referring to the currency
TaxTotal	TaxTotal	A reference to TaxTotal describing the tax and how it is combined from tax sub-totals

Class: TaxTotal

TaxTotal represents the tax that applies to something with a Total. Any object with a Total also has a TaxTotal.

Name	Type	Description
Id	String	This value is only available if an external system is used to perform the tax lookup and the external system provides an id (for audit purposes, for example).
Description	String	Arbitrary text
Amount	Float	Representing the total tax amount. Currency is assumed to be the same as for the Total
TaxSubtotals	List<TaxSubtotal>	A list of entries that affect the total tax

Class: TaxSubtotal

TaxSubtotal represents a specific tax that applies to an object, and the amount of the tax. This level of granularity is required in some countries.

Name	Type	Description
TaxSubtotalType	String	Examples are "CA state tax", "NYC city tax", "special levy 003a"
Description	String	
Percent	Float	Percentage per unit, zero if fixed value is used
PerUnitAmount	Float	Fixed value per unit, zero if percentage is used
BaseUnitMeasure	Float	The number of items in a unit for which the fixed value (PerUnitAmount) applies to. Not applicable if percentage is used.

2.2.2 Pricing Service Methods

Service providers are wrapper objects designed to make it easier to interact with Connect pipelines. The providers implement no logic other than calling Connect pipelines. All of the business logic is implemented in the pipeline processors.

The Pricing Service Provider contains the following methods for interacting with pricing data.

GetProductPrices

Description:	**Gets the prices for a specific product.**
Usage:	Called when Sitecore needs the prices for a specific product.
Signature:	`GetProductPricesResult GetProductPrices(GetProductPricesRequest request)`
Parameters:	
`request.ProductId –`	Required
`Request.UserId –`	Prices typically vary depending on the actual user
`Request.CurrencyCode –`	Required
`Request.Location –`	Prices often depends on the location. Location can be a city or a state.
`Request.Quantity –`	If not specified, quantity is assumed to be 1.
`Request.DateTime –`	Needed when campaigns promote products at discount prices within a certain period of time
`Request.ShopName –`	Multi-shop support
`Request.PriceTypeIds –`	List of the types of prices to retrieve. If not specified, only the base/list price is returned. Examples include list, break, and sale prices. The actual PriceTypeIds depends on the specific Connect provider implementation
Returns:	
`result.Prices –`	A collection of price objects
`result.ExternalSystem Messages –`	Collection of messages from the external system. This is how error conditions can be reported.
Exceptions:	
	No exceptions are thrown by this method.

Usage Example:

```
var pricingServiceProvider = new PricingServiceProvider();

// Create a GetProductPricesRequest object, specify the product's ID and do not
// specify any price types. Default price type is ListPrice
var request = new GetProductPricesRequest("Audi A8L");

// Call the service provider and receive the result.
var result = pricingServiceProvider.GetProductPrices(request);

// Result prices contains the list price only.
var price = result.Prices.First().Value.Amount;

// You can use the GetProductPrices to get the prices of a specific type.
// The following sample shows an example of retrieving a price of type Customer
// opposed to the default List price type:

// Create a GetProductPricesRequest object, specify the product's ID and price type
// 'Customer'.
```

```
request = new GetProductPricesRequest("Audi A8L", "Customer");

// Call service provider and receive the result.
var result2 = pricingServiceProvider.GetProductPrices(request);

// Result prices contains the Customer price only.
var price2 = result.Prices.First().Value.Amount;
```

GetProductBulkPrices

Description:	Gets the bulk prices for a specific product.
Usage:	Called when Sitecore needs the break prices for a specific product.
Signature:	GetProductBreakPricesResult GetProductBreakPrices(GetProductBreakPricesRequest request)
Parameters:	
request.ProductId –	Required
Request.UserId –	Prices typically vary depending on the actual user
Request.CurrencyCode –	Required
Request.Location –	Prices often depends on the location. Location can be a city or a state.
Request.Quantity –	If not specified, quantity is assumed to be 1.
Request.DateTime –	Needed when campaigns promote products at discount prices within a certain period of time
Request.ShopName –	Multi-shop support
Returns:	
result.Prices –	A collection of price objects
result.ExternalSystem Messages –	Collection of messages from the external system. This is how error conditions can be reported.
Exceptions:	
	No exceptions are thrown by this method.

Usage Example:

```
var pricingServiceProvider = new PricingServiceProvider();

// Create a GetProductPricesRequest object, specify the product's ID and price type
// 'Customer'. The price type argument is optional and defaults to List.
var request = new GetProductBulkPricesRequest(
    new List<string>()
    {
        "Audi A8L",
        "Renault Grand Scenic",
        "Skoda Octavia RS"
    },
    "Customer");

// Call service provider and receive the result.
var result = pricingServiceProvider.GetProductBulkPrices(request);

// Result contains a dictionary of <key, value> pairs, where the key is the
// product ID and the value represent the corresponding Price.
var price = result.Prices["Audi A8L"].Amount;
```

GetCartTotal

Description:	**Gets the price for a specific cart.**	
Usage:	Called when Sitecore needs the price for a specific cart.	
Signature:	`GetCartPriceResult GetCartPrice(GetCartPriceRequest request)`	
Parameters:		
	`request.Cart –`	Required
	`request.UserId –`	Prices typically vary depending on the actual user
	`Request.CurrencyCode –`	Required
	`Request.Location –`	Prices often depends on the location. Location can be a city or a state
	`Request.ShopName –`	Multi-shop support
	`Request.DateTime –`	Needed when campaigns promote products at discount prices within a certain period of time
Returns:		
	`result.Cart –`	An instance of a Total
	`result.ExternalSystem Messages –`	Collection of messages from the external system. This is how error conditions can be reported.
Exceptions:		
		No exceptions are thrown by this method.

Usage Example:

```
var cartServiceProvider = new CartServiceProvider();
var pricingServiceProvider = new PricingServiceProvider();

// Create LoadCart request.
var cartRequest = new CreateOrResumeCartRequest("MyShop", "MyCart");

// Call CreateOrResumeCart and get the cart
var cart = cartServiceProvider.CreateOrResumeCart(cartRequest).Cart

// Create a GetCartTotalRequest object, specify the Cart and shop name
var request = new GetCartTotalRequest {Cart = cart, ShopName = "MyShop"};

// Call service provider and receive the result.
var result = pricingServiceProvider.GetCartTotal(request);

// Result contains the updated cart augmented with Total, TaxTotal,
// and TaxSubTotal instances
var cartTotal = result.Cart.Total.Amount;
```

2.2.3   Pricing Pipelines

GetProductPrices

Name:	GetProductPrices
Description:	Gets the price object that matches the specified criteria. This pipeline is responsible for reading pricing data from a commerce system. This pipeline requests product pricing information from the commerce system and then converts the output into the proper Connect format.
Usage:	Called by the Pricing Service Provider.
Args Parameters:	
Request –	Includes the search criteria. Is set by the Pricing Service Provider prior to calling the pipeline.
Response –	Includes the price collection object. Is read by the Pricing Service Provider after the pipeline is called.
Processors:	
GetProductPrices –	Retrieves the prices specified by request.ProductTypeIds.

GetProductBreakPrices

Name:	GetProductBreakPrices
Description:	Gets the break price objects with corresponding conditions that matches the specified criteria. This pipeline is responsible for reading break pricing data from a commerce system. This pipeline requests product pricing information from the commerce system and then converts the output into the proper Connect format.
Usage:	Called by the Pricing Service Provider.
Args Parameters:	
Request –	Includes the search criteria. Is set by the Pricing Service Provider prior to calling the pipeline.
Response –	Includes the price collection object. Is read by the Pricing Service Provider after the pipeline is called.
Processors:	
EvaluatePriceConditions –	In a case where multiple prices exist for the product, determine which price applies. With break prices several prices needs to be returned and conditions created. It's the responsibility of this processor to build the conditions with the associated prices. For more info, see definition of condition

GetCartTotals

Name:	GetCartTotals
Description:	Gets the totals object that matches the specified criteria. This pipeline is responsible for reading pricing data from a commerce system. This pipeline converts the contents of a Connect cart into a format the commerce system can understand, requests the commerce system calculate the totals, and then converts the output into the proper Connect format.
Usage:	Called by the Pricing Service Provider.
Args Parameters:	
`Request –`	Includes the search criteria. Is set by the Pricing Service Provider prior to calling the pipeline.
`Response –`	Includes the totals for the cart. Is read by the Pricing Service Provider after the pipeline is called.
Processors:	
`ApplyCartAdjustments –`	Adjustments represent charges or discounts that needs to be resolved and applied. For example, discount codes/promotions, special charges for products
`GetTaxesForCart –`	Taxes might be calculated by a separate service
`GetShippingChargesFor Cart –`	Shipping might be calculated by a separate service
`GetPricesForCart –`	Does the final calculations based on content of cart

2.3 Order

2.3.1 The Order Domain Model
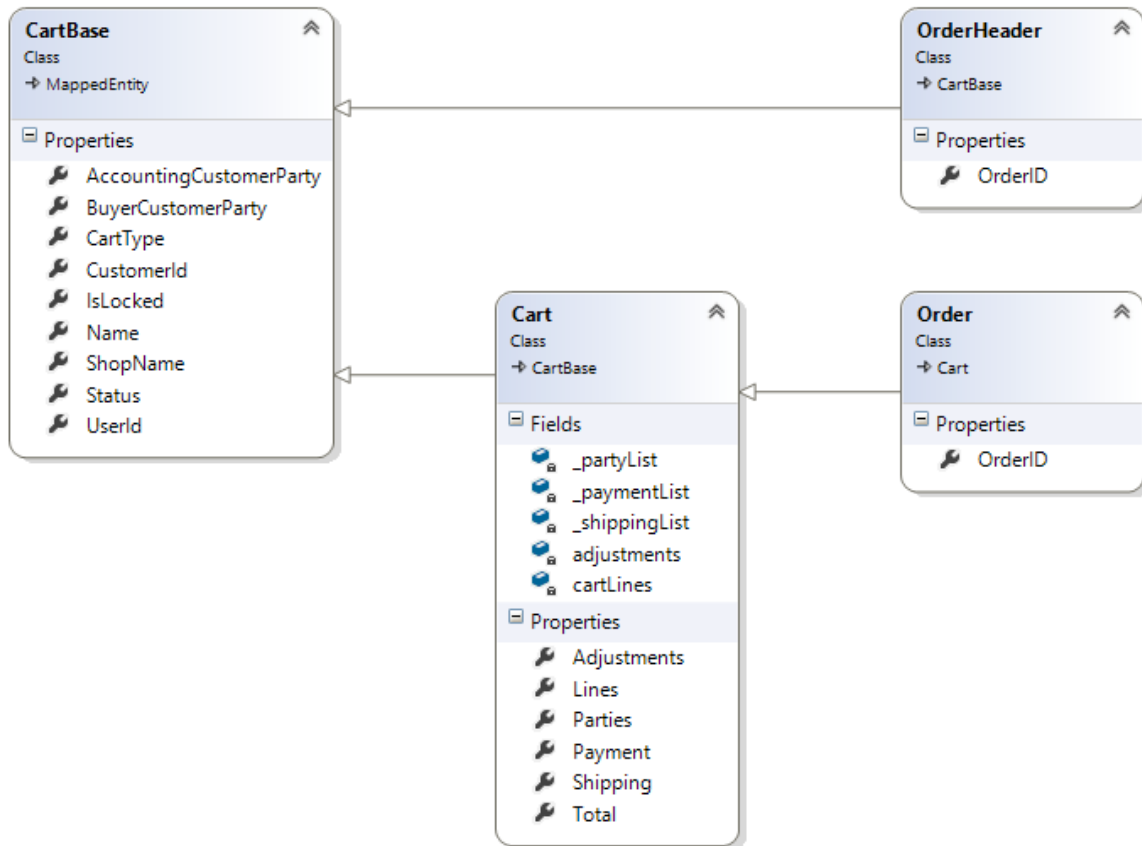
Class: Order

The Order class is responsible for representing an order.

Name	Type	Description
OrderId	String	Unique identifier for the order in the commerce system. This can be used to get a reference to the order using the commerce system's native API. Will typically be empty until an order has been created in external system

Class: OrderHeader

The OrderHeader class is responsible for representing an order header.

Name	Type	Description

OrderId	String	Unique identifier for the order in the commerce system. This can be used to get a reference to the order using the commerce system's native API.
		Will typically be empty until an order has been created in external system

2.3.2 Order Service Methods

Service providers are wrapper objects designed to make it easier to interact with Connect pipelines. The providers implement no logic other than calling Connect pipelines. All of the business logic is implemented in the pipeline processors.

The Order Service Provider contains the following methods for interacting with order data.

SubmitVisitorOrder

Name:	**SubmitVisitorOrder**
Description:	Creates an order based on the shopping cart. Calls the pipeline "SubmitVisitorOrder"
Usage:	Called from Sitecore when visitor is submitting the shopping cart to create an order.
Signature:	`SubmitVisitorOrderResult` `SubmitVisitorOrder(SubmitVisitorOrderRequest request)`
Input:	
	Cart – Cart. An instance of the shopping cart
Output:	
	Order – Cart. *In case of success, an order is returned and because order is inherited from Cart that will work fine but it needs to be cast as an order* *In case of failure, an instance of the cart object is returned, potentially modified and augmented with more data and error comments*
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var cartService = new CartServiceProvider();

// get the cart
var cart = cartService.CreateOrResumeCart(new CreateOrResumeCartRequest("MyShop",
"Me")).Cart;

// add parties, payment and shipping info
cart.Parties = new List<Party>
{
    new Party() { ExternalId = "1", PartyId = "{F73904C0-2A45-4A2F-A99B-
F934ABDCFC99}", FirstName = "Joe", LastName = "Smith", Address1 = "123 Street", City =
"Ottawa", State = "Ontario", Country = "Canada" },
    new Party() { ExternalId = "2", PartyId = "{294B7DD1-7397-4322-996C-
E87E592EF621}", FirstName = "Jane", LastName = "Smith", Address1 = "234 Street", City =
"Toronto", State = "Ontario", Country = "Canada" }
};

cart.BuyerCustomerParty = new CartParty() { ExternalId = "1", PartyID = "{F73904C0-
2A45-4A2F-A99B-F934ABDCFC99}" };
cart.AccountingCustomerParty = new CartParty() { ExternalId = "2", PartyID =
"{294B7DD1-7397-4322-996C-E87E592EF621}" };

cart.Payment = new List<PaymentInfo>
{
    new PaymentInfo() { ExternalId = "1" },
    new PaymentInfo() { ExternalId = "2" },
};
```

```
cart.Shipping = new List<ShippingInfo>
{
    new ShippingInfo() { ExternalId = "1" },
    new ShippingInfo() { ExternalId = "2" },
};

cartService.SaveCart(new SaveCartRequest(cart));

var orderService = new OrderServiceProvider();

var request = new SubmitVisitorOrderRequest(cart);
var result = orderService.SubmitVisitorOrder(request);

var order = result.Order;
var orderId = order.OrderID;
```

GetVisitorOrder

Name:	GetVisitorOrder
Description:	Get the order by Id placed by the visitor. Calls the pipeline " GetVisitorOrder"
Usage:	Called from Sitecore when order details are needed for specific order.
Signature:	GetVisitorOrderResult GetVisitorOrder (GetVisitorOrderRequest request)
Input:	
	ShopName – The name of the shop
	OrderId – The ID of the order
	CustomerID – Mandatory The ID of the customer / visitor / contact If the customer ID is not provided there is a potential security risk, that any visitor can access orders from all customers
Output:	
	Order – An instance of the order object is returned. The order object is created by the external commerce system.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var orderService = new OrderServiceProvider();

// need a valid order id for the first param
var visitorOrderRequest = new GetVisitorOrderRequest("Order_7777", "Me", "MyShop");

var result = orderService.GetVisitorOrder(visitorOrderRequest);
```

GetVisitorOrders

Name:	GetVisitorOrders
Description:	Get the order summary data of orders placed by the given visitor. Calls the pipeline " GetVisitorOrders"
Usage:	Called from Sitecore when order history is needed for visitor.
Signature:	GetVisitorOrdersResult GetVisitorOrders (GetVisitorOrdersRequest request)
Input:	
	ShopName - The name of the shop
	CustomerId – Id of the buyer customer party
Output:	
	IReadOnlyCollection<OrderBase> – An instance of the order object is returned. The order object is created by the external commerce system.

	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var orderService = new OrderServiceProvider();

var visitorOrdersRequest = new GetVisitorOrdersRequest("Me", "MyShop");

var result = orderService.GetVisitorOrders(visitorOrdersRequest);
```

VisitorCancelOrder

Purpose is for a visitor to cancel an existing order if the option is present on the web shop and if business logic does not prevent it. For example, order has already been fulfilled and/or shipped.

Typically triggered when showing order details to the customer launched from the order history view and the customer chooses to cancel the order

Name:	VisitorCancelOrder
Description:	Is used to cancel an order placed by the visitor. The decision on whether the order is cancelled or not lies in business logic in the external commerce system. Typically an order cannot be cancelled once its shop owner has started fulfilling/processing it. If the order cannot be cancelled, it must be reflected in the returned SystemMessages
Usage:	Called from Sitecore
Signature:	VisitorCancelOrderResult VisitorCancelOrder(VisitorCancelOrderRequest request)
Input:	
	ShopName – Mandatory The name of the shop
	OrderId – Mandatory The ID of the order
	CustomerID – Mandatory The ID of the customer / visitor If the customer ID is not provided there is a potential security risk, that any visitor can access orders from all customers
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var orderService = new OrderServiceProvider();

// need a valid order id for the first param
var visitorCancelOrder = new VisitorCancelOrderRequest("Order 7777", "Me", "MyShop");

var result = orderService.VisitorCancelOrder(visitorCancelOrder);
```

2.3.3   Order Pipelines

SubmitVisitorOrder

Name:	VisitorSubmitOrder
Description:	This pipeline is responsible for creating an order. The orders are managed by the commerce system.
Usage:	Called from Sitecore.
Args:	

	Request - Contains cart with the Shop name, cart, customer ID and customer party IDs for buyer (shipping) and accounting (Invoice).
	Response - Contains the order object.
Processors:	**CreateOrder –** Creates an order in the external commerce system based on the given parameters *Note: If an error occurs during processing of the cart, the Success property of the SubmitVisitorOrderResult is set to false*
	TriggerOrderGoal– the goal "Visitor Order Created" is triggered with values ShopName, Customer ID, Order Id and total order amount. The engagement value must be set to the amount of the order total!! *Note: If the Success property of the SubmitVisitorOrderResult is false no goal is triggered*
	AddOrderToEAPlan– Adds visitor to EA plan. For example, "New Order Placed", which sends the order confirmation and follows-up on purchase, customer satisfaction and new offers *Note: If the Success property of the SubmitVisitorOrderResult is false no goal is triggered*

GetVisitorOrders

Name:	**GetVisitorOrders**
Description:	Gets a list of orders for the specified customer
Usage:	Called from Sitecore.
Args:	
	Request – ShopName and CustomerID
	Response – A list of **O**rderBase objects
Processors:	
	GetVisitorOrdersFromECS– Responsibility: Get the list of orders for the specified customer from the ECS. It must be possible to have the ECS and Sitecore installed in different locations, so it must be possible to access remotely. Usage: The processor is mandatory Ownership: The processor is provided with the ECS connector integrating with Connect Customization: The processor must always have an implementation that works with the ECS
	TriggerPageEvent – Responsibility: Trigger the Connect specific page event Visitor Viewed Order History along with information about the ShopName and Customer ID Usage: Mandatory. Ownership: The processor is provided with Connect Customization: Not needed, but can be overwritten if other values from the order should be registered with the page event

GetVisitorOrder

Name:	GetVisitorOrder
Description:	Gets the order by Id placed by the visitor. Executed from method "GetVisitorOrder"
Usage:	Called from Sitecore.
Args:	
	Request – ShopName and Order ID
	Response – An instance of an order
Processors:	
	GetVisitorOrdersFromECS– Responsibility: Get the order details for the specified order ID from the ECS. Usage: The processor is mandatory Ownership: The processor is provided with the ECS connector integrating with Connect Customization: The processor must always have an implementation that works with the ECS
	TriggerPageEvent – Responsibility: Trigger the Connect specific page event Visitor Viewed Order Details along with information about the ShopName, Order ID and total order amount Usage: Mandatory. Ownership: The processor is provided with Connect Customization: Overwrite the processor if other values from the order should be registered

VisitorCancelOrder

Name:	VisitorCancelOrder
Description:	Called when a visitor order is being cancelled
Usage:	Called from Sitecore.
Args:	
	Request – OrderId, CustomerId and ShopName
	Response – an instance of the order
Processors:	
	VisitorCancelOrderFromECS–

Responsibility: Get the order details for the specified order ID from the ECS.

Usage: The processor is mandatory

Ownership: The processor is provided with the ECS connector integrating with Connect

Customization: The processor must always have an implementation that works with the ECS

TriggerPageEvent –

Responsibility: Trigger the Connect specific page event Visitor Cancelled Order Details along with information about the ShopName, Order ID and total order amount

Usage: Mandatory.

Ownership: The processor is provided with Connect

Customization: Overwrite the processor if other values from the order should be registered

2.4 Inventory

2.4.1 The Inventory Domain Model

Note: The domain model consists of classes that make up the contracts with the external system. The contracts are defined as classes instead of interfaces to allow the model to be easily extended later if needed. This follow the best practice guidelines defined in the book Framework Design Guidelines.

Default implementation of the contracts are delivered as part of Connect. If an actual Connect provider with an external commerce system contains more functionality that provided by default, the implementation can be replaced. All instantiation of actual classes will be handled through dependency injection.

Class: StockInformation

StockInformation is used as a strongly typed composite return value for service method GetStockInformation.

Name	Type	Description
Product	InventoryProduct	Identifier for the product or product variant in the commerce system
Status	StockStatus	Default possible values are: In-Stock, Out-Of-Stock, Pre-Orderable, Back-Orderable
Count	Double	In case of products being bundled in quantities there might be fractional numbers
AvailabilityDate	DateTime	In Case the product is out-of-stock or pre-orderable, an availability date can be present

Class: OrderableInformation

OrderableInformation is used as a strongly typed composite return value for service methods GetPreOrdableInformation and GetBackOrderableInformation.

Name	Type	Description
Product	InventoryProduct	Identifier for the product or product variant in the commerce system
Status	StockStatus	Default possible values are: In-Stock, Out-Of-Stock, Pre-Orderable, Back-Orderable
InStockDate	Datetime	An ETA date for when the product is back in stock
ShippingDate	DateTime	An ETA date for when the product is shippable
CartQuantityLimit	Double	A limit for the visitor to add to his or her cart
OrderableStartDate	DateTime	A date and time for when the first orders can be placed for the given product
OrderableEndDate	DateTime	A date and time for when the last orders can be placed for the given product
RemainingQuantity	Double	In case of a pre-orderable product then there might be a remaining quantity to be placed as orders

Class: IndexStockInformation

IndexStockInformation is used as a strongly typed composite value used in pipeline StockStatusForIndexing when indexing products and including basic stock information.

The entity inherits from the base entity StockLocations. In the table, the inherited properties are marked in Italics

Name	Type	Description

Product	InventoryProduct	Identifier for the product or product variant in the commerce system
InStockLocations	*List<string>*	*A list of locations where the product is in stock*
OutOfStockLocations	*List<string>*	*A list of locations where the product is out of stock*
OrderableLocations	*List<string>*	*A list of locations where the product can be ordered from*
PreOrderable	Boolean	Indicates if the product is pre-orderable or not

Class: StockInformationUpdate

StockInformationUpdate is used as a strongly typed composite return value from method GetBackInStockInformation to indicate the product and the locations where it will be back in stock optionally along with availability date and count.

Name	Type	Description
Product ID	String	Id of the product
StockInformationUpdateLocation	List< StockInformationUpdateLocation >	A list of locations where the product will become available along with the count and availability date as optional values

Class: StockInformationUpdateLocation

StockInformationUpdateLocation is used as a strongly typed value nested only into StockInformationUpdate returned from method GetBackInStockInformation to indicate the locations where the product will be back in stock optionally along with availability date and count.

Name	Type	Description
Location	String	Name of the location
AvailabilityDate	DateTime?	An optional date and time indicating when the product will be in stock. It can be used in comparison with the optional interest date that the visitor provided. If the interest date is before the availability date, then the notification signup becomes irrelevant and it can be acted upon
Count	Double?	An optional count of products that are in stock at the location at the specified date and time

Class: StockDetailsLevel

It is anticipated that there is a performance related difference between obtaining a simple stock status and getting the actual stock count. In order to allow for flexibility it is possible to specify the level of details that are requested.

Class StockDetailsLevel is used as a strongly typed request parameter for service method GetStockInformation to indicate the level of stock details that is requested. Using a strongly typed parameter will ease the use of the API for solution developers. The following example illustrates the use of the class as an enum-like parameter.

```
        StockInformation stockInformation = GetStockInformation(
                new StockInformationRequest {   shopName = "MyShop";
                                                products = new list<string> { "Aw123x" };
                                                detailsLevel = StockDetailsLevel.Status
                                            }).Result;
```

The following table contains the list of default StockDetailsLevel options. Below is an example of how the list of options can be extended.

Name	Type	Description
Status	public const int Status = 1	Indicates that the minimum information is to be returned, which is stock status
StatusAndAvailability	public const int StatusAndAvailability = 2	Indicates that the status and availability date information is to be returned. Availability date is relevant in case status is equal to
Count	public const int Count = 3	
All	public const int All = 4	

The class is introduced as an extensible enum. In order to extend and customize the StockDetailsLevel options:

```
        public class MyECSStockDetailsLevel : StockDetailsLevel
        {
                public const int MyCustomDetailLevel = 4;
                public MyECSStockDetailsLevel (int value)   : base(value)
                {    }
        }
```

Class: StockStatus

Class StockStatus is used as a strongly typed value to indicate stock status. Using a strongly typed value will ease the use of the API for solution developers. The following example illustrates the use of the class as an enum-like parameter.

```
        StockInformation stockInformation = GetStockInformation(
                new StockInformationRequest {   shopName = "MyShop";
                                                products = new list<string> { "Aw123x" };
                                                detailsLevel = StockDetailsLevel.Status
                                            }).Result.StockInformation[0];

        If (stockInformation.Status == StockStatus.PreOrderable)
        {
         // Do work
        }
```

The following table contains the list of default StockStatus options. Below is an example of how the list of options can be extended.

Name	Type	Description
InStock	public const int InStock = 1	Indicates that the requested product is in stock
OutOfStock	public const int OutOfStock = 2	Indicates that the requested product is out of stock
PreOrderable	public const int PreOrderable = 3	Indicates that the requested product is not in stock yet, but is pre-orderable
BackOrderable	public const int BackOrderable = 4	Indicates that the requested product is out of stock, but is back-orderable

The class is introduced as an extensible enum. In order to extend and customize the StockDetailsLevel options:

```
public class MyECSStockStatus : StockStatus
{
    public const int MyCustomStatus = 4;
    public MyECSStockStatus (int value)   : base(value)
    {   }
}
```

Class: InventoryProduct

Class InventoryProduct is used as a strongly typed value to identify a product. Using a strongly typed value will ease the use of the API for solution developers.

Name	Type	Description
ProductId	string	Unique identifier for the product or product variant in the commerce system

Class: StockLocations

StockLocations is used as a strongly typed composite value used in pipeline GetProductStockLocations when retrieving locations for a particular product.

StockLocations is also used as the base entity.

Name	Type	Description
InStockLocations	List<string>	A list of locations where the product is InStock
OutOfStockLocations	List<string>	A list of locations where the product is out of stock
OrderableLocations	List<string>	A list of locations where the product can be ordered from

2.4.2   Inventory Service Methods

Service providers are wrapper objects designed to make it easier to interact with Connect pipelines. The providers implement no logic other than calling Connect pipelines. All of the business logic is implemented in the pipeline processors.

The Inventory Service Provider contains the following methods for interacting with inventory data.

GetStockInformation

Name:	GetStockInformation
Description:	Retrieves different levels of stock information from the ECS Calls the pipeline "GetStockInformation".
Usage:	Called from Sitecore when stock information is needed for a list of specified products
Signature:	GetStockInformationResult GetStockInformation(GetStockInformationRequest request)
Input:	

	ShopName – string. Mandatory The name of the shop
	Products – list<InventoryProduct>. Mandatory A list of InventoryProduct. Whether it is products or product variants is up to the implementation with the ECS
	DetailsLevel – StockDetailsLevel. Mandatory Class StockDetailsLevel is used as a strongly typed request parameter for service method GetStockInformation to indicate the level of stock details that is requested
	Location – string. Optional The specific warehouse or central storage where the stock information is Default is Central storage
	Customer ID – string. Optional The ID of the customer in case the stock information is dependent on the actual customer
Output:	
	List<StockInformation> StockInformation – A list of StockInformation objects
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();

var request = new GetStockInformationRequest("shopname", new List<InventoryProduct>
    {
        new InventoryProduct
        {
            ProductId = "product_1"
        },
        new InventoryProduct
        {
            ProductId = "product_2"
        },
        new InventoryProduct
        { ProductId = "product 3" }
    },
    StockDetailsLevel.StatusAndAvailability);

var result = inventoryService.GetStockInformation(request);
```

GetPreOrderableInformation

Name:	GetPreOrderableInformation
Description:	Retrieves the pre-orderable information
Usage:	Called from Sitecore
Signature:	Result [Name](Request request)
Input:	
	ShopName – String. Mandatory The name of the shop
	Products - List< InventoryProduct >. Mandatory The list of InventoryProduct
	Visitor ID – string. Optional **The ID of the visitor / contact or customer**
Output:	
	OrderableInformation – List<OrderableInformation> A list of strongly typed objects each with the information for a specific product
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();
```

```
var request = new GetPreOrderableInformationRequest("shopname",
    new List<InventoryProduct>
    {
        new InventoryProduct
        {
            ProductId = "product_1"
        },
        new InventoryProduct
        {
            ProductId = "product_2"
        },
        new InventoryProduct
        { ProductId = "product 3" }
    });

var result = inventoryService.GetPreOrderableInformation(request);
```

GetBackOrderableInformation

Name:	**[Name]**
Description:	Gets the back-orderable information
Usage:	Called from Sitecore
Signature:	`Result [Name](Request request)`
Input:	
	ShopName – String. Mandatory The name of the shop
	Products - List< InventoryProduct **>. Mandatory** The list of InventoryProduct
	Visitor ID – string. Optional **The ID of the visitor / contact or customer**
Output:	
	OrderableInformation – List<OrderableInformation> A list of strongly typed objects each with the information for a specific product
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();

var request = new GetBackOrderableInformationRequest("shopname",
    new List<InventoryProduct>
    {
        new InventoryProduct
        {
            ProductId = "product_1"
        },
        new InventoryProduct
        {
            ProductId = "product 2"
        },
        new InventoryProduct
        { ProductId = "product 3" }
    });

var result = inventoryService.GetBackOrderableInformation(request);
```

VisitedProductStockStatus

Name:	**VisitedProductStockStatus**
Description:	Should be called in the event of the customer pays a visit to a product details page which shows
Usage:	Called from Sitecore

Signature:	VisitedProductStockStatusResult VisitedProductStockStatus (VisitedProductStockStatusRequest request)
Input:	
	ShopName – string. Mandatory The name of the shop
	StockInformation - StockInformation. Mandatory The stock information previously retrieved by calling GetStockInformation
	Location – string. Optional The specific warehouse or central storage where the stock information is Default is Central storage
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();

var stockInfo = new StockInformation
{
    Product = new InventoryProduct
    {
        ProductId = "product_1"
    },
    Status = StockStatus.BackOrderable
};

var request = new VisitedProductStockStatusRequest("shopname", stockInfo);

var result = inventoryService.VisitedProductStockStatus(request);
```

ProductsAreBackInStock

Name:	**ProductsAreBackInStock**
Description:	A method that is exposed so that it can be called remotely to signal when products are back in stock. Executes the corresponding pipeline ProductsAreBackInStock
Usage:	Called from ECS
Signature:	ProductsAreBackInStockResult ProductsAreBackInStock(ProductsAreBackInStockRequest request)
Input:	
	Shop Name – string. Mandatory The name of the shop for which this relates
	Products – list< InventoryProduct**>. Mandatory** A list of InventoryProduct that signals which products have updated stock information
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();

var request = new ProductsAreBackInStockRequest("shopname",
    new List<InventoryProduct>
    {
        new InventoryProduct
        {
            ProductId = "product_1"
        },
        new InventoryProduct
        {
```

```
            ProductId = "product_2"
        },
        new InventoryProduct
        { ProductId = "product 3" }
    });

    var result = inventoryService.ProductsAreBackInStock(request);
```

VisitorSignUpForStockNotification

Name:	VisitorSignUpForStockNotification
Description:	Is used to add visitor to EA plan so they can be notified when the product gets back in stock
Usage:	Called from Sitecore
Signature:	`VisitorSignUpForStockNotificationResult VisitorSignUpForStockNotification(VisitorSignUpForStockNotificationRequest request)`
Input:	
	ShopName – string. Mandatory
	Visitor ID – string. Mandatory The ID of the current visitor / contact
	Email – string. Mandatory E-mail address to send the notification to
	Product **–** InventoryProduct**. Mandatory**
	Location – string. Optional
	InterestDate – DateTime. Optional A date and time that signals the deadline for which to notify the visitor Default is 6 months from now
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var inventoryService = new InventoryServiceProvider();

    var request = new VisitorSignUpForStockNotificationRequest("shopname",
        "visitorId",
        "email",
        new InventoryProduct { ProductId = "product_1" });

    var result = inventoryService.VisitorSignUpForStockNotification(request);
```

RemoveVisitorFromStockNotification

Name:	RemoveVisitorFromStockNotification
Description:	Typically called from Sitecore, when the visitor has decided to unsubscribe from the stock notification and hence the EA plan
Usage:	Called from Sitecore
Signature:	`RemoveVisitorFromStockNotificationResult RemoveVisitorFromStockNotification(RemoveVisitorFromStockNotificationRequest request)`
Input:	
	ShopName – string. Mandatory
	Visitor ID – string. Mandatory The ID of the current visitor / contact
	Product ID – string. Mandatory
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();

var request = new RemoveVisitorFromStockNotificationRequest("shopname",
    "visitorId",
    new InventoryProduct { ProductId = "product 1" });

var result = inventoryService.RemoveVisitorFromStockNotification(request);
```

GetBackInStockInformation

Name:	**GetBackInStockInformation**
Description:	The method is used to get the updated stock information from the ECS The method is normally called because of method ProductsAreBackInStock has been called remotely and in that case the pipeline is implicitly called The difference between GetStockInformation and GetBackInStockInformation is that the first queries for stock information in a specified location and for a given customer, where the latter gets the stock information for all locations ignoring customer context
Usage:	Called from Sitecore
Signature:	`GetBackInStockInformationResult GetBackInStockInformation (GetBackInStockInformationRequest request)`
Input:	
	Shop Name – string. Mandatory The name of the shop for which this relates
	Products – list< InventoryProduct >. Optional A list of InventoryProduct to get updated stock information from. If the list is empty, for example no list is provided, it is up to the ECS to return stock information updates for the products that have been updated.
Output:	
	StockInformationUpdates – list<StockInformationUpdate>. Mandatory A list of StockInformationUpdate object that each signals the product and a list of locations for which the product is back in stock, when (availability date) and the count, where the latter two are optional values.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var inventoryService = new InventoryServiceProvider();

var request = new GetBackInStockInformationRequest("shopname")
{
    Products = new List<InventoryProduct>
    {
        new InventoryProduct
        {
            ProductId = "product_1"
        },
        new InventoryProduct
        {
            ProductId = "product_2"
        },
        new InventoryProduct
        {
            ProductId = "product 3"
        }
    }
};

var result = inventoryService.GetBackInStockInformation(request);
```

2.4.3　Inventory Pipelines

GetStockInformation

Name:	GetStockInformation
Description:	This pipeline is responsible for retrieving stock information for one or more products specified
Usage:	Called from Sitecore.
Args:	
	Request - Contains the list of InventoryProduct, details level, shop name, location and customer ID. Is set prior to calling the pipeline.
	Response - Contains the user object. Is read after the pipeline is called.
Processors:	
	GetStockInformation – Responsibility: To retrieve stock information for the list of specified product IDs Usage: Calls the ECS to get the stock information Ownership: The processor is provided by the ECS Customization: Must be created as part of the connector integrating with the ECS

StockStatusForIndexing

Developer story:

- As a developer I have a StockStatusForIndexing pipeline that runs when the crawler is indexing products and that returns the information to be indexed.

Acceptance criteria

- Check pipeline StockStatusForIndexing has no service layer method associated
- Check pipeline is separate from the pipeline associated with GetProductStockInformation
- Check the output retrieves in-stock or our-of-stock information associated with each location
- Check the output includes which location the product is sold

Notes

- User story is 405718

- As a developer I get stock status information included in the product index so when I query the index I can include criteria regarding stock status in connection with location

Acceptance criteria

- o Check that we have a InStock index field that lists all the locations where the product is in stock
- o Check that we have a OutOfStock index field that lists all the locations where the product is out of stock
- o Check that we have a Location index field that contains the locations where the product is orderable from
- o Check that we have a Pre-Orderable index field (Boolean) that indicates whether the product is pre-orderable or not

Notes

- o The index contains only stock status information per product and not per variant
- o The table below shows an example of the product index content for a T-shirt product that comes in different variants and with the In-Stock and Out-of-Stock columns

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

Name:	**StockStatusForIndexing**
Description:	Called during indexing to populate the index with stock information
Usage:	Called from Sitecore.
Args:	
	Request – A list of Product IDs is provided from the Sitecore indexing
	Response – A list of IndexStockInformation objects is returned for processing and inclusion into the index
Processors:	
	StockStatusForIndexing – Responsibility: To call the ECS and retrieve stock information used to populate the product index Usage: Called when crawling product repository and indexing the products. Ownership: Custom processor provided with the connector to the ECS Customization:

GetPreOrderableInformation

Name:	**GetPreOrderableInformation**
Description:	Contacts the ECS to get the pre-orderable information
Usage:	Called from Sitecore
Args:	
	Request – ShopName, list of InventoryProduct, Visitor ID, Location
	Response – A list of OrderableInformation
Processors:	
	GetPreOrderableInformation **–**
	Responsibility: Contacts the ECS to get the pre-orderable information
	Usage: Called from Sitecore to retrieve information to be used for rendering to the visitor as well as restricting placing orders
	Ownership: Provided with the ECS connector
	Customization: Must be customized

GetBackOrderableInformation

Name:	**GetBackOrderableInformation**
Description:	Contacts the ECS to get the back-orderable information
Usage:	Called from Sitecore
Args:	
	Request – ShopName, list of InventoryProduct, Visitor ID, Location
	Response – A list of OrderableInformation
Processors:	
	GetBackOrderableInformation **–**
	Responsibility: Contacts the ECS to get the back-orderable information
	Usage: Called from Sitecore to retrieve information to be used for rendering to the visitor as well as restricting placing orders
	Ownership: Provided with the ECS connector
	Customization: Must be customized

VisitorAppliedFacet

Name:	VisitorAppliedFacet
Description:	This pipeline is responsible for triggering page event FacetedBy
Usage:	Called from Sitecore.
Args:	
	Request - Contains the additional pageevent event information
	Response - None
Processors:	
	TriggerPageEvent – Responsibility: To trigger page event "FacetedBy" to register the facet used by the visitor Usage: Called from Sitecore and typically doesn't call the ECS at all Ownership: The processor is provided with Connect Customization: No immediate need for overwriting the default functionality, unless more information should be registered with the page event

VisitorAppliedSortOrder

Name:	VisitorAppliedSortOrder
Description:	This pipeline is responsible for triggering page event SortedBy
Usage:	Called from Sitecore.
Args:	
	Request - Contains the additional pageevent event information
	Response – None
Processors:	
	TriggerPageEvent – Responsibility: To trigger page event "SortedBy" to register the columns and directions that has been chosen by the user

Usage: Called from Sitecore and typically doesn't call the ECS at all

Ownership: The processor is provided with Connect

Customization: No immediate need for overwriting the default functionality, unless more information should be registered with the page event

ProductsAreBackInStock

Name:	**ProductsAreBackInStock**
Description:	Triggers the page event Products Back In Stock, so that the visitors which have signed up, can be notified
Usage:	Called from Sitecore.
Args:	
	Request – Shop name and a list of InventoryProduct for which the product is back in stock
	Response – None
Processors:	
	TriggerPageEvent–

Responsibility: Trigger page event Products Back In Stock along with the shop name and a list of product IDs. Firing the event will trigger the EA plan to re-evaluate the visitors and determine whether they should be notified

Usage: Called from the ECS to signal when products have come back in stock

Ownership: Provided with Connect

Customization: No immediate need |

GetBackInStockInformation

Name:	**GetBackInStockInformation**
Description:	The method and pipeline is used to get the updated stock information from the ECS

The method is normally called because of method ProductsAreBackInStock has been called remotely and in that case the pipeline is implicitly called |

	The difference between GetStockInformation and GetBackInStockInformation is that the first queries for stock information in a specified location and for a given customer, where the latter gets the stock information for all locations ignoring customer context
Usage:	Called from Sitecore.
Args:	
	Request – Shop name and optionally a list of InventoryProduct.
	Response – A list of StockInformationUpdate objects
Processors:	
	GetStockInformationUpdates– Responsibility: To retrieve a list of StockInformation Updates objects from the ECS, each describing the product and a list of locations for which the product is back in stock, when (availability date) and the count, where the latter two are optional values The parameters can optionally include a list of product IDs specifying the products for which a stock update is requested. If the list is empty then it is up to the ECS to keep track of which products that new stock updates. It's needed in case the ECS is not able to notify Connect of stock update changes. In That case Connect should be able to query for any updates in order for the EA plan to work Usage: Called from the Sitecore to get the information needed to follow-up in the EA plan Ownership: Provided with Connect Customization: No immediate need

VisitorSignUpForStockNotification

Name:	**VisitorSignUpForStockNotification**
Description:	Called from Sitecore when a visitor wants to be notified when a product gets back in stock
Usage:	Called from Sitecore.
Args:	
	Request – ShopName, Visitor ID, InventoryProduct **and interest Date**
	Response – None
Processors:	
	VisitorSignUpForStockNotification – Responsibility:

- Check visitors, who are **not** already in the EA plan Back In Stock Notification, are added

- Check Product ID and interest date are stored in the EA state

- Check visitors who **are** already in the plan stays in the same state, but has an additional product ID and interest date added to the list

Usage:

Ownership: Provided with Connect

Customization: No immediate need

TriggerPageEvent

Responsibility:

- Check that page event Back In Stock Subscription is triggered which includes the product ID, e-mail address and interest date

Usage:

Ownership: Provided with Connect

Customization: No immediate need

RemoveVisitorFromStockNotification

Name:	RemoveVisitorFromStockNotification
Description:	Typically called from Sitecore, when the visitor has decided to unsubscribe from the stock notification and hence the EA plan
	Removes the visitor from the EA plan and triggers a page event
Usage:	Executed from Sitecore, when the method with the same name is called
Args:	
	Request – ShopName, VisitorID and ProductID
	Response – None
Processors:	
	RemoveVisitorFromStockNotification **–**
	Responsibility:
	- Check visitor ID and product ID are provided

-	Check that the product ID and interest date is removed from the visitors list and saved to EA state
-	Check that if the visitors list of product IDs is empty, then the visitor is removed from the plan all together

Usage:

Ownership: Provided with Connect

Customization: No immediate need for customization

TriggerPageEvent –

Responsibility: Check that page event Back In Stock Unsubscription is triggered which includes the product ID, e-mail address and interest date.

Usage:

Ownership: Provided with Connect

Customization: No immediate need for customization

OrderedProductStockStatus

Name:	**OrderedProductStockStatus**
Description:	Pipeline is called as a part of the SubmitOrder pipeline
Usage:	Called implicitly from pipeline AddLinesToCart from the Cart service layer to trigger a page event whenever a product which is our of stock is added to the cart
Args:	
	Request – ShopName and Cart
	Response - None, except for external system messages
Processors:	
	TriggerPageEvent– Responsibility: For each order line, **t**rigger **a** page event ProductsOutOfStockOrdered along with the ShopName, Order ID, Product ID, Stock Status, if and only if, the stock status is NOT InStock Usage: Mandatory Ownership: Provided with Connect

Customization: Not needed

AddToCartStockStatus

Name:	AddToCartStockStatus
Description:	Pipeline is called as a part of the AddLinesToCart pipeline
Usage:	Called implicitly from pipeline AddLinesToCart from the Cart service layer to trigger a page event whenever a product which is our of stock is added to the cart
Args:	
	Request – ShopName, Customer ID (visitor ID), Cart ID and Cart-Lines
	Response – None, except for external system messages
Processors:	
	TriggerPageEvent– Responsibility: Trigger page event AddToCartStockStatus along with the ShopName, Cart ID, Product ID , Stock Status, Pre-orderable, In-Stock Date, Shipping Date, if and only if, the stock status is NOT InStock Usage: Mandatory Ownership: Provided with Connect Customization: Not needed

VisitedProductStockStatus

Name:	VisitedProductStockStatus
Description:	
Usage:	Called from Sitecore.
Args:	
	Request – ShopName, Location and an instance of StockInformation
	Response – None except system messages
Processors:	
	TriggerPageEvent– Responsibility: Trigger page event VisitedProductStockStatus along with ShopName and the information fund in the StockInformation entity, for example Product ID, Stock Status, AvailabilityDate and count. Note that not all the stock information fields might have values filled in.

Usage: Mandatory

Ownership: Provided with Connect

Customization: Not needed

CurrentProductID

Name:	CurrentProductID
Description:	The processor is implicitly used by rendering rule conditions that needs to know the context of which product is the currently selected product
	The pipeline is called from Connect.Context when resolving the property ProductId which returns the current product ID
Usage:	Called from Sitecore.
Args:	
	Request -
	Response – The product ID of the current product ID
Processors:	
	GetCurrentProductID–
	Responsibility:
	Establish the external ID of the currently selected product.
	Depending on the solution, there are different ways to setup the current product
	Default implementation will resolve the ID according to how the ECS is setup. That means the URL indicates the product ID of the current product when viewing a product details page. On all other pages, the value is undefined
	Usage: Used from Rendering rule conditions
	Ownership: ECS
	Customization: Must be customized in every solution

2.5 Customer

2.5.1   The Customer Domain Model
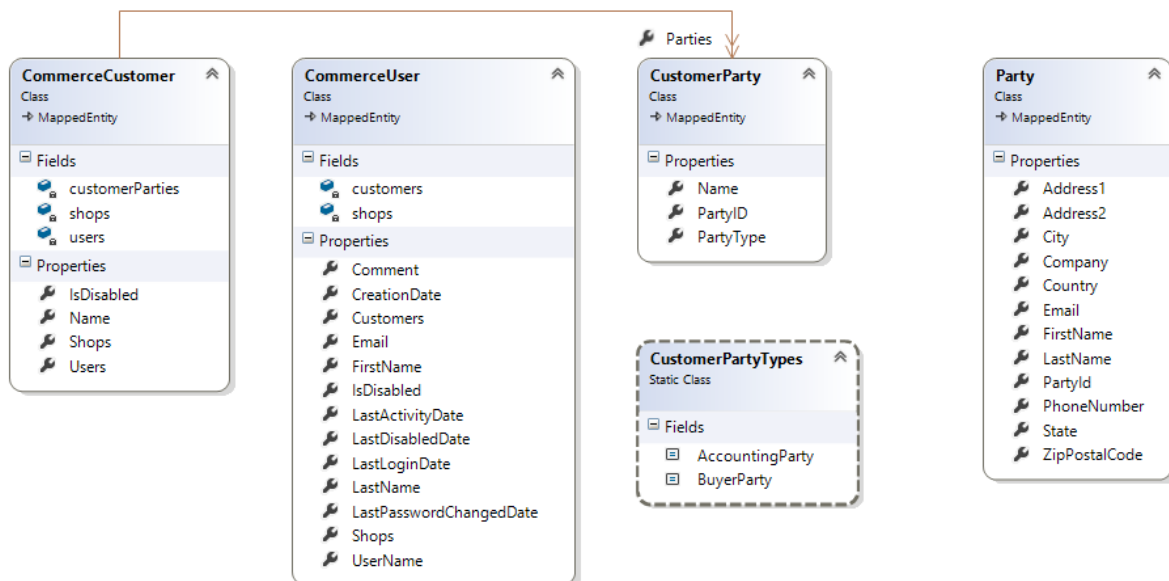
Class: CommerceUser

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

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

The following assertions relates to a user:

- A user represents a person who can log in to a website. For example, meaning that an account exists for that user in the system.
- A visitor going through the checkout without registering a user account. For example, anonymous checkout, will be created as a customer, but without a user account
- Customer and User relationship:
  - o A user can represent multiple different customers acting as an agent on the customers behalf
  - o A customer, such as a company, can have multiple users representing the company. For example, employees of different departments
  - o Based on the two previous statements it means that there is a many to many relationship between the two.
- Multiple users can manage the same shopping cart on behalf of the same customer or have individual shopping carts

Usage scenarios:

- When a new account is explicitly created on the site
- When an existing user account is used during checkout
- When the returning user authenticates him- or herself to get the customer specific product prices and discounts
- When the user and/or customer profile is edited by the user
- When the shop owner sends out welcome mail to new users
- When the shop owner wants to follow-up on new users that hasn't returned to the shop for some time (B2C)

Name	Type	Description
ExternalID	String	Unique identifier for the user in the commerce system. This can be used to get a reference to the user using the commerce system's native API. Will be empty until account has been created in external system
ID	String	Unique ID
Email	String	The users e-mail
FirstName	String	
LastName	String	
Shops	List<String>	The list of shops the user has access to.
Disabled	Boolean	Indicates if the user account is disabled or not
Comment	String	Free text comment
CreationDate	DateTime	Gets the date and time when the account was added.
LastActivityDate	DateTime	Gets or sets the date and time when the user was last authenticated or accessed the shop.
LastLoginDate	DateTime	Gets or sets the date and time when the user was last authenticated.
LastDisabledDate	DateTime	Gets the most recent date and time when the user account was disabled.
LastPasswordChangedDate	DateTime	Gets the date and time when the user's password was last updated.
Customers	List<string>	The list of customer IDs of the customers that the user is associated with

Class: CommerceCustomer

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

The CommerceCustomer entity will always be extended to include custom information particular to the external commerce system.

Definition

- A customer represents a person in a B2C shops and a company in a B2B shops.
- A customer account **cannot** be used to log in to the webshop. In order to log in, a user account is needed. A Customer account is different from a user account and the two can carry different information.
- Not all commerce systems support the concept of both a user and a customer. Example: SES only has users whereas InSite Commerce has both customers and users.
  - When the external system does not support customers, the information might be stored together in the User profile, but the Relation will typically not be available.
- In B2C scenarios a customer and a user is typically the same
- In B2B scenarios a customer typically has 1-many users associated:
  - A customer is typically the one paying the invoices for the orders placed in the system by its users
  - Users are the ones managing shopping carts on behalf of the customer and go through the checkout process, placing the order.
  - A customer can have multiple shopping carts associated and they can be managed by one or many different users

Usage scenarios:

- When a new customer account is created implicitly during anonymous checkout
- When the customer profile is edited by the shop owner
- When the customer places an order and gets an order confirmation mail
- When the shop owner sends out welcome mail to new customers in B2B solutions
- When the shop owner wants to make a campaign or promote special offers in B2B solutions
- When the shop owner wants to follow-up on new customers that hasn't placed orders in the shop for some time

Name	Type	Description
ExternalID	String	Unique identifier for the user in the commerce system. This can be used to get a reference to the user using the commerce system's native API. If ASP.NET is used, then the ExternalUserId will equal UserName Will be empty until account has been created in external system
Name	String	The name of the customer
ID	String	Unique ID
IsDisabled	Boolean	Indicates if the customer account is disabled or not
Shops	List<String>	The list of shops the customer has access to
CustomerParties	List<CustomerParty>	The list of parties consisting of contact and address information
CommerceUsers	List<string>	The list of user IDs of the users that the customer is associated with

Class: CustomerParty

CustomerParty is used to represent the type and 0-to-many relationship between the customer and a list of parties

Name	Type	Description
ExternalID	String	Unique identifier for the party
PartyId	String	ID of the party. Reserved for future use in Sitecore

Name	String	An optional string for that describes the relationship
Type	CustomerPartyTypes	Represent the type of relationship

Class: CustomerPartyTypes

Class CustomerPartyTypes is used to indicate the type of relationship between the customer and party.

The class is introduced as an extensible enum. In order to extend and customize the CustomerPartyTypes options:

```
public class MyECSCustomerPartyTypes : CustomerPartyTypes
{
      public const int MyCustomPartyType = 3;
      public MyECSCustomerPartyTypes(int value)   : base(value)
      {    }
}
```

Name	Type	Description
BuyerParty	public const int BuyerParty = 1	BuyerParty represents the buyer and are typically used as the party where products are shipped to
AccountingParty	public const int AccountingParty = 2	AccountingParty represents the buyer and are typically used as the party where the invoice is sent to

Class: Party

The CustomerParty entity represents address contact information and a list of CustomerParty objects is included in the Customer class

Note: The CustomerParty object is used in both the customer and order service layers.

Name	Type	Description
ExternalID	String	Unique identifier for the party
PartyID	String	ID of the party
FirstName	String	First name
LastName	String	Last name
Email	String	E-mail address
Company	String	Company name
Address1	String	Streetname
Address2	String	Region, District, County etc
ZipPostalCode	String	Zip/Postal code
City	String	City name
State	String	State
Country	String	Country name
PhoneNumber	String	Phone number

2.5.2   Customer Service Methods

Service providers are wrapper objects designed to make it easier to interact with Connect pipelines. The providers implement no logic other than calling pipelines. All of the business logic is implemented in the pipeline processors.

For each method there is a corresponding Request and Result object getting returned, ex. CreateUser takes a CreateUserRequest object and returns a CreateUserResult object.

The Customer Service Provider contains the following methods for interacting with customer and user data.

CreateUser

Name:	**CreateUser**
Description:	Creates a user account by which the user can re-authenticate him- or herself upon return. By default the account is disabled until it has be confirmed by visitor to be a valid request and ActivateUserAccount has been called Calls the pipeline "`CreateUser`".
Usage:	Called from Sitecore when a visitor is registering for an account. It could be during the checkout process or through plain signup.
Signature:	`CreateUserResult CreateUser(CreateUserRequest request)`
Input:	
	Username – string. Mandatory The user name for the new user.
	Email – string. Mandatory The e-mail address for the new user.
	Password – string. Mandatory The password for the new user.
	Shops– Mandatory An instance of the CommerceUser object is parsed in. Mandatory fields: Username, Email, Shops.
Output:	
	User – An instance of the user object is returned. The user object is updated by the external commerce system by supplying the External ID value.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");

var result = customerService.CreateUser(request);
```

UpdateUser

Name:	**UpdateUser**
Description:	Updates an existing user account. Calls the pipeline "UpdateUser"
Usage:	Called from Sitecore when visitor wants to update the information stored on the account.
Signature:	`UpdateUserResult UpdateUser(UpdateUserRequest request)`
Input:	
	CommerceUser – An instance of the modified CommerceUser object is passed in
Output:	
	User – An instance of the user object is returned.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();
```

```
    // create a user
    var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");

    var user = customerService.CreateUser(request).CommerceUser;
    user.FirstName = "John";

    // update the user
    var updateRequest = new UpdateUserRequest(user);
    var result = customerService.UpdateUser(updateRequest);
```

DeleteUser

Name:	DeleteUser
Description:	Deletes a user account. Calls the pipeline "DeleteUser".
Usage:	Called from Sitecore when the shop owner wants to delete an account. It's a solution business decision whether the account is actually deleted or simply disabled
Signature:	DeleteUserResult DeleteUser(DeleteUserRequest request)
Input:	
	CommerceUser – An instance of the CommerceUser object is parsed in
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var customerService = new CustomerServiceProvider();

    // create a user
    var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
    var user = customerService.CreateUser(request).CommerceUser;

    // delete the user
    var deleteRequest = new DeleteUserRequest(user);
    var result = customerService.DeleteUser(deleteRequest);
```

DisableUser

Name:	DisableUser
Description:	Disables a user account. Calls the pipeline "DisableUser"
Usage:	Called from Sitecore when the user account should be disabled
Signature:	DisableUserResult DisableUser(DisableUserRequest request)
Input:	
	CommerceUser – Mandatory An instance of the user object is parsed in
	Comment – Optional An optional string that can explain why the user account was disabled. Will be put in the Page Event as explanation
Output:	
	CommerceUser – The disabled CommerceUser entity
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var customerService = new CustomerServiceProvider();

    // create a user
    var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
    var user = customerService.CreateUser(request).CommerceUser;
```

```
// disable the user
var disableRequest = new DisableUserRequest(user);
var disableResult = customerService.DisableUser(disableRequest);

// enable the user
var enableRequest = new EnableUserRequest(user);
var enableResult = customerService.EnableUser(enableRequest);
```

EnableUser

Name:	EnableUser
Description:	Enables a user account. Calls the pipeline "EnableUser" A user account can be disabled for different reasons and triggered by shop owner, visitor or by system (EA plan). When a user account is disabled it must be possible to enable it again, which is the purpose of this method. The method seems similar to ActivateUserAccount in the way that both enable the account. The difference lies in the usage scenario and possible actions associated. EnableUser is a generic method whereas UserAccountActivation is used in a specific scenario when a new account is created and must be validated and then activated. The two trigger different page events, where the event User Account Activated triggers the EA plan User Account Registration to proceed.
Usage:	Called from Sitecore when the user account should be enabled
Signature:	`EnableUserResult EnableUser(EnableUserRequest request)`
Input:	
	CommerceUser – Mandatory An instance of the user object is parsed in
	Comment – Optional An optional string that can explain why the user account was enabled. Will be put in the Page Event as explanation
Output:	
	CommerceUser – The enabled CommerceUser entity
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create a user
var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
var user = customerService.CreateUser(request).CommerceUser;

// disable the user
var disableRequest = new DisableUserRequest(user);
var disableResult = customerService.DisableUser(disableRequest);

// enable the user
var enableRequest = new EnableUserRequest(user);
var enableResult = customerService.EnableUser(enableRequest);
```

GetUser

Name:	GetUser
Description:	Returns a single user account. Calls the pipeline "GetUser".

Usage:	Called from Sitecore when searching for one or more accounts
Signature:	`GetUserResult GetUser(GetUserRequest request)`
Input:	
	ShopName – Mandatory
	UserName – Mandatory
	The ID of the user to retrieve
Output:	
	User – A single instance of a User
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create a user
var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
var user = customerService.CreateUser(request).CommerceUser;

var getRequest = new GetUserRequest("JohnSmith");
var result = customerService.GetUser(getRequest);
```

GetUsers

Name:	**GetUsers**
Description:	Queries and returns user accounts. Calls the pipeline "Get`Users`".
	Different input parameters can be provided and they will be combined using logical *and*
Usage:	Called from Sitecore when searching for one or more accounts
Signature:	`GetUsersResult GetUsers(GetUsersRequest request)`
Input:	
	ExternalIDs – Optional.
	Can be a single or a list of IDs. When provided, it takes precedence
	SitecoreIDs – Optional
	Can be a single or a list of IDs
	UserName – optional
	Email – optional
	ExternalCustomerIDs – optional
	Can be a single or a list of IDs. Used when looking for users associated with a given customer
	Disabled - optional
	ShopName – optional
Output:	
	List<User>
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create some users
var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
var result = customerService.CreateUser(request);
request = new CreateUserRequest("JaneSmith", "passWord", "jane@jane.net", "webstore");
result = customerService.CreateUser(request);
request = new CreateUserRequest("Rob", "abcdefghij", "r@r.com", "webstore");
result = customerService.CreateUser(request);

var getRequest = new GetUsersRequest(new UserSearchCriteria { ShopName = "webstore"
});
var getResult = customerService.GetUsers(getRequest);
Sitecore.Diagnostics.Assert.AreEqual(3, getResult.CommerceUsers.Count, "should have a
count of 3");
```

```
getRequest = new GetUsersRequest(new UserSearchCriteria { UserName = "JohnSmith" });
getResult = customerService.GetUsers(getRequest);
Sitecore.Diagnostics.Assert.AreEqual(1, getResult.CommerceUsers.Count, "should have a
count of 1");
```

CreateCustomer

Name:	CreateCustomer
Description:	Creates a customer. Calls the pipeline "CreateCustomer"
Usage:	Typically called when a visitor is going through the checkout process
Signature:	CreateCustomerResult CreateCustomer(CreateCustomerRequest request)
Input:	
	CommerceCustomer – Mandatory An instance of the CommerceCustomer object is parsed in Mandatory field values are: Name and Shops
Output:	
	CommerceCustomer – An instance of the customer object is returned. The customer object is updated by the external commerce system by supplying the External ID.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
        IsDisabled = true,
        ExternalId = "Jeff",
        Users = new[] { "Jeff" }
    }));

customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        IsDisabled = false,
        ExternalId = "Jeff",
        Users = new[] { "Bob" }
    }));
```

UpdateCustomer

Name:	UpdateCustomer
Description:	Updates an existing customer account. Calls the pipeline "UpdateCustomer"
Usage:	Called from Sitecore when visitor or shop owner wants to update the information stored on the customer account.
Signature:	UpdateCustomerResult UpdateCustomer(UpdateCustomerRequest request)
Input:	
	CommerceCustomer – An instance of the modified CommerceCustomer object is parsed in
Output:	
	CommerceCustomer – An instance of the customer object is returned.
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
        IsDisabled = true,
        ExternalId = "Jeff",
        Users = new[] { "Jeff" }
    }));

var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        IsDisabled = false,
        ExternalId = "Jeff",
        Users = new[] { "Bob" }
    })).CommerceCustomer;

customer.IsDisabled = true;
customer.Name = "Bobby";

var request = new UpdateCustomerRequest(customer);
var result = customerService.UpdateCustomer(request);
```

DisableCustomer

Name:	DisableCustomer
Description:	Disables a customer account. Calls the pipeline "DisableCustomer"
Usage:	Called from Sitecore when the customer account should be disabled
Signature:	DisableCustomerResult DisableCustomer(DisableCustomerRequest request)
Input:	
	CommerceCustomer – Mandatory An instance of the customer object is parsed in
	Comment – Optional An optional string that can explain why the user account was disabled. Will be put in the Page Event as explanation
Output:	
	CommerceCustomer – The disabled customer
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        IsDisabled = false,
        ExternalId = "Jeff",
        Users = new[] { "Bob" }
    })).CommerceCustomer;

var disableRequest = new DisableCustomerRequest(customer);
var disableResult = customerService.DisableCustomer(disableRequest);

var enableRequest = new EnableCustomerRequest(customer);
var enableResult = customerService.EnableCustomer(enableRequest);
```

EnableCustomer

Name:	EnableCustomer
Description:	Enables a customer account. Calls the pipeline "EnableCustomer"
Usage:	Called from Sitecore when the customer account should be enabled
Signature:	`EnableCustomerResult EnableCustomer(EnableCustomerRequest request)`
Input:	
	CommerceCustomer – Mandatory An instance of the customer object is parsed in
	Comment – Optional An optional string that can explain why the user account was enabled. Will be put in the Page Event as explanation
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        IsDisabled = false,
        ExternalId = "Jeff",
        Users = new[] { "Bob" }
    })).CommerceCustomer;

var disableRequest = new DisableCustomerRequest(customer);
var disableResult = customerService.DisableCustomer(disableRequest);

var enableRequest = new EnableCustomerRequest(customer);
var enableResult = customerService.EnableCustomer(enableRequest);
```

DeleteCustomer

Name:	DeleteCustomer
Description:	Deletes a customer account. Calls the pipeline "`DeleteUser`".
Usage:	Called when an account should be deleted It's a solution business decision whether the account is actually deleted or simply disabled
Signature:	`DeleteCustomerResult DeleteCustomer(DeleteCustomerRequest request)`
Input:	
	CommerceCustomer – An instance of the CommerceCustomer object is parsed in
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
```

```
        IsDisabled = false,
        ExternalId = "Jeff",
        Users = new[] { "Bob" }
    })).CommerceCustomer;

var deleteRequest = new DeleteCustomerRequest(customer);
var deleteResult = customerService.DeleteCustomer(deleteRequest);
```

GetCustomer

Name:	GetCustomer
Description:	Returns a single customer instance. Calls the pipeline "GetCustomer".
Usage:	Called from Sitecore when searching for an account
Signature:	GetCustomerResult GetCustomer(GetCustomerRequest request)
Input:	
	ShopName – Mandatory
	ExternalID – Mandatory The unique ID of the customer in the given shop.
Output:	
	CommerceCustomer – An instance of the CommerceCustomer if it exists
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        IsDisabled = false,
        ExternalId = "1234567890",
        Users = new[] { "Bob" }
    }));

var getRequest = new GetCustomerRequest("1234567890");
var result = customerService.GetCustomer(getRequest);
```

GetCustomers

Name:	GetCustomers
Description:	Queries and returns customer entities. Calls the pipeline "GetCustomers". Different input parameters can be provided and they will be combined using logical *and*
Usage:	Called from Sitecore when searching for an account
Signature:	GetCustomersResult GetCustomers(GetCustomersRequest request)
Input:	
	ExternalIDs – optional Can be a single or a list of IDs. When provided, it takes precedence
	SitecoreIDs – optional Can be a single or a list of IDs.
	Name – optional
	ExternalUserIDs – Optional. Can be a single or a list of IDs. Used when looking for customers associated with a given user
	Disabled – optional
	ShopName – optional
Output:	

	List<CommerceCustomer>
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
        IsDisabled = true,
        ExternalId = "Jeff",
        Users = new[] { "Jeff" }
    }));
customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        IsDisabled = false,
        ExternalId = "Jeff",
        Users = new[] { "Bob" }
    }));
customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Michael",
        Shops = new[] { "autohaus" },
        IsDisabled = false,
        ExternalId = "Michael",
        Users = new[] { "Michael" }
    }));
customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Kerry",
        Shops = new[] { "webstore" },
        IsDisabled = true,
        ExternalId = "Michael",
        Users = new[] { "Michael" }
    }));

var result = customerService.GetCustomers(new GetCustomersRequest(new
CustomerSearchCriteria { IsDisabled = true }));
Sitecore.Diagnostics.Assert.AreEqual(2, result.CommerceCustomers.Count, "Should have 2
disabled");

result = customerService.GetCustomers(new GetCustomersRequest(new
CustomerSearchCriteria { ExternalIDs = new[] { "Jeff" }, Name = "Bob" }));
Sitecore.Diagnostics.Assert.AreEqual(2, result.CommerceCustomers.Count, "Should have 1
match");
```

AddCustomers

Name:	**AddCustomers**
Description:	Add a customer to a user. Calls the pipeline "`AddCustomers`"
Usage:	Called to associated a visitor to a user
Signature:	`AddCustomersResult AddCustomers(AddCustomersRequest request)`
Input:	
	List<string> CustomerIds – the customer ids to add to the user
Output:	
	IReadOnlyCollection<string> CustomerIds – the list of customer ids associated with the user
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var customerService = new CustomerServiceProvider();

    // create a customer
    customerService.CreateCustomer(new CreateCustomerRequest(
        new CommerceCustomer
        {
            Name = "Jeff",
            Shops = new[] { "autohaus" },
            ExternalId = "1234567890"
        }));

    // create a user
    var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
    var user = customerService.CreateUser(request).CommerceUser;

    // add the customer to the user
    var addRequest = new AddCustomersRequest(user, new List<string> { "1234567890" });
    var addResult = customerService.AddCustomers(addRequest);
```

AddUsers

Name:	AddUsers
Description:	Add a user to a customer. Calls the pipeline "`AddUsers`"
Usage:	Called to associated a user to a visitor
Signature:	`AddUsersResult AddUsers(AddUsersRequest request)`
Input:	
	List<string> UserIds – the user ids to add to the customer
Output:	
	IReadOnlyCollection<string> UserIds – the list of user ids associated with the customer
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var customerService = new CustomerServiceProvider();

    // create a user
    var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
    var user = customerService.CreateUser(request).CommerceUser;

    // create a customer
    var customer = customerService.CreateCustomer(new CreateCustomerRequest(
        new CommerceCustomer
        {
            Name = "Jeff",
            Shops = new[] { "autohaus" },
            ExternalId = "1234567890"
        })).CommerceCustomer;

    // add the user to the customer
    var addRequest = new AddUsersRequest(customer, new List<string> { user.ExternalId });
    var addResult = customerService.AddUsers(addRequest);
```

RemoveCustomers

Name:	RemoveCustomers
Description:	Removes customers from a user. Calls the pipeline "`RemoveCustomers`"
Usage:	Called to remove an associated visitor to a user
Signature:	`RemoveCustomersResult` `RemoveCustomers(RemoveCustomersRequest request)`

Input:	
	List<string> CustomerIds – the customer ids to remove from the user
Output:	
	IReadOnlyCollection<string> CustomerIds – the list of customer ids associated with the user
	SystemMessages - Collection of messages from the external system.

Usage Example:

```csharp
var customerService = new CustomerServiceProvider();

// create 2 customers
customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "webstore" },
        ExternalId = "1234567890"
    }));
customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Bob",
        Shops = new[] { "webstore" },
        ExternalId = "9876543210"
    }));

// create a user
var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
var user = customerService.CreateUser(request).CommerceUser;

// add the customers to the user
var addRequest = new AddCustomersRequest(user, new List<string> { "1234567890",
"9876543210" });
var addResult = customerService.AddCustomers(addRequest);

// remove a customer
var removeRequest = new RemoveCustomersRequest(user, new List<string> { "1234567890"
});
var removeResult = customerService.RemoveCustomers(removeRequest);
```

RemoveUsers

Name:	RemoveUsers
Description:	Removes users from a customer. Calls the pipeline "RemoveUsers"
Usage:	Called to remove an associated user to a visitor
Signature:	RemoveUsersResult RemoveUsers(RemoveUsersRequest request)
Input:	
	List<string> UserIds – the user ids to remove from the customer
Output:	
	IReadOnlyCollection<string> UserIds – the list of user ids associated with the customer
	SystemMessages - Collection of messages from the external system.

Usage Example:

```csharp
var customerService = new CustomerServiceProvider();

// create 2 users
var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
var user = customerService.CreateUser(request).CommerceUser;
request = new CreateUserRequest("JaneSmith", "passWord", "jane@jane.net", "webstore");
var user2 = customerService.CreateUser(request).CommerceUser;

// create a customer
var customer = customerService.CreateCustomer(new CreateCustomerRequest(
```

```
            new CommerceCustomer
            {
                Name = "Jeff",
                Shops = new[] { "autohaus" },
                ExternalId = "1234567890"
            })).CommerceCustomer;

        // add the users to the customer
        var addRequest = new AddUsersRequest(customer, new List<string> { user.ExternalId,
user2.ExternalId });
        var addResult = customerService.AddUsers(addRequest);

        // remove a user
        var removeRequest = new RemoveUsersRequest(customer, new List<string> {
user.ExternalId });
        var removeResult = customerService.RemoveUsers(removeRequest);
```

AddCustomerParties

Name:	**AddCustomerParties**
Description:	This method is responsible for adding one or more given customer parties to the specified customer
Usage:	Called from Sitecore when adding parties to a customer account, typically during checkout or editing the customer account
Signature:	AddCustomerPartiesResult AddCustomerParties (AddCustomerPartiesRequest request)
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
	CustomerParties – List<CustomerParty>**.** Mandatory A list of customer parties to associate with the customer
Output:	
	Customer - Customer. Mandatory
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
        var customerService = new CustomerServiceProvider();

        // create a customer
        var customer = customerService.CreateCustomer(new CreateCustomerRequest(
            new CommerceCustomer
            {
                Name = "Jeff",
                Shops = new[] { "autohaus" },
                ExternalId = "1234567890"
            })).CommerceCustomer;

        // create the add request
        var request = new AddCustomerPartiesRequest(customer,
            new List<CustomerParty>
            {
                new CustomerParty {ExternalId = "1234567890", PartyID = "1", Name =
"HomeAddess", PartyType = 2}
            });

        var result = customerService.AddCustomerParties(request);
```

RemoveCustomerParties

Name:	**RemoveCustomerParties**
Description:	This method is responsible for removing one or more given customer parties from the specified customer
Usage:	Called from Sitecore when removing parties to a customer account, typically when editing the customer account

Signature:	RemoveCustomerPartiesResult RemoveCustomerParties (RemoveCustomerPartiesRequest request)
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
	Parties – List<Party>**.** Mandatory A list of customer parities to un-associate with the customer
Output:	
	Customer - Customer. Mandatory
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create a customer
var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
        ExternalId = "1234567890"
    })).CommerceCustomer;

var parties = new List<CustomerParty>
{
    new CustomerParty {ExternalId = "1234567890", PartyID = "1", Name = "HomeAddess",
PartyType = 2}
};

// create the add request
var request = new AddCustomerPartiesRequest(customer, parties);
var result = customerService.AddCustomerParties(request);

// remove the parties
var removeRequest = new RemoveCustomerPartiesRequest(customer, parties);
var removeResult = customerService.RemoveCustomerParties(removeRequest);
```

UpdateCustomerParties

Name:	**UpdateCustomerParties**
Description:	This method is responsible for updating one or more given customer parties on the specified customer
Usage:	Called from Sitecore when updating parties on a customer account, typically when editing the customer account
Signature:	UpdateCustomerPartiesResult UpdateCustomerParties (UpdateCustomerPartiesRequest request)
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
	Parties – List<Party>**.** Mandatory A list of customer parities to update on the customer
Output:	
	Customer - Customer. Mandatory
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create a customer
var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
```

```
            ExternalId = "1234567890"
        })).CommerceCustomer;

    var parties = new List<CustomerParty>
    {
        new CustomerParty {ExternalId = "1234567890", PartyID = "1", Name = "HomeAddess",
PartyType = 2}
    };

    // create the add request
    var request = new AddCustomerPartiesRequest(customer, parties);
    var result = customerService.AddCustomerParties(request);

    parties.Add(new CustomerParty { ExternalId = "9876543210", PartyID = "2", Name =
"BillingAddess", PartyType = 1 });

    // update the parties
    var updateRequest = new UpdateCustomerPartiesRequest(customer, parties);
    var updateResult = customerService.UpdateCustomerParties(updateRequest);
```

AddParties

Name:	AddParties
Description:	This method is responsible for adding one or more given customer parties to the specified customer
Usage:	Called from Sitecore when adding parties to a customer account, typically during checkout or editing the customer account
Signature:	AddPartiesResult AddParties(AddPartiesRequest request)
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
	Parties – List<Party>. Mandatory A list of customer parties to add
Output:	
	Customer - Customer. Mandatory
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var customerService = new CustomerServiceProvider();

    // create a customer
    var customer = customerService.CreateCustomer(new CreateCustomerRequest(
        new CommerceCustomer
        {
            Name = "Jeff",
            Shops = new[] { "autohaus" },
            ExternalId = "1234567890"
        })).CommerceCustomer;

    var parties = new List<Party>
    {
        new Party {ExternalId = "1234567890", Address1 = "123 My Street", City = "My
City", Country = "My Country", PartyId = "1"}
    };

    // add the party
    var addRequest = new AddPartiesRequest(customer, parties);
    var addResult = customerService.AddParties(addRequest);
```

GetParties

Name:	GetParties
Description:	This method is responsible for getting all the parties

Usage:	Called from Sitecore when adding parties to a customer account, typically during checkout or editing the customer account
Signature:	`AddPartiesResult AddParties(AddPartiesRequest request)`
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
Output:	
	Parties – List<Party> A list of parties
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create a customer
var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
        ExternalId = "1234567890"
    })).CommerceCustomer;

var parties = new List<Party>
{
    new Party {ExternalId = "1234567890", Address1 = "123 My Street", City = "My
City", Country = "My Country", PartyId = "1"}
};

// add the party
var addRequest = new AddPartiesRequest(customer, parties);
var addResult = customerService.AddParties(addRequest);

// get the party
var getRequest = new GetPartiesRequest(customer);
var getResult = customerService.GetParties(getRequest);
```

RemoveParties

Name:	**RemoveParties**
Description:	This method is responsible for removing one or more given customer parties from the specified customer
Usage:	Called from Sitecore when removing parties to a customer account, typically when editing the customer account
Signature:	`RemovePartiesResult RemoveParties(RemovePartiesRequest request)`
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
	Parties – List<Party>**.** Mandatory A list of customer parties to remove
Output:	
	Customer - Customer. Mandatory
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
var customerService = new CustomerServiceProvider();

// create a customer
var customer = customerService.CreateCustomer(new CreateCustomerRequest(
    new CommerceCustomer
    {
        Name = "Jeff",
        Shops = new[] { "autohaus" },
```

```
            ExternalId = "1234567890"
        })).CommerceCustomer;

    var parties = new List<Party>
    {
        new Party {ExternalId = "1234567890", Address1 = "123 My Street", City = "My
City", Country = "My Country", PartyId = "1"}
    };

    // add the party
    var addRequest = new AddPartiesRequest(customer, parties);
    var addResult = customerService.AddParties(addRequest);

    // remove the party
    var removeRequest = new RemovePartiesRequest(customer, parties);
    var getResult = customerService.RemoveParties(removeRequest);
```

UpdateParties

Name:	UpdateParties
Description:	This method is responsible for updating one or more given customer parties on the specified customer
Usage:	Called from Sitecore when updating parties on a customer account, typically when editing the customer account
Signature:	`UpdatePartiesResult UpdateParties(UpdatePartiesRequest request)`
Input:	
	Customer – CommerceCustomer. Mandatory An instance of the customer
	Parties – List<Party>**.** Mandatory A list of parties to update
Output:	
	Customer - Customer. Mandatory
	SystemMessages - Collection of messages from the external system.

Usage Example:

```
    var customerService = new CustomerServiceProvider();

    // create a customer
    var customer = customerService.CreateCustomer(new CreateCustomerRequest(
        new CommerceCustomer
        {
            Name = "Jeff",
            Shops = new[] { "autohaus" },
            ExternalId = "1234567890"
        })).CommerceCustomer;

    var parties = new List<Party>
    {
        new Party {ExternalId = "1234567890", Address1 = "123 My Street", City = "My
City", Country = "My Country", PartyId = "1"}
    };

    // add the party
    var addRequest = new AddPartiesRequest(customer, parties);
    var addResult = customerService.AddParties(addRequest);

    parties.Add(new Party { ExternalId = "9876543210", Address1 = "123 My Road", City =
"My Town", Country = "Your Country", PartyId = "2" });

    // update the parties
    var updateRequest = new UpdatePartiesRequest(customer, parties);
    var getResult = customerService.UpdateParties(updateRequest);
```

UpdatePassword

Name:	**UpdatePassword**
Description:	Change the user password. Calls the pipeline "UpdatePassword"
Usage:	Called from Sitecore when password needs to be changed.
Signature:	`UpdatePasswordResult UpdatePassword(UpdatePasswordRequest request)`
Input:	
	UserID – Mandatory The ID of the user to change the password for
	OldPassword – **Mandatory**. The old password
	NewPassword – Mandatory The new password
Output:	
	SystemMessages - Collection of messages from the external system.

Usage Example:

```csharp
var customerService = new CustomerServiceProvider();

// create a customer
var request = new CreateUserRequest("JohnSmith", "password", "john@abczyx.net",
"webstore");
customerService.CreateUser(request);

// update the password
var updateRequest = new UpdatePasswordRequest("JohnSmith", "password", "nEwPaSsWoRd");
var result = customerService.UpdatePassword(updateRequest);
```

2.5.3   Customer Pipelines

CreateUser

Depending on the actual integration with ECS and the solution then the user can be disabled by default and will get activated when ActivateUserAccount is called

Name:	**CreateUser**
Description:	This pipeline is responsible for creating a user account
Usage:	Called from Sitecore.
Args:	
	Request - Contains the user entity and a password. Is set prior to calling the pipeline.
	Response - Contains the user object. Is read after the pipeline is called.
Processors:	
	CreateUserInECS – Optional Creates a user in the external commerce system and updates the field ExternalID on the user entity. **Note:** If External ID is parsed in and the user already exists, nothing should happen

Note: This processor is optional but recommended if CommerceUsers are supported in the ECS.

CreateUserInCMS – (Mandatory)

Creates the user but more importantly stores the relation to CommerceCustomer in Sitecore, using the membership and profile providers.

If user already exists, then the pipeline should be aborted to avoid triggering the goal.

Note: This processor is mandatory to store the 1-to-many relationship between the CommerceUser and CommerceCustomer. If that is handled in the ECS, the processor is not mandatory

TriggerDMSGoal –

The goal "User Account Created" is triggered with values User name and ShopName.

AddVisitorToEAPlan –

Adds visitor / contact to EA plan. For example, "User Account Registered", which sends an account activation mail and later a welcome mail when the account has been activated

UpdateUser

Name:	UpdateUser
Description:	This pipeline is responsible for updating an existing user account.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the user entity. Is set prior to calling the pipeline.
	Response - Contains the user entity. Is read after the pipeline is called.
Processors:	

UpdateUserInECS – Optional

Updates an existing user account in the external commerce system.

Note: This processor is optional but recommended if CommerceUsers are supported in the ECS.

UpdateUserInCMS – Mandatory

Updates the user account in the Sitecore. Since the user is updated externally first, then the user data can be changed there and the final version is stored in CMS.

Note: This processor is mandatory to store the 1-to-many relationship between the CommerceUser and CommerceCustomer. If that is handled in the ECS, the processor is not mandatory

TriggerDMSEvent – the page event "User Account Updated" is triggered with values User name and ShopName.

DeleteUser

Name:	DeleteUser
Description:	This pipeline is responsible for deleting an existing user account.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the user entity. Is set prior to calling the pipeline.
	Response -
Processors:	
	DeleteUserInECS – Optional Deletes an existing user in the external commerce system. **Note:** If user account doesn't exists it can be noted in returned collection of External System Messages, but the pipeline must continue **Note:** This processor is optional but recommended if CommerceUsers are supported in the ECS.
	DeleteUserInCMS – Mandatory Deletes the user in the Sitecore. **Note:** If user account doesn't exist, the pipeline is aborted **Note:** This processor is mandatory to store the 1-to-many relationship between the CommerceUser and CommerceCustomer. If that is handled in the ECS, the processor is not mandatory
	TriggerDMSEvent – the page event "User Account Deleted" is triggered with values User name and ShopName.

DisableUser

Name:	DisableUser
Description:	This pipeline is responsible for disabling an existing user account.
Usage:	Called from Sitecore.
Args:	

Request - Contains the user entity and a comment. Is set prior to calling the pipeline.

Response - Is read after the pipeline is called.

Processors:

DisableUserInECS – Optional

Disables an existing user in the external commerce system.

DisableUserInCMS – Mandatory

Disables the user in Sitecore.

Note: Since the user is attempted disabled externally first, then it is possible that the pipeline is aborted due to business rules and the users are still synchronized

Note: This processor is mandatory to store the 1-to-many relationship between the CommerceUser and CommerceCustomer. If that is handled in the ECS, the processor is not mandatory

TriggerDMSEvent – the page event "User Account Disabled" is triggered with values User name, ShopName and the provided comment.

EnableUser

Name:	EnableUser
Description:	This pipeline is responsible for enabling an existing user account.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the user entity and a comment. Is set prior to calling the pipeline.
	Response - Is read after the pipeline is called.
Processors:	

EnableUserInECS – Optional

Enables an existing user in the external commerce system.

Note: Either this processor or the next should be in the pipeline, normally not both

EnableUserInCMS – Mandatory

Enables the user in Sitecore.

Note: Since the user is attempted enabled externally first, then it is possible that the pipeline is aborted due to business rules and the users are still synchronized

Note: This processor is mandatory to store the 1-to-many relationship between the CommerceUser and CommerceCustomer. If that is handled in the ECS, the processor is not mandatory

TriggerDMSEvent – the goal "User Account Enabled" is triggered with values User name, ShopName and the provided comment.

GetUsers

Name:	GetUsers
Description:	Queries and returns user accounts.
	Different input parameters can be provided and they will be combined using logical *and*
Usage:	Called from Sitecore when a visitor is registering for an account. It could be during the checkout process or through plain signup.
Args:	
	Request - Contains search parameters. Is set prior to calling the pipeline.
	Response – Returns a list of user entities. Is read after the pipeline is called.
Processors:	
	GetUsersFromECS – Optional
	Queries against users in the external commerce system.
	Note: Either this processor or the next should be in the pipeline, normally not both
	Note: This processor is optional but recommended if CommerceUsers are supported in the ECS.
	GetUsersFromCMS – Mandatory
	Queries against users in CMS.
	Note: Either this processor or the next should be in the pipeline, normally not both

GetUser

Name:	GetUser
Description:	Returns a single user account.
Usage:	Called from Sitecore when a visitor a specific user account is needed
Args:	
	Request - Contains ShopName and User ID. Is set prior to calling the pipeline.
	Response – Returns a single user entity. Is read after the pipeline is called.
Processors:	

GetUserFromECS – Optional

Gets the specified user from the external commerce system.

Note: Either this processor or the next should be in the pipeline, normally not both

Note: This processor is optional but recommended if CommerceUsers are supported in the ECS.

GetUserFromCMS – Mandatory

Gets the specified user from CMS.

Note: Either this processor or the next should be in the pipeline, normally not both

CreateCustomer

Name:	CreateCustomer
Description:	This pipeline is responsible for creating a customer account. The customers are managed by the commerce system.
Usage:	Called from Sitecore
Args:	
	Request - Contains the customer entity. Is set prior to calling the pipeline.
	Response - Contains the customer entity after the pipeline is called.
Processors:	

	CreateCustomerInECS – Optional
	
	Depends on whether Customers are supported in the ECS
	
	Creates customer in external commerce system.
	
	It involves:
	
	  o Create unique customer account in external system
	
	  o Create mapping between Customer and any given user accounts
	
	Note: If External ID is parsed in and the customer already exists, nothing should happen

	CreateCustomerInSitecore – (Mandatory) Creates customer in Sitecore **Note:** This processor is mandatory to store the 1-to-many relationship between the CommerceUser and CommerceCustomers. If that is handled in the ECS, the processor is not mandatory
	AddCustomerToEAplan– Adds the customer to EA plan Customers storing a reference to the CommerceCustomer in EA state

Since the customer potentially is created externally first, then the external ID of the customer is given and can be stored in Sitecore too, making the connection between the two.

Note: If External ID is parsed in and the customer already exists, then the pipeline should be aborted to avoid triggering the goal.

TriggerDMSGoal – the goal "Customer Account Created" is triggered with values customer name and ShopName

GetCustomers

Name:	GetCustomers
Description:	Queries and returns customer accounts.
	Different input parameters can be provided and they will be combined using logical *and*
	Whether the customers are retrieved from ECS or CMS depends on the support of customers in ECS
Usage:	Called from Sitecore
Args:	
	Request - Contains search parameters.. Is set prior to calling the pipeline.
	Response - Returns a list of customer entities after the pipeline is called.
Processors:	
	GetCustomersFromECS – optional
	Queries against customers in the external commerce system. If the required search functionality is not supported in the ECS, then it can potentially be handled by searching for customers in CMS.
	Note: Either this processor or the next should be in the pipeline, normally not both
	GetCustomersFromCMS – optional
	Queries against Customers in CMS. If the required search functionality is not supported in CMS, then it can potentially be handled by searching for users in the ECS
	Note: Either this processor or the previous should be in the pipeline, normally not both

GetCustomer

Name:	GetCustomer
Description:	Returns the single customer entity with the specified ID

	Whether the customer is retrieved from ECS or CMS depends on the support of customers in ECS
Usage:	Called from Sitecore
Args:	
	Request - Contains shop name and Customer ID. Is set prior to calling the pipeline.
	Response - Returns a single customer entity after the pipeline is called.
Processors:	
	GetCustomerFromECS – optional Gets the specified customer from the external commerce system. **Note:** Either this processor or the next should be in the pipeline, normally not both
	GetCustomersFromCMS – Mandatory Gets the specified customer from CMS. **Note:** Either this processor or the previous should be in the pipeline, normally not both

UpdateCustomer

Name:	**UpdateCustomer**
Description:	This pipeline is responsible for updating an existing customer account. The customers are managed by the commerce system.
Usage:	Called from Sitecore
Args:	
	Request - Contains the customer entity. Is set prior to calling the pipeline.
	Response - Contains the customer object. Is read after the pipeline is called.
Processors:	
	UpdateCustomerInECS – Mandatory Updates customer in external commerce system.
	SaveCustomertoEAState – mandatory Updates the customer stored in EA state based on updated CommerceCustomer returned from previous processor
	TriggerDMSEvent – the goal "Customer Account Updated" is triggered with values customer name and ShopName

DeleteCustomer

Name:	**DeleteCustomer**

Description:	This pipeline is responsible for deleting an existing customer account. The customers are managed by the commerce system.
Usage:	Called from Sitecore
Args:	
	Request - Contains the customer entity. Is set prior to calling the pipeline.
	Response -
Processors:	
	DeleteCustomerInECS – Mandatory Deletes customer in external system. **Note:** If customer account doesn't exists it can be noted in returned collection of External System Messages, but the pipeline must continue
	RemoveCustomerFromEAState – Mandatory Removes the customer stored in EA state
	TriggerDMSEvent – the goal "Customer Account Deleted" is triggered with values customer name and ShopName

DisableCustomer

Name:	**DisableCustomer**
Description:	This pipeline is responsible for disabling an existing customer account.
Usage:	Called from Sitecore.
Args:	
	Request - Contains the customer entity and a comment. Is set prior to calling the pipeline.
	Response – The disabled customer entity. Is read after the pipeline is called.
Processors:	
	DisableCustomerInECS – Disables an existing customer in the external commerce system.
	SaveCustomertoEAState – mandatory Updates the customer stored in EA state based on updated CommerceCustomer returned from previous processor
	TriggerDMSEvent – the page event "Customer Account Disabled" is triggered with values Name, ShopName and the provided comment.

EnableCustomer

Name:	**EnableCustomer**
Description:	This pipeline is responsible for enabling an existing customer account.
Usage:	Called from Sitecore.

Args:	
	Request - Contains the customer entity and a comment. Is set prior to calling the pipeline.
	Response – The enabled customer entity**.** Is read after the pipeline is called.
Processors:	
	EnableCustomerInECS – Enables an existing customer in the external commerce system.
	SaveCustomertoEAState – mandatory Updates the customer stored in EA state based on updated CommerceCustomer returned from previous processor
	TriggerDMSEvent – the pageevent "Customer Account Enabled" is triggered with values Name, ShopName and the provided comment.

AddCustomerParties

Name:	AddCustomerParties
Description:	This pipeline is responsible for adding customer parties to the specified customer
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer and a list of CustomerParty instances. Is set prior to calling the pipeline.
	Response – A new instance of the customer Is read after the pipeline is called.
Processors:	
	AddCustomerParties – Responsibility: Is to add the provided parties to the customer account and persist them Usage: Mandatory. Ownership: The processor is provided with Connect and stores the parties with the customer account using the Sitecore membership provider Customization: No immediate need to customize. There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

RemoveCustomerParties

Name:	RemoveCustomerParties
Description:	This pipeline is responsible for removing customer parties from the specified customer
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer and a list of CustomerParty instances. Is set prior to calling the pipeline.
	Response – A new instance of the customer Is read after the pipeline is called.
Processors:	
	RemoveCustomerParties – Responsibility: Is to remove the provided parties from the customer account Usage: Mandatory. Ownership: The processor is provided with Connect and removes the parties which will no longer be persisted anywhere in Sitecore Customization: No immediate need to customize. There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

UpdateCustomerParties

Name:	UpdateCustomerParties
Description:	This pipeline is responsible for updating customer parties on the specified customer
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer and a list of CustomerParty instances. Is set prior to calling the pipeline.
	Response – A new instance of the customer Is read after the pipeline is called.
Processors:	
	UpdateCustomerParties – Responsibility: Is to update the provided parties on the customer account and persist them

Usage: Mandatory.

Ownership: The processor is provided with Connect and stores the updated parties with the customer account using the Sitecore membership provider

Customization: No immediate need to customize.

There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

GetParties

Name:	GetParties
Description:	This pipeline is responsible for getting the parties
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer. Is set prior to calling the pipeline.
	Response – A list of parties. Is read after the pipeline is called.
Processors:	
	GetParties – Responsibility: Is to return the complete list of parties stored with the customer Usage: Mandatory. Ownership: The processor is provided with Connect and stores the parties with the customer account using the Sitecore membership provider Customization: No immediate need to customize. There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

AddParties

Name:	**AddParties**
Description:	This pipeline is responsible for adding parties to the specified customer
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer and a list of CustomerParty instances. Is set prior to calling the pipeline.
	Response – A new instance of the customer Is read after the pipeline is called.
Processors:	
	AddParties – Responsibility: Is to add the provided parties to the customer account and persist them Usage: Mandatory. Ownership: The processor is provided with Connect and stores the parties with the customer account using the Sitecore membership provider Customization: No immediate need to customize. There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

RemoveParties

Name:	**RemoveParties**
Description:	This pipeline is responsible for removing parties stored with the specified customer
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer and a list of party instances. Is set prior to calling the pipeline.
	Response – A new instance of the customer Is read after the pipeline is called.
Processors:	
	RemoveParties – Responsibility: Is to remove the provided parties from the customer account Usage: Mandatory.

Ownership: The processor is provided with Connect and removes the parties which will no longer be persisted anywhere in Sitecore

Customization: No immediate need to customize.

There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

UpdateParties

| Name:	UpdateParties
Description:	This pipeline is responsible for updating customer parties on the specified customer
Usage:	Called from Sitecore.
Args:	
	Request - Contains a customer and a list of CustomerParty instances. Is set prior to calling the pipeline.
	Response – A new instance of the customer Is read after the pipeline is called.
Processors:	
	UpdateParties – Responsibility: Is to update the provided parties on the customer account and persist them Usage: Mandatory. Ownership: The processor is provided with Connect and stores the updated parties with the customer account using the Sitecore membership provider Customization: No immediate need to customize. There should be a separate processor for storing the parties in the ECS. The processor should either replace this processor or be added in addition to this processor

2.6 Product Sync

2.6.1   The Product Sync Domain Model

This chapter describes the domain model that represents the product objects where the following three assertions holds true:

- The domain model primarily serves as DTO objects for transferring information between the external commerce system and Sitecore.
- The domain model contains the minimum required information as contracts and will typically be extended, when integrated with a specific commerce system.
- The domain model is used when a Sitecore developer needs to synchronize product data with the external commerce system or product data is pushed into Sitecore from the external commerce system

The class diagram below shows the domain model.
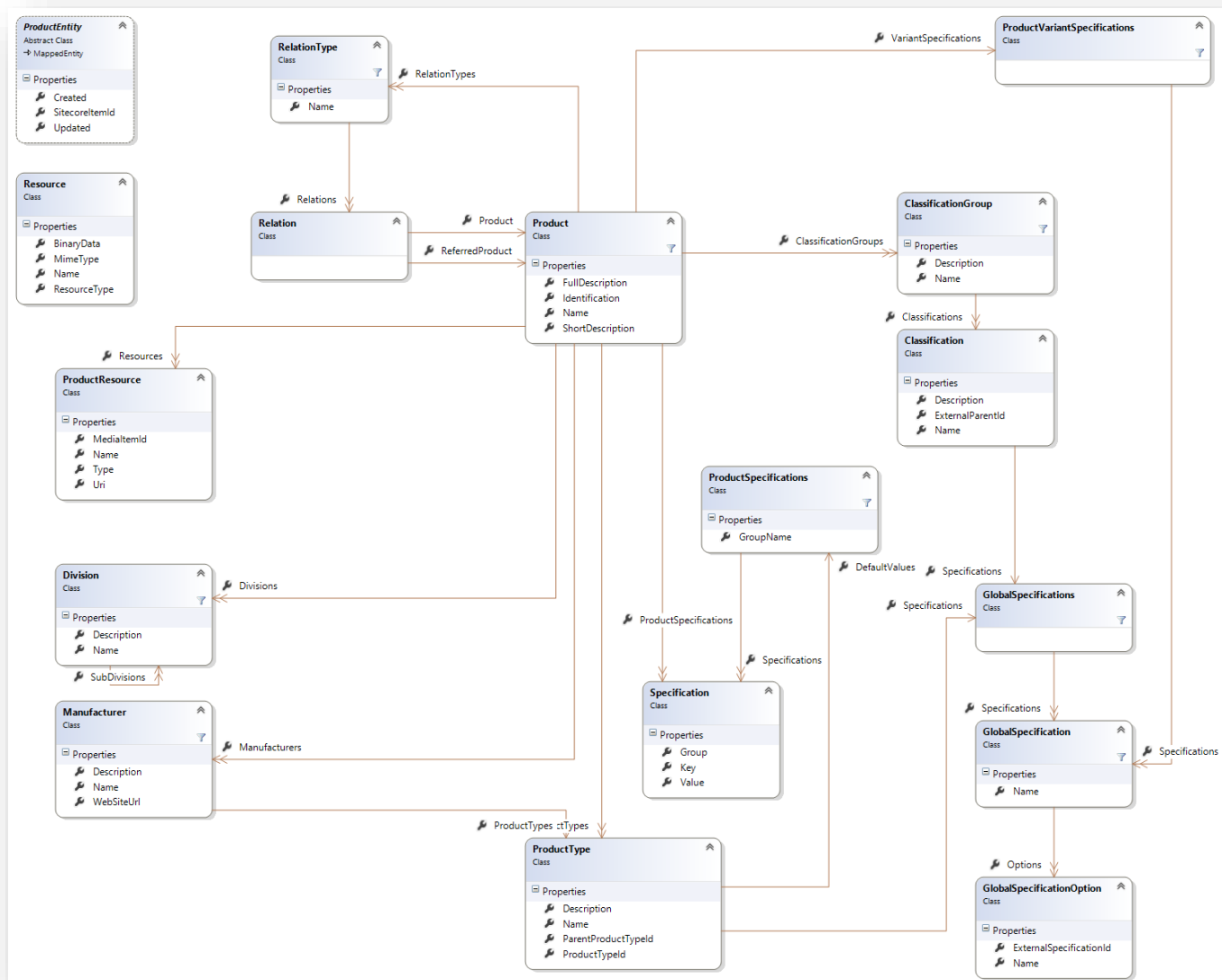
Note: The domain model consists of abstract classes that make up the contracts with the external system. The contracts are defined as abstract classes instead of interfaces to allow the model to be easily extended later if needed. This follow the best practice guidelines defined in the book Framework Design Guidelines.

Default implementation of the contracts are delivered as part of Connect. If an actual Connect provider with an external commerce system contains more functionality that provided by default, the implementation can be replaced. All instantiation of actual classes will be handled through dependency injection.

Class: Product

The product class is responsible for representing a product or any variant of it, hence a variant is a product in this model.

Name	Type	Description
ExternalId	string	Unique identifier for the product in the commerce system. This can be used to get a reference to the product using the commerce system's native API.
SitecoreItemId	string	Returns the Sitecore ID
Name	string	Name of the product
ShortDescription	string	The short description of the product.
FullDescription	string	The full description of the product.
ProductType	ProductType	A reference to the product type
Manufacturers	Manufacturer	Reference to the manufacturers
ClassificationGroups	IReadOnlyCollection<ProductClassificationGroup>	Reference to the associated classifications and categories
Specifications	ProductSpecifications	Collection of specifications set directly on the product
VariantSpecifications	ProductVariantSpecifications	List of references to specifications that tells the variants apart and which potentially can be selectable to the visitor
Resources	IReadOnlyCollection<ProductResource>	Reference to the associated resources
Divisions	IReadOnlyCollection<Division>	Reference to the associated divisions
RelationTypes	IReadOnlyCollection<ProductRelationType>	Reference to the related products
Created	DateTime	Date of creation
Updated	DateTime	Date of last update

Class: ProductSpecifications

Name	Type	Description

ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Specifications	IReadOnlyCollection<ProductSpecification>	
Created	DateTime	
Updated	DateTime	

Class: ProductSpecification

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Group	String	
Key	String	
Value	String	
Created	DateTime	
Updated	DateTime	

Class: ProductClassification

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
ExternalParentId	string	
SitecoreItemId	string	Represents the Sitecore Id.
Name	string	
Description	string	
Specifications	ProductSpecifications	
Created	DateTime	
Updated		

Class: ProductType

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
ProductTypeId	String	
Description	String	
Specifications	ProductSpecifications	
Created	DateTime	
Updated	DateTime	

Class: ProductManufacturer

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Name	String	
Description	String	

WebSiteUrl	String
ProductTypes	IReadOnlyCollection<ProductType>
Created	DateTime
Updated	DateTime

Class: ProductClassificationGroup

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Name	String	
Description	String	
Classifications	IReadOnlyCollection<ProductClassification>	
Created	DateTime	
Updated	Name	

Class: ProductVariantSpecificaions

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Specifications	IReadOnlyCollection<ProductSpecification>	
Created	DateTime	
Updated	DateTime	

Class: ProductResource

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Created	DateTime	
Updated	DateTime	

Class: Division

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Name	String	
SubDivisions	IReadOnlyCollection<Division>	
Created	DateTime	
Updated	DateTime	

Class: ProductRelation

Name	Type	Description

ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Product	Product	
ReferredProduct	Product	
Created	DateTime	
Updated	DateTime	

Class: ProductRelationType

Name	Type	Description
ExternalId	string	Unique identifier used to identify the item in an external system.
SitecoreItemId	string	Represents the Sitecore Id.
Name	String	
Relations	IReadOnlyCollection<ProductRelation>	
Created	DateTime	
Updated	DateTime	

2.6.2   Product Sync Service Methods

Service providers are wrapper objects designed to make it easier to interact with Connect pipelines. The providers implement no logic other than calling Connect pipelines. All of the business logic is implemented in the pipeline processors.

The Product Sync Service Provider contains the following methods for interacting with product sync data.

SynchronizeProducts

SyncronizeProducts is used to synchronize a collection of products between the external commerce system and Sitecore.

The synchronization can go both ways, so changes made to product data in CMS content are pushed to the external commerce system.

A log must be kept of events registered during synchronization. At minimum it should contain a list of products successfully updated. It would be better

Upon return the result contains the list of messages generated during synchronization, which would be the Ids of the products that failed during synchronization

Name:	**SynchronizeProducts**
Description:	SynchronizeProducts calls the pipeline " SynchronizeProducts" to synchronize changes to all updated products and related repositories
Usage:	Called from Sitecore or the ECS when the product manager wants to update products both from and to the ECS
Signature:	`SynchronizeProductsResult` `SynchronizeProducts(SynchronizeProductsRequest request)`
Input:	
	Language – string, optional. The language for the product data being synchronized. Default is English ("en" or "US-EN")

	Direction – optional**.** An enum type indicating whether synchronization goes from ECS -> SC, SC -> ECS or both ways. The default is ECS -> SC. See section **Error! Reference source not found.** for more.
Output:	
	SystemMessages - Collection of messages from the external system.

SynchronizeProductList

Name:	**SynchronizeProductList**
Description:	SynchronizeProductList calls the pipeline " SynchronizeProductList"
Usage:	Called from Sitecore or the ECS when the product manager wants to update a list of products both from and to the ECS
Signature:	`SynchronizeProductListResult` `SynchronizeProductList(SynchronizeProductListRequest` `request)`
Input:	
	List<ExternalProductIds> - List of strings, mandatory The list of external product ids to synchronize
	Language – string, optional. The language for the product data being synchronized. Default is English ("en" or "US-EN")
	Direction – enum, optional An enum type indicating whether synchronization goes from ECS -> SC, SC -> ECS or both ways. The default is ECS -> SC. See section **Error! Reference ource not found.** for more.
Output:	
	SystemMessages - Collection of messages from the external system.

SynchronizeProduct

SyncProduct is used to synchronize a single product between the external commerce system and Sitecore. The product to synchronize is given by ID.

The synchronization can go both ways, so changes made to product data in CMS content are pushed to the external commerce system as well.

Name:	**SynchronizeProduct**
Description:	SynchronizeProduct calls the pipeline " SynchronizeProduct"
Usage:	Called from Sitecore or the ECS when the product manager wants to update a single product both from and to the ECS
Signature:	`SynchronizeProductResult` `SynchronizeProduct(SynchronizeProductRequest request)`
Input:	
	ProductId – string, mandatory The external product id to be synchronized
	Language – string, optional. The language for the product data being synchronized. Default is English ("en" or "US-EN")
	Direction – enum, optional An enum type indicating whether synchronization goes from ECS -> SC, SC -> ECS or both ways. The default is ECS -> SC. See section **Error! Reference ource not found.** for more.
Output:	
	SystemMessages - Collection of messages from the external system.

SynchronizeArtifacts

SynchronizeArtifacts is responsible for synchronizing all the related repositories: Manufacturers, Types, Classifications, Divisions, Resources, and Specifications before the individual products are synchronized the references to repositories are updated.

Name:	SynchronizeArtifacts
Description:	SynchronizeArtifacts calls the pipeline " SynchronizeArtifacts" to synchronize all the related repositories: Manufacturers, Types, Classifications, Divisions, Resources, Specifications
Usage:	Called from Sitecore or the ECS when the product manager wants to update the product related repositories
Signature:	`SynchronizeArtifactsResult SynchronizeArtifacts (SynchronizeArtifactsRequest request)`
Input:	
	Language – string, optional. The language for the product data being synchronized. Default is English ("en" or "US-EN")
Output:	
	SystemMessages - Collection of messages from the external system.

2.6.3   Product Sync Pipelines

SynchronizeProducts

SyncronizeProducts is used to synchronize products between the external commerce system and Sitecore.

Name:	SynchronizeProducts
Description:	This pipeline is responsible for obtaining the lists of product Ids to be synchronized and iterate over them
Usage:	Called from Sitecore or the external commerce system
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization, which would be the Ids of the products that failed during synchronization Is read after the pipeline is called.
Processors:	
	RunSynchronizeArtifacts – Calls individual pipeline to synchronize all the related repositories: Manufacturers, Types, Classifications, Divisions, Resources, Specifications
	RunGetSCProductList – Obtain the list of product ids to synchronize from Sitecore. This processor can be left out if product data is only pushed from the external system
	RunGetECSProductList - Obtain the list of product ids to synchronize from ECS
	EvaluateProductListUnionToSynchronize – Creates the union of product Ids to be synchronized based on the two lists obtained from ECS and SC
	RunSynchronizeProductList – Calls individual pipeline with the evaluated list of product IDs to synchronize

SynchronizeProductList

SyncronizeProductlist is used to synchronize a given list of products between the external commerce system and Sitecore.

Name:	**SynchronizeProductList**
Description:	This pipeline is responsible for iterating over the given list of product Ids and synchronize
Usage:	Called from Sitecore or the external commerce system
Args:	
	Request – List of product Ids to synchronize. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization, which would be the Ids of the products that failed during synchronization Is read after the pipeline is called.
Processors:	
	SynchronizeProductList - Iterates over the given list of product Ids and runs pipeline SynchronizeProduct for each product

GetExternalCommerceSystemProductList

Name:	**GetExternalCommerceSystemProductList**
Description:	This pipeline is responsible for obtaining the list of product Ids to be synchronized from the external commerce system
Usage:	Called internally from SynchronizeProducts but can also be called explicitly from both ECS or SC
Args:	
	Request – No default data. Is set prior to calling the pipeline.
	Response - Contains the list of product Ids to be synchronized and SystemMessages. Is read after the pipeline is called.
Processors:	
	GetExternalCommerceSystemProductList – Get list of IDs from ECS

GetSitecoreProductList

Name:	**GetSitecoreProductList**
Description:	This pipeline is responsible for obtaining the list of product Ids to be synchronized from Sitecore
Usage:	Called internally from SynchronizeProducts but can also be called explicitly from both ECS or SC
Args:	
	Request – No default data. Is set prior to calling the pipeline.
	Response - Contains the list of product Ids to be synchronized and SystemMessages. Is read after the pipeline is called.
Processors:	
	GetSitecoreProductList – Get list of IDs from Sitecore

SynchronizeArtifacts

Name:	SynchronizeArtifacts
Description:	This pipeline is responsible for synchronizing all the related repositories: Manufacturers, Types, Classifications, Divisions, Resources, Specifications
Usage:	Called from Sitecore or the external commerce system
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization
Processors:	
	RunSynchronizeManufacturers – Calls individual pipeline to synchronize manufacturers respository
	RunSynchronizeTypes - Calls individual pipeline to synchronize types repository
	RunSynchronizeClassifications - Calls individual pipeline to synchronize classifications respository
	RunSynchronizeDivisions - Calls individual pipeline to synchronize divisions respository
	RunSynchronizeTypes - Calls individual pipeline to synchronize type repository
	RunSynchronizeResources - Calls individual pipeline to synchronize resources respository
	RunSynchronizeSpecifications - Calls individual pipeline to synchronize global, Category and type specifications

SynchronizeManufacturers

Name:	SynchronizeManufacturers
Description:	This pipeline is responsible for synchronizing all manufacturers in the separate Manufactures repository
Usage:	Called from pipeline SynchronizeArtifacts as initialization of separate repositories before synchronizing products and their references to these repositories.
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization
Processors:	
	ReadManufacturersFromSC – Optional Reads the manufacturers to synchronize from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadManufacturersFromECS – Mandatory Reads the manufacturers to synchronize from ECS
	ResolveManufacturersChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveManufacturersToECS – Optional Saves synchronized manufacturers to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveManufacturersToSC – Mandatory Saves synchronized manufacturers to SC.

SynchronizeClassifications

Name:	SynchronizeClassifications
Description:	This pipeline is responsible for synchronizing all classifications in the separate Classifications repository Since multiple different classification schemes are supported, this pipeline is responsible for synchronizing all schemes
Usage:	Called from pipeline SynchronizeArtifacts as initialization of separate repositories before synchronizing products and their references to these repositories.
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization
Processors:	
	ReadClassificationsFromSC – Optional Reads the classifications to synchronize from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadClassificationsFromECS – Mandatory Reads the classifications to synchronize from ECS
	ResolveClassificationsChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveClassificationsToECS – Optional Saves synchronized classifications to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveClassificationsToSC – Mandatory Saves synchronized classifications to SC.

SynchronizeTypes

Name:	SynchronizeTypes
Description:	This pipeline is responsible for synchronizing all Types in the separate Product Types repository
Usage:	Called from pipeline SynchronizeArtifacts as initialization of separate repositories before synchronizing products and their references to these repositories.
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization
Processors:	
	ReadTypesFromSC – Optional Reads the types to synchronize from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadTypesFromECS – Mandatory Reads the types to synchronize from ECS
	ResolveTypesChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveTypesToECS – Optional Saves synchronized types to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveTypesToSC – Mandatory

	Saves synchronized types to SC.

SynchronizeDivisions

Name:	SynchronizeDivisions
Description:	This pipeline is responsible for synchronizing all divisions in the separate Divisions repository
Usage:	Called from pipeline SynchronizeArtifacts as initialization of separate repositories before synchronizing products and their references to these repositories.
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization
Processors:	
	ReadDivisionsFromSC – Optional Reads the divisions to synchronize from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadDivisionsFromECS – Mandatory Reads the divisions to synchronize from ECS
	ResolveDivisionsChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveDivisionsToECS – Optional Saves synchronized divisions to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveDivisionsToSC – Mandatory Saves synchronized divisions to SC.

SynchronizeResources

Name:	SynchronizeResources
Description:	This pipeline is responsible for synchronizing all resources in Sitecore Media Library
Usage:	Called from pipeline SynchronizeArtifacts as initialization of separate repositories before synchronizing products and their references to these repositories. NB In case resources are kept only in the ECS, then this pipeline can be skipped or configured empty
Args:	
	Request – Is empty by default. Is set prior to calling the pipeline.
	Response - Contains the list of messages generated during synchronization
Processors:	
	ReadResourcesFromSC – Optional Reads the resources to synchronize from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadResourcesFromECS – Mandatory Reads the resources to synchronize from ECS
	ResolveResourcesChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.

	SaveResourcesToECS – Optional
	Saves synchronized resources to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveResourcesToSC – Mandatory
	Saves synchronized resources to SC.

SynchronizeSpecifications

Name:	SynchronizeSpecifications
Description:	This pipeline is responsible for synchronizing specifications on type, category and globally by running separate pipelines for each
Usage:	Called from pipeline SynchronizeArtifacts as initialization of separate repositories before synchronizing products and their references to these repositories
Args:	
	Request - Is set prior to calling the pipeline.
	Response - Is read after the pipeline is called.
Processors:	
	RunSynchronizeGlobalSpecifications – Runs a separate pipeline to synchronize global specifications
	RunSynchronizeTypeSpecifications - Runs a separate pipeline to synchronize type specifications
	RunSynchronizeClassificationSpecifications - Runs a separate pipeline to synchronize category specifications

SynchronizeGlobalSpecifications

Name:	SynchronizeGlobalSpecifications
Description:	This pipeline is responsible for synchronizing global specifications The specifications and the tables for fixed key-value pairs are stored under "/sitecore/content/Product Artifacts/Lookups/Global Product Specification Lookups"
Usage:	Called internally from pipeline SynchronizeSpecifications
Args:	
	Request - Is set prior to calling the pipeline.
	Response - Is read after the pipeline is called.
Processors:	
	ReadGlobalSpecificationsFromSC – Optional Reads the product specifications data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadGlobalSpecificationsFromECS – Mandatory Reads the product specifications data from ECS
	ResolveGlobalSpecificationsChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveGlobalSpecificationsToECS – Optional Saves synchronized product specifications data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveGlobalSpecificationsToSC – Mandatory Saves synchronized product specifications data to SC.

SynchronizeTypeSpecifications

Name:	SynchronizeTypeSpecifications
Description:	This pipeline is responsible for synchronizing type specifications The specifications and the tables for fixed key-value pairs are stored under "/sitecore/content/Product Artifacts/Product Types" Note: For types also specification options and default values are synchronized as part of this pipeline
Usage:	Called internally from pipeline SynchronizeSpecifications
Args:	
	Request - Is set prior to calling the pipeline.
	Response - Is read after the pipeline is called.
Processors:	
	ReadTypeSpecificationsFromSC – Optional Reads the product specifications data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadTypeSpecificationsFromECS – Mandatory Reads the product specifications data from ECS
	ResolveTypeSpecificationsChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveTypeSpecificationsToECS – Optional Saves synchronized product specifications data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveTypeSpecificationsToSC – Mandatory Saves synchronized product specifications data to SC.

SynchronizeClassificationSpecifications

Name:	SynchronizeClassificationSpecifications
Description:	This pipeline is responsible for synchronizing Category specifications The specifications and the tables for fixed key-value pairs are stored under "/sitecore/content/Product Artifacts/Classifications" Note: Since multiple different classification schemes are supported, this pipeline is responsible for synchronizing specifications for all schemes
Usage:	Called internally from pipeline SynchronizeSpecifications
Args:	
	Request - Is set prior to calling the pipeline.
	Response - Is read after the pipeline is called.
Processors:	
	ReadClassificationSpecificationsFromSC – Optional Reads the classifications specifications data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadClassificationSpecificationsFromECS – Mandatory Reads the classifications specifications data from ECS
	ResolveClassificationSpecificationsChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveClassificationSpecificationsToECS – Optional

Saves synchronized classifications specifications data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
SaveClassificationSpecificationsToSC – Mandatory Saves synchronized classifications specifications data to SC.

After running the pipeline the categories will have a folder called Specifications containing all the specifications for the category, including tables with fixed set key-value pairs for reference from products.

SynchronizeProduct

SynchronizeProduct is used to synchronize a single product between the external commerce system and Sitecore. The product to synchronize is given by Id.

Name:	SynchronizeProduct
Description:	This pipeline is responsible for synchronizing a single product by calling a number of individual pipelines. Each pipeline will update the references between the product and the related separate repositories except pipeline SynchronizeProductItem, which operates on the main product data on the product item itself.
Usage:	Called directly from service method SynchronizeProduct and from SynchronizeProductList indirectly
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the SystemMessages. Is read after the pipeline is called.
Processors:	
	RunSyncronizeProductManufacturers – Synchronizes relations to manufacturers
	RunSyncronizeProductType – Synchronizes relation to product type
	RunSyncronizeProductClassifications – Synchronizes relations to classifications
	RunSyncronizeProductResources – Synchronizes resources and relations to resources
	RunSyncronizeProductRelations – Synchronizes relations to other products thorugh cross-sell, variants etc
	RunSyncronizeProductDivisions – Synchronizes relations to divisions
	RunSyncronizeProductItem – Synchronizes main product data on the product item itself
	RunSyncronizeProductSpecifications – Calls individual pipeline to synchronize product specifications

SynchronizeProductManufacturers

SynchronizeProductManufacturers is used to synchronize references between a single product and separate repository Manufacturers between the external commerce system and Sitecore. The product to synchronize is given by external product Id.

Name:	SynchronizeProductManufacturers
Description:	This pipeline is responsible for synchronizing and updating the relation between a given product and manufacturers. It's assumed that manufacturers are already synchronized and present in CMS The references to manufacturers are stored directly on the main product item
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the Manufacturer. Is read after the pipeline is called.
Processors:	
	ReadProductManafacturersFromSC – Optional Reads the product manafacturers reference data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductManafacturersFromECS – Mandatory Reads the product manafacturers reference data from ECS
	ResolveProductManafacturersChanges – Optional

Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
SaveProductManafacturersToECS – Optional
Saves synchronized product manafacturers reference data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
SaveProductManafacturersToSC – Mandatory
Saves synchronized product manafacturers reference data to SC.

SynchronizeProductType

Name:	**SynchronizeProductType**
Description:	This pipeline is responsible for synchronizing and updating the references between a given product and its product type. It's assumed that types are already synchronized and present in CMS The references to product type is stored directly on the main product item
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the Manufacturer. Is read after the pipeline is called.
Processors:	
	ReadProductTypeFromSC – Optional
	Reads the product type reference data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductTypeFromECS – Mandatory
	Reads the product type reference data from ECS
	ResolveProductTypeChanges – Optional
	Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductTypeToECS – Optional
	Saves synchronized product type reference data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductTypeToSC – Mandatory
	Saves synchronized product type reference data to SC.

SynchronizeProductClassifications

Name:	**SynchronizeProductClassifications**
Description:	This pipeline is responsible for synchronizing and updating the references between a given product and associated classifications and categories within. It's assumed that classifications are already synchronized and present in CMS The references to categories are stored directly on the main product item
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the Manufacturer. Is read after the pipeline is called.
Processors:	
	ReadProductClassificationsFromSC – Optional
	Reads the product classification reference data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductClassificationsFromECS – Mandatory
	Reads the product classification reference data from ECS
	ResolveProductClassificationsChanges – Optional

Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.	
SaveProductClassificationsToECS – Optional Saves synchronized product classification reference data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.	
SaveProductClassificationsToSC – Mandatory Saves synchronized product classification reference data to SC.	

SynchronizeProductEntity

Name:	SynchronizeProductEntity
Description:	This pipeline is responsible for synchronizing and updating the main product entity (data(for the product with the given external product ID.
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the Manufacturer. Is read after the pipeline is called.
Processors:	
	ReadProductFromSC – Optional Reads the product data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductFromECS – Mandatory Reads the product data from ECS
	ResolveProductChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductToECS – Optional Saves synchronized product data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductToSC – Mandatory Saves synchronized product data to SC.

SynchronizeProductDivisions

Name:	SynchronizeProductDivisions
Description:	This pipeline is responsible for synchronizing and updating the references between a given product and associated divisions. It's assumed that divisions are already synchronized and present in CMS The references to divisions are stored directly on the main product item
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the Manufacturer. Is read after the pipeline is called.
Processors:	
	ReadProductDivisionsFromSC – Optional Reads the product divisions reference data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductDivisionsFromECS – Mandatory Reads the product divisions reference data from ECS
	ResolveProductDivisionsChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.

SaveProductDivisionsToECS – Optional Saves synchronized product divisions reference data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.	
SaveProductDivisionsToSC – Mandatory Saves synchronized product divisions reference data to SC.	

SynchronizeProductResources

Name:	**SynchronizeProductDivisions**
Description:	This pipeline is responsible for synchronizing and updating the references between a given product and associated resources. It's assumed that resources are already synchronized and present in CMS or that the references are external using an URI to point to the location The references to resources are stored under the main product item on the path "[Product Item]/Resources/[Resource]"
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id. Is set prior to calling the pipeline.
	Response - Contains the Manufacturer. Is read after the pipeline is called.
Processors:	
	ReadProductResourcesFromSC – Optional Reads the product resources reference data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductResourcesFromECS – Mandatory Reads the product resources reference data from ECS
	ResolveProductResourcesChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductResourcesToECS – Optional Saves synchronized product resources reference data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductResourcesToSC – Mandatory Saves synchronized product resources reference data to SC.

SynchronizeProductRelations

Name:	**SynchronizeProductRelations**
Description:	This pipeline is responsible for synchronizing product relations for a single product
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id . Is set prior to calling the pipeline.
	Response - Contains the Product relations. Is read after the pipeline is called.
Processors:	
	ReadProductRelationsFromSC - Reads the product relations data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductRelationsFromECS – Reads the product relations data from ECS
	ResolveProductRelationsChanges – Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductRelationsToECS – Saves synchronized product relations data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.

SaveProductRelationsToSC - Saves synchronized product relations data to SC.	

SynchronizeProductSpecifications

Name:	**SynchronizeProductSpecifications**
Description:	This pipeline is responsible for synchronizing product specifications for a single product. It's assumed that specification tables of fixed key-value pairs (lookups) are already synchronized and present in CMS when this pipeline is run The references to specifications are stored under the main product item on the path "[Product Item]/Specifications/[Specification]" Specifications that reference lookup tables can point to specification tables located under global, classification or type.
Usage:	Called internally from pipeline SynchronizeProduct
Args:	
	Request - Contains the external product Id . Is set prior to calling the pipeline.
	Response - Contains the Product relations. Is read after the pipeline is called.
Processors:	
	ReadProductSpecificationsFromSC - Optional Reads the product specification data from SC. This processor can be skipped if changes only are pushed from ECS to SC.
	ReadProductSpecificationsFromECS – Mandatory Reads the product specification data from ECS
	ResolveProductSpecificationChanges – Optional Resolves differences between ECS and SC. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductSpecificationsToECS – Optional Saves synchronized product specification data to ECS. This processor can be skipped if changes only are pushed from ECS to SC.
	SaveProductSpecificationsToSC – Mandatory Saves synchronized product specification data to SC.

2.7 Connect Configuration

You can use the `Sitecore.Commerce.config` file and the individual service layer configuration files to register entities, repositories, pipeline processors and service providers. In the following sections the Cart service layer is used as example.

2.7.1 Factories and entities

The Factory Method Pattern is an object-oriented creational design pattern that implements the concept of factories. You can create objects without basing it on a specific class. The core of this pattern is to define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.

To configure the entity factory to use, set the type. By default the Sitecore Factory is used implicitly through Connect:

```xml
<!-- 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" />
```

To configure custom objects in the `Sitecore.Commerce.Carts.config` file:

```xml
<!-- Connect ENTITIES Contains all the Connect 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" />
</commerce.Entities>
```

In an actual Connect provider implementation, the custom objects are known and do not necessarily have to be created through use of Factory.

You can use references to the factory as a parameter in some processors, for example, `CreateCart`:

```xml
<processor type="Sitecore.Commerce.Pipelines.Carts.CreateCart.CreateCart,
Sitecore.Commerce">
        <param ref="entityFactory" />
</processor>
```

You must use the `Factory.Create` method to get an instance of the needed type. For example, in the following code snippet, we need a cart and calls the factory to create and return a cart. The cart domain model can be completely modified and customized so that you can replace the default cart type with your own implementation:

```csharp
public override void Process(ServicePipelineArgs args)
{
  var result = (CartResult)args.Result;
  var cart = this.entityFactory.Create<Cart>("Cart");
  var request = (CreateOrResumeCartRequest)args.Request;
  cart.UserId = request.UserId;
  cart.ShopName = request.ShopName;
  cart.CartName = request.CartName;
  cart.CustomerId = request.CustomerId;
  cart.CartStatus = CartStatus.InProcess;
  result.Cart = cart;
}
```

2.7.2 Pipelines for Methods

In the `Sitecore.Commerce.Carts.config` file, you can set your processors into pipelines that to inject business logic.

The following table contains some examples of pipelines in the configuration file of Connect:

Pipeline	Description
getCarts	This pipeline searches for all carts that match some specific criteria. The carts are managed by the commerce system. This pipeline reads cart data from the commerce system and converts that data into Connect format.
createOrResumeCart	This pipeline: • Initiates the creation of a shopping cart. • Loads persisted, abandoned cart, if present. • Calls `resumeCart` pipeline to resume loaded cart. • Calls `createCart` pipeline to create cart if no cart was found in the previous steps.
createCart	This pipeline: • Is internally used by the `createOrResumeCart` pipeline if the existing cart was not found and should be created. • Creates a cart with the minimal number of required fields. • Moves a visitor / contact to the initial state in the engagement plan. • Saves a cart to the storage and triggers the `CartCreate` event.
resumeCart	This pipeline: • Is internally used by the `createOrResumeCart` pipeline if a cart was loaded and should be resumed. • Sets the initial state to the loaded cart, moves a visitor / contact to the initial state in the engagement plan. • Saves a cart to the storage and triggers the `CartResume` event.
loadCart	This pipeline: • Gets a cart object that matches a specified criteria. • Reads data for a cart that is managed by the commerce system. • Reads the cart data from the commerce system and converts that data into the Connect format.
saveCart	This pipeline saves the cart object to an external system and in Sitecore EA state.
addCartLines	This pipeline adds a new line to the shopping cart and records a corresponding page event in DMS. This happens when a product is added to the cart.
removeCartLines	This pipeline removes cart lines from cart.
updateCartLines	This pipeline updates lines on cart.
deleteCart	This pipeline: • Deletes a cart permanently: • When the cart is deleted, it triggers the event in DMS to indicate that the cart is deleted.
updateCart	This pipeline: • Passes an updated cart to the external commerce system. • Triggers an event in DMS to indicate that the cart is being updated.
lockCart	This pipeline sets the cart to a locked state and prevents any modifications.
unlockCart	This pipeline sets the cart to an unlocked state.
getCartTotal	This pipeline: • Gets the totals object that matches the specified criteria. • Is responsible for reading pricing data from a commerce system. • Converts the contents of a Connect cart into a format the commerce system can work with.

	• Sends a request to the commerce system to calculate the totals, and converts the output into the proper Connect format.
getProductPrices	This pipeline: • Gets the price object that matches the specified criteria. • Reads the pricing data from a commerce system. • Requests the product pricing information from the commerce system and converts the output into the proper Connect format.