

Rev: January 29, 2019

Sitecore Commerce Developer's Guide

Sitecore Commerce 8.2.1

A guide for extending the Sitecore Commerce solution



sitecore[®]
Own the experience[™]

Table of Contents

Chapter 1	Getting Started with Sitecore Commerce.....	5
1.1	Guided Tours.....	5
Chapter 2	Sitecore Commerce Engine.....	6
2.1	Sitecore Commerce Engine Concepts.....	6
2.2	Getting the Customer.Sample.Solution Up and Running.....	6
2.3	Functional Testing of the Solution.....	6
2.4	Adding a Plugin.....	8
2.5	Deploying Your Solution.....	8
2.6	Commerce Engine Roles.....	8
Chapter 3	Sitecore Commerce Core.....	10
3.1	Commerce Core Concepts.....	10
3.2	Sitecore Deployment Environments.....	12
3.3	Commerce Entity.....	13
3.4	EntityStore.....	13
3.5	Compositional Extensibility.....	14
3.6	Commerce List.....	14
3.7	Entity Journaling.....	15
3.8	Sitecore Commerce Service API.....	16
3.9	Localization.....	17
Chapter 4	Commerce Views Service.....	19
4.1	Authoring API.....	19
4.2	EntityViews.....	19
4.3	Composite EntityViews.....	19
4.4	EntityView Properties.....	20
4.5	EntityView Actions, Commands, Pipelines.....	21
4.6	EntityView Samples.....	21
4.7	EntityActions.....	25
Chapter 5	Rules.....	26
5.1	Rules Commands and Pipelines.....	26
5.2	Rules Models.....	26
Chapter 6	Orders Service.....	27
6.1	Orders Concepts.....	27
6.2	Orders Views.....	28
6.3	Orders Actions, Commands and Pipelines.....	29
6.4	Orders Models.....	30
6.5	Orders Policies.....	30
Chapter 7	Orders Service – Shopping Cart.....	31
7.1	Cart Actions, Commands and Pipelines.....	31
7.2	Cart Models.....	32
7.3	Cart Policies.....	32
Chapter 8	Orders Service – Returns.....	33
8.1	Returns Views.....	33
8.2	Returns Actions, Commands and Pipelines.....	33
8.3	Returns Models.....	34
8.4	Returns Policies.....	34
Chapter 9	Pricing Service.....	35
9.1	Pricing Concepts.....	35
9.2	Pricing Views.....	36
9.3	Pricing Actions, Commands, Pipelines.....	37
9.4	Pricing Models.....	39
9.5	Pricing Policies.....	39
9.6	Pricing Transparency.....	41

Chapter 10	Promotions Service.....	46
10.1	Promotions Concepts.....	46
10.2	Promotions – Qualifications.....	47
10.3	Promotions – Benefits.....	48
10.4	Promotions Samples.....	48
10.5	Calculating Promotions.....	49
10.6	Promotions Views.....	51
10.7	Promotions Actions, Commands, Pipelines.....	51
10.8	Promotions Models.....	54
10.9	Promotions Policies.....	54
Chapter 11	Promotion Service – Coupons.....	55
11.1	Coupons Concepts.....	55
11.2	Coupons Views.....	56
11.3	Coupons Actions, Commands, Pipelines.....	56
11.4	Coupons Models.....	57
11.5	Coupons Policies.....	57
Chapter 12	Entitlements Service.....	59
12.1	Entitlement Concepts.....	59
12.2	Entitlement Views.....	59
12.3	Entitlements Actions, Commands, Pipelines.....	60
12.4	Entitlements Policies.....	60
Chapter 13	Customer Service.....	61
13.1	Customer Views.....	61
13.2	Customer Actions, Commands, Pipelines.....	62
13.3	Customer Models.....	62
13.4	Customer Policies.....	62
Chapter 14	Catalog Service.....	65
14.1	Catalog Actions, Commands, Pipelines.....	65
14.2	Catalog Models.....	66
Chapter 15	Availability Service.....	67
15.1	Availability Commands and Pipelines.....	67
15.2	Availability Policies.....	67
Chapter 16	Inventory Service.....	69
16.1	Inventory Commands and Pipelines.....	69
16.2	Inventory Policies.....	69
Chapter 17	Payment Service.....	70
17.1	Payment Concepts.....	70
17.2	Payment Views.....	71
17.3	Payment Actions, Commands, Pipelines.....	71
17.4	Payment Policies.....	72
Chapter 18	Fulfillment Service.....	73
18.1	Fulfillment Concepts.....	73
18.2	Fulfillment Views.....	76
18.3	Fulfillment Actions, Commands, Pipelines.....	76
18.4	Fulfillment Models.....	78
18.5	Fulfillment Policies.....	78
Chapter 19	Shops Service.....	79
19.1	Shops Concepts.....	79
19.2	Accessing a Shop from Rules or from Pipeline Blocks.....	80
19.3	Shops Components.....	80
19.4	Shops Models.....	80
Chapter 20	Guided Tours.....	81
20.1	Get the Customer.Sample.Solution Up and Running.....	81
20.2	Creating Your First Plugin.....	83
20.3	Mapping Additional Properties from a Commerce Server Catalog.....	86
20.4	Extending a Cart Line.....	91
20.5	Extending a Commerce View to Show Additional Information in the Business Tools.....	93

Sitecore® is a registered trademark. All other brand and product names are the property of their respective holders. The contents of this document are the property of Sitecore. Copyright © 2001-2019 Sitecore. All rights reserved.

Chapter 1 Getting Started with Sitecore Commerce

Sitecore Commerce provides multiple layers of extensibility, depending on the needs of the solution. These layers are described in the following sections.

Sitecore Commerce Engine

The Sitecore Commerce Engine is an extensible commerce core framework, hosting commerce services such as: Cart, Order, and Pricing and Promotion. It includes a pluggable framework for extending the engine to modify or add to existing functionality.

In general, this is the layer that should be extended when modifying existing functionality or designing new commerce functionality. This layer provides the greatest ability to extend the product, while retaining upgradeability. This layer is also structured to ensure individual extensions can be logically separated from each other.

This guide focusses on the developer experience of extending the Sitecore Commerce Engine using its plugin technology.

Sitecore Commerce Business Tools

The Sitecore Commerce Business Tools are a set of rich business tools for merchandisers and customer service representatives. The business tools are built on the [Sitecore Process Enablement & Accelerator Kit \(SPEAK\)](#) framework for developing Sitecore applications.

The business tools can be extended using the pluggable framework, where the same separation of concerns approach as the Sitecore Commerce Engine layer applies, and upgradeability can be retained. The delivered business tools are not modified directly; instead they are extended using plugins. This approach allows the delivered business tools to be updated to newer technologies in future releases without affecting or requiring refactoring of any existing plugin extensions.

When extending the business tools, the Commerce Views service can be leveraged; this provides a set of data-driven services that feed data to the business tools in a structured way. Developers can choose to modify, add, or remove functionality from the existing business tools. Additionally, plugins provided by Sitecore's partners or those downloaded from a NuGet feed can automatically wire-in and extend the business tools. This allows a rich extensibility framework comprising multiple sources, while minimizing the risk of limiting future upgrades.

Sitecore Commerce Reference Storefront

The Sitecore Commerce Reference Storefront is a sample storefront website that is integrated with the Sitecore Commerce Engine. It can be used as a starting point to building a customized storefront. The Reference Storefront source code and supporting documentation can be downloaded from [GitHub](#).

Sitecore Commerce Connect

Sitecore Commerce Connect is an integration layer between the Sitecore Experience Platform and Commerce specific functionality, and between the Reference Storefront and the Commerce back-end functionality. Documentation for extending the Sitecore Commerce Connect layer is located on [doc.sitecore.net](#).

1.1 Guided Tours

To help illustrate extensibility concepts, a series of guided tours are provided at the end of this document. These represent an end-to-end sample extension for adding Customer Loyalty functionality. They illustrate various extensibility patterns. These Guided Tours are in the back of this document and different sections of the document may reference specific sections of the Guided Tour to illustrate extensibility in their area.

Chapter 2 Sitecore Commerce Engine

2.1 Sitecore Commerce Engine Concepts

The Sitecore Commerce Engine is a thin ASP.NET Core application that serves as a host for a group of microservices that enable commerce functionality. These services are loosely coupled with each other. Together they provide a robust, extensible set of commerce enabling services. Services exposed from the engine are exposed as OData/REST based APIs. This enables either direct REST calls to the services, or calls through a generated smart proxy, which examines OData metadata and generates a strongly typed experience for accessing the service.

The initial out-of-the-box Sitecore Commerce Engine is provided as part of the Sitecore Commerce SDK. It enables a developer to extend their solution by choosing which plugins are enabled in their solution. Plugins can be retrieved from:

- The set of plugins provided with the product.
- Plugins provided by Sitecore partners.
- Public plugins downloaded from a NuGet feed.

The Sitecore Commerce Engine provided in the SDK is called the *Customer.Sample.Solution*, where customer in this case means a Sitecore customer (also referred to in this guide as a *solution developer*). You can open the Sitecore Commerce solution in Visual Studio 2015. It contains the Sitecore Commerce Engine project itself and some sample plugins that demonstrate a typical solution developer's extension of the solution.

A typical pattern for extending the solution is:

1. Open the *Customer.Sample.Solution*.
2. Rename the solution to something more appropriate to the solution developer.
3. F5 to refresh the solution to ensure that the initial solution runs.
4. Exercise the solution through the postman samples or using the sample Console app.
5. Extend the solution by creating additional plugins as separate projects within the solution using a strong separation of concerns.
6. Test your plugin using a combination of unit tests and functional tests
7. When the solution meets requirements, create a new deployment for the solution
8. Send the new deployment through any QA or DevOps process to get the deployment deployed to production.

2.2 Getting the Customer.Sample.Solution Up and Running

The basic steps for getting the solution up and running are documented in the guided tour in this document: [Get the Customer.Sample.Solution Up and Running](#).

2.3 Functional Testing of the Solution

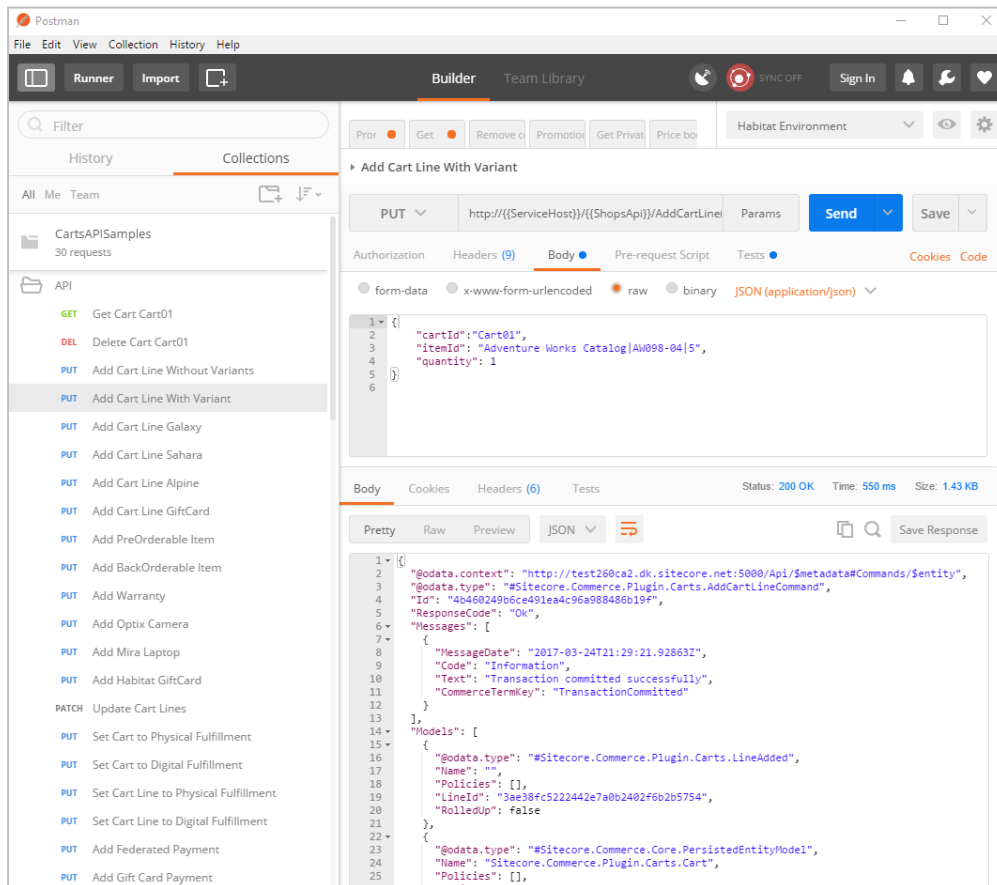
Sitecore Commerce improves the development lifecycle by providing examples of various pieces of functionality, as exercised through the exposed service APIs. These are intended to represent functional samples. They exercise a running solution but are not intended to represent a full set of tests that an actual quality assurance test would implement, only a representative sample of functionality to illustrate patterns and provide a simple mechanism for validating functionality for the developer.

The following sections describe the tools that are available for basic functional testing.

Developer's Guide

Postman Samples

Postman samples are a set of sample calls to the Sitecore Commerce Engine services using the [Postman REST Client](#) to illustrate REST based calls to execute functionality. These are simple examples of calling the engine and demonstrating the call pattern and its expected result. Postman is a simple, free tool for managing HTTP level calls and responses for creating a simple test surface. The developer can use this tool to perform simple tests of existing functionality and to review patterns for use in creating their own tests. Postman is not required to develop a solution using Sitecore Commerce. Any HTTP testing tool will work; however, use Postman for the delivered working samples, which you can then transfer to your HTTP testing tool of choice.



Sample Console Application

Sitecore Commerce Services are exposed as OData services. An OData service provides typical REST-based functionality, allowing the service to respond to traditional REST Queries. It also provides rich metadata for the generation of a strongly typed proxy, which gives a client the ability to develop connectivity with the service using a strongly typed client proxy. This helps improve maintainability by allowing strongly typed client development, which quickly identifies breaking changes.

To demonstrate this programming style, the SDK provides a fully functional set of examples in the *Sitecore.Commerce.Sample.Console* solution. This tool can be opened in Visual Studio 2015 and is intended to run end-to-end to exercise the complete solution. This includes exercising Pricing, Promotion, and Order solutions with samples of typical order scenarios.

When extending the solution, it is advised that you also tie into these console samples. Modify a sample to remove any functionality that you are not interested in. Retrofit the existing samples to your own scenarios. This allows you to generate orders, for example, supporting multiple scenarios from day one. You can use the sample data provided and rapidly iterate to modify the solution to demonstrate your own scenarios.

Unit Testing the Solution

Sitecore Commerce plugins support the ability to be unit tested. This provides the ability to have a high quality plugin that has unit tests exercising the extensions within the plugin.

The Sitecore product department uses the [NUnit](#) testing framework internally. NUnit is an open source unit testing framework for Microsoft .NET. It is lightweight, integrates with Visual Studio, and supports Dependency Injection.

A Sitecore customer or partner solution developer can choose their own unit testing framework, or they may want to consider NUnit.

2.4 Adding a Plugin

The Sitecore Commerce Engine offers a pluggable extensibility model that enables extensive customization of engine functionality. It uses a model that promotes upgradability and a clear separation of concerns between individual extensions. Plugins can be obtained via a NuGet service, or could be provided by a Sitecore partner as part of custom development.

All Sitecore Commerce plugins are published using a Sitecore private NuGet feed. This enables publishing new versions of the Sitecore Commerce plugins without the overhead of a major release. A walkthrough of the basic steps is described in this document, in the guided tour: [Creating Your First Plugin](#)

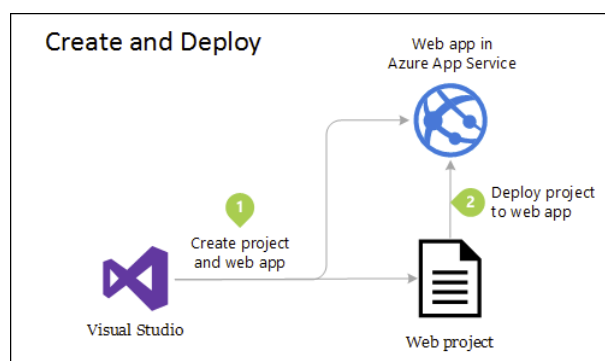
2.5 Deploying Your Solution

The *Customer.Sample.Solution* is the core Visual Studio solution in which all extensions can be developed, in the form of plugins. You can run this solution directly in Visual Studio for testing, which allows a quick develop/test/develop/test cycle.

When the testing is complete and you are ready to make the extensions available to QA and eventually into a production environment, you must create a deployment. The Sitecore Commerce Engine leverages the *dotnet* framework. You can publish a solution from the dotnet framework in one of the following ways:

- Directly in Visual Studio using the *Publish* option.
- Through a Command line, using the *dotnet publish* command.
- Inside a Build environment, as part of a Continuous Integration environment.

Publishing can directly update a running deployment, such as publishing directly to IIS from Visual Studio or publishing directly to an Azure Web App. Or, it can create a Web Deploy package that can then be taken through a QA process and deployed by dedicated DevOps personas.



2.6 Commerce Engine Roles

In a development environment, there is usually only one instance of the Commerce Engine running. It services all traffic, both from the storefront layer and from the business tools layer.

In a production environment, this traffic is usually split up among multiple installed instances of the Commerce Engine, which are usually physically located close to their traffic sources. These instances are referred to as *Engine Roles*. This distinction is purely logical; there is no real difference in the deployed bits between different installed roles. These roles are defined by where the traffic is originating from.

The following engine roles are considered during solution implementation. You must apply these in any production environment, even though implementation details can vary.

Authoring Role

The Authoring role is the instance of the Commerce Engine that serves traffic from the business tools, including the Merchandising Manager and the Customer & Order Manager. It is generally installed close to where the business activity takes place. This role serves light traffic so that scale requirements are normally relatively low. This is because these solutions generally have relatively few business users, compared to the number of shoppers.

Shops Role

The Shops role is the instance of the Commerce Engine that serves traffic from one or more storefronts. This role is intended to scale to support demand. It is usually installed in close proximity to the Sitecore Experience Platform instances that are generating the traffic. To scale the solution, the Sitecore XP instances and/or the Commerce Engine instances can be scaled depending on traffic mix and where bottlenecks are identified.

Minions Role

The Minions role is an instance of the Commerce Engine that runs independently and supports asynchronous processing. This includes any post-order capture processing as well as any cleanup or pruning tasks that might be desired.

DevOps Role

The DevOps role is an instance of the Commerce Engine that is internal and only available to DevOps personas. This role can have an identity with higher privileges allowing DevOps members to perform maintenance tasks that must not be allowed in other roles, for example, bootstrapping and environment initialization functions.

Each deployed role can have different policies and behaviors, which can be specified using a specific *Commerce Environment* for that role. When a call is made to the Commerce Engine, an environment is specified in a *Header*. This environment is used by the engine to control policies and behavior for that call. This allows explicit independent control per role of functions, such as caching, data storage, integration connections, and so on.

Using environment policies, engine roles can either share a common persistent store, or each use a separate dedicated storage. All roles share the same storage in the out-of-box implementation. This allows artifacts to be visible to each role without a publishing process. For example, this makes orders immediately available to the Authoring role, and makes approved promotions and pricing changes immediately available to the Shops role.

Each role can have independent caching policies. For example, reduced caching can be configured on the Authoring role so that changes are seen right away, and heavier caching can be configured on the Shops role to increase performance.

Chapter 3 Sitecore Commerce Core

Sitecore Commerce Core is a lightweight framework that provides core application capabilities in an encapsulated package. The Core does not itself offer any commerce capabilities; commerce-related functionality is enabled through plugins that take a dependence on the Core functionality.

The following diagram depicts the lower logical layers of the Commerce Engine. The Content Foundation is a Sitecore XP construct that provides the basic infrastructure. The Commerce Core provides the structure and capabilities to implement the functional plugins.



3.1 Commerce Core Concepts

The following table describes the concepts of Sitecore Commerce Core:

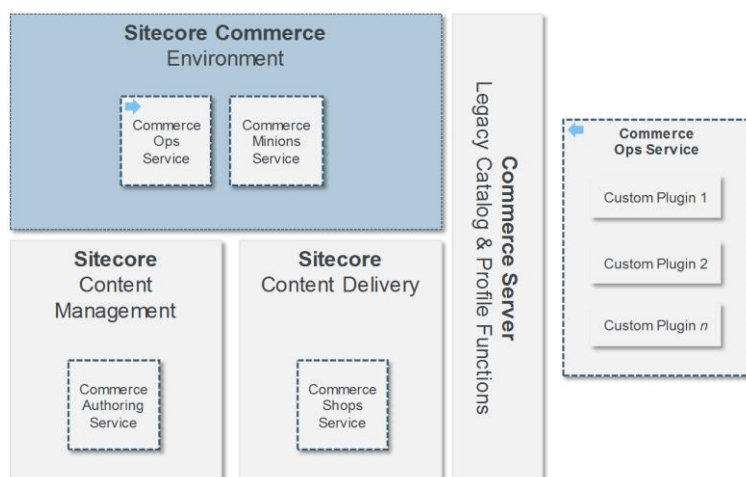
Concept	Description
Activity	An internal construct that wraps a specific set of work for activity tracking, for example, for reporting and performance. The solution implements this as a <code>CommandActivity</code> ; all commands are wrapped to enable performance reporting on the command. A <code>TrackActivityPipeline</code> is executed when the activity is complete, which enables a plugin to take action. An example of an action is to monitor performance of an action against an SLA and report actions outside the SLA, or to generate an alert.
Approval	Basic approval commands and pipelines to facilitate a basic approval process. Approvals are used by pricing and promotions to seek the approval of changes before implementing.
Authentication	The ability to authenticate a call to the service API using certificates.
Bootstrap	Commands and pipelines to support bootstrapping the solution. The <code>BootstrapPipeline</code> loads environment configurations from the <code>wwwroot/data/environments</code> folder into the <code>SitecoreCommerce_Global</code> database. After bootstrapping, those environment configurations do not need to be present. Any subsequent environment configuration is retrieved from the database during normal runtime operations.
Caching	Commands and pipelines to support in-memory caching. This provides the ability to have environment-specific caching, to specify cache priorities, and to clear the cache. Caching functionality, in turn, leverages the <code>Sitecore.Framework.Caching</code> functionality for the actual caching.
Command	Basic structure for supporting the concept of commands. A command acts like the API in a task-driven API.

Concept	Description
Component	Basic structure for supporting compositional extensibility, including the component class and various base components.
Context	A call-level context called <code>CommerceContext</code> , which is initialized when a call enters the service and is carried throughout the service call. The <code>CommerceContext</code> provides an object cache, messaging, headers, and other core call-level concepts.
Controller	Basic controllers that make core functionality available through the Service API.
Converter	Custom JSON converters for the Service API.
Entity	Commands, policies, and pipelines to support reading and writing commerce entities. A <code>CommerceEntity</code> is a core artifact designed to directly represent a business concept, which is stored as a single unit in a persistent storage. Entities have identifiers and can be retrieved through the Service API.
Environment	Sitecore Commerce environments provide the ability to have separately configurable pools of data and service functionality that runs together in a single-service instance. Environments can share the same persistent store as other environments or be separated into their own exclusive persistent store.
Event	Basic infrastructure to support events and event-driven actions.
Exception	Basic <code>CommerceException</code> base class.
Globalization	Commands and their pipelines to support globalization, including support for multicurrency and localization.
List	Commands and pipelines to support basic list functionality, including basic list management. Use <code>ManagedLists</code> to track lists of entities either based on their state or based on activities that need to be performed on them. Lists are used to provide organizational structure and to support business processes.
Location	Commands and pipelines to support locations, for example, retrieving supported countries and country information.
Logging	Support for core logging using Serilog and to specify logging using Microsoft Application Insights.
Media	Core classes to support media types and policies, including a <code>GlobalImagePolicy</code> and an image class.
Minion	Commands and pipelines to support minions including the <code>MinionBoss</code> and <code>RunMinion</code> pipelines, and the policies to support configuring minions in the environment configuration.
Model	Basic core models, which are POCO classes that are reusable inside entities and components. Models can be used to present data as part of a command response, in the models collection. Models are listed in the Sitecore.Commerce.Documentation.chm on the Model Class page.
Node	Core pipelines, blocks, and policies that enable basic node functionality. A node is a running instance of the Service API.
Performance	Commands and policies to support tracking and integrating with performance counters for commands.
Pipeline	Core commands and models to support pipeline functionality. Pipelines, in turn, leverage the <code>Sitecore.Framework.Pipelines</code> infrastructure.
Plugin	Core support for the Sitecore Commerce pluggable extensibility.
Policy	A named, versionable and variable set of data that can be used as facts within behaviors to influence behavioral outcomes. This provides an auditable mechanism for viewing, simulating, and changing core decision criteria that might be used by business processes or other policy-driven behavior. Various plugins have specialized policies to add additional facts to existing policies and/or new policies, or to implement new concepts, such as dynamic pricing. Policy characteristics include: <ul style="list-style-type: none"> • Centralized policy store using abstract entity storage. • Worker processes only need a link to the policy store to bootstrap. • Single point of truth for policies. • Publish workflow without moving data. • Policies are heavily cached and rarely change.

Concept	Description
	<ul style="list-style-type: none"> Policies can have attached rules to deliver personalized policies. Policies are listed in the Sitecore.Commerce.Documentation.chm on the Policy Class page.
Provider	Core interfaces for an <code>EntityProvider</code> and an <code>IndexProvider</code> . The concept of a provider is avoided because any plugin could potentially be a provider. This is only used by the <code>FileSystemProvider</code> , which enables reading a <code>CommerceEntity</code> from the file system.
Search	Provides a <code>SearchOption</code> class that enables specifying search parameters when traversing a list. Currently direct searching is not supported; this can only be used to specify skip/take properties to enable batch traversing a list. In the future, this may be extended to enable more powerful search capabilities.
ServiceApi	Core models and policies to enable the basic Service API.
Transaction	Core functionality to support transactionality in the solution.

3.2 Sitecore Deployment Environments

As defined previously, Sitecore Commerce environments provide the ability to have separately configurable pools of data and service functionality that runs together in a single-service instance. An environment has another application at the overall solution level. This is illustrated in the following figure:



The diagram shows four Sitecore deployment environments in a production deployment, including where the Commerce Engine roles are deployed:

- Sitecore Content Management – internal environment for authoring content before publishing for public consumption. Content authors and marketers create and edit content. They build and publish the web experiences from here.
- Sitecore Content Delivery – external environment for delivering content to the public; for example, the Storefront is hosted here. It serves requested pages and media assets to site visitors and collects analytics data in session, for delivering dynamic personalized experiences to shoppers.
- Sitecore Commerce – the Commerce Engine and Commerce databases. This environment provides the commerce services, for example, orders, carts, pricing, and promotions.
- Commerce Server – the legacy Commerce Server component is shown separately in this diagram. In this release, Commerce Server provides Catalog and Profile functions for the overall Commerce solution. In a future release, the Sitecore Commerce environment will host new, replaced Catalog and Profile services.

3.3 Commerce Entity

A commerce entity represents a core unit of persistence in the form of a POCO class that inherits from commerce entity and can extend it along with behavior defined based on lifecycle events of the entity or defined commands.

Commerce entities are designed to be serialized rapidly into JSON and stored in a variety of persistent stores. This provides the ability to support rich, deep extensible business entities with a simplistic API enabling a wide variety of persistence options.

Leveraging a document-oriented philosophy unlocks key capabilities seen and valued in CMS systems like Sitecore, such as versioning, drafting, and so on.

Commerce entities are:

- Simple inheritable base classes.
- Usually represent a real business concept.
- Persist as documents using serialization.
- Extended by composition.
- Extended by policy.
- Independently addressable using persistence.

The following table describes the properties of a commerce entity:

Property	Description
Namespace	The first part of its unique identifier (string).
Id	Its unique identifier (string).
Name	Name of the entity (string).
FriendlyId	Human readable instance of the unique identifier (string).
DisplayName	Displayable name of the entity (Localized<String>).
DateCreated	Date and time the entity was created, automatically updated (DateTime).
Description	Description of entity (Localized<String>).
DateUpdated	Date and time the entity was updated, automatically updated (DateTime).
Policies	List of policies applicable to this entity (List<Policy>).
Components	List of components applicable to this entity (List<Component>).
ListMemberships	Delimited list of lists names (string).
SortOrder	The order this item is sorted (string).
IsDeleted	Flag to signify that the item is deleted (Bool).
IsPersisted	Flag to signify that the item is persisted (Bool).
DateDeleted	Date and time that the entity was deleted (DateTime).

3.4 EntityStore

A Sitecore Commerce EntityStore is an abstract mechanism for specifying the persistence and retrieval of Sitecore Commerce Entities. This provides a customization point that allows the solution developer to customize where and how entities are persisted. Sitecore Commerce uses an SQL-based EntityStore.

Characteristics of an EntityStore include:

- Simple extensible entity storage.
- Policy driven.
- Leverages plugins for persistence.
- Distributed caching.
- Pluggable encryption.
- Uniform transactional model.

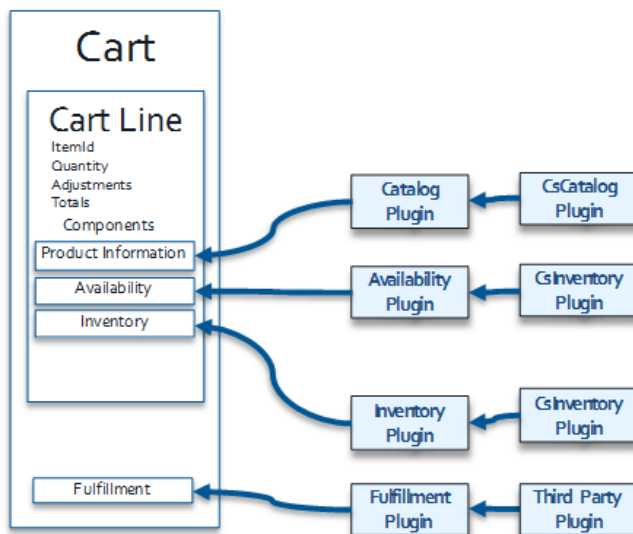
3.5 Compositional Extensibility

Compositional Extensibility is a simple class supplied by plugins as a way of extending a commerce entity. Components can be added or removed by a plugin into a commerce entity by adding to the components property. This is usually performed in a `PipelineBlock` or a `RuleAction`.

Only one of a specific type of component is allowed in a commerce entity. This should be enforced on the persistence of an entity. Other plugins do not need to have knowledge of components supplied by other plugins, unless they want to.

The existence of a component in an entity may trigger business actions. For example, if a delivery component is on a `LineItem` of a shopping cart upon order creation, then business processes are invoked by the delivery pipeline.

Compositional extensibility is exemplified in the following figure:



3.6 Commerce List

A managed commerce list represents a core mechanism for organizing and relating commerce entities. Support is provided for simplistic named lists as well as more tightly managed lists. Lists can be curated by picking individual members of the list or implemented as expressions against the search provider for dynamic listing.

Lists can be iterated using common paging metaphors or treated like a queue with push and pop semantics. Lists can also be used as work queues for commerce minions to traverse and perform work against. Lists can have policies that govern what kind of entities can access the list, how many entities, caching, and so on.

For example, a policy must define whether an entire list is retained in memory once loaded, or whether to iterate as a search list.

The following table describes commerce list actions, commands, and pipelines:

Action/Command/Pipeline	Description
AddRelatedManagedList AddRelatedManagedListCommand AddRelatedManagedListPipeline	Creates a relationship between two managed lists . The parameters are: <ul style="list-style-type: none"> listName – name of the parent lists (string). childListName – the new child list name (string).
CreateManagedList CreateManagedListCommand CreateManagedListPipeline	Creates a new managed list. The parameters are: <ul style="list-style-type: none"> name – the name of the list (string). displayName – a displayable name of the list (string).

Action/Command/Pipeline	Description
	<ul style="list-style-type: none"> <code>relatedListNames</code> – a list of related lists by name (list<string>). <code>policies</code> – a list of policies applying to the managed list (list<policy>). <code>components</code> – a list of components to add to the new managed list (list<component>).
<code>GetManagedList</code> <code>GetManagedListCommand</code> <code>GetManagedListPipeline</code>	Retrieves a new managed list. The parameter is: <ul style="list-style-type: none"> <code>listName</code> – name of the list (string).
<code>LogListMetadata</code> <code>LogListMetadataCommand</code> (no pipeline)	Logs the count of a list to the logging system. The parameter is: <ul style="list-style-type: none"> <code>listName</code> – name of the list (string).

Sitecore Commerce Plugins

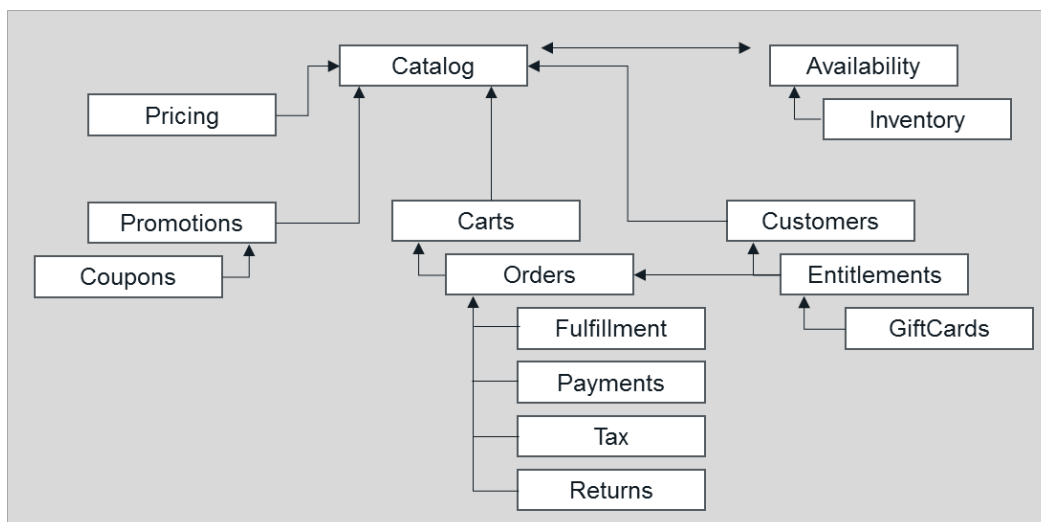
A plugin is an independently publishable extension to the Sitecore Commerce Engine. They are published as a lightweight NuGet package, either to a public repository or a secure private one. Solution developers can acquire plugins or create their own to support their own specific scenarios. Similarly, Sitecore partners create Plugins to use as value-added IP in commerce engagements.

Plugins can contribute the following:

- Entity
- Component
- Command
- Pipeline
- Model
- Policy
- Event
- List

A plugin can take a dependency on another plugin in order to extend its functionality.

The following figure presents some of the key plugins provided with the Sitecore Commerce solution, including a simplified view of some of the relationships between them:



3.7 Entity Journaling

Entity journaling is provided by the **Sitecore.Commerce.Plugin.Journaling** plugin.

Entity journaling enables an entity to be flagged to have a journal updated whenever it changes. This saves a complete previous copy of the entity, enabling a complete log of changes to be tracked.

Journaling is defined by the `EntityJournalingPolicy` policy.

The `EntityJournalingPolicy` has the following properties:

Property	Description
<code>EntityFullName</code>	The full name of the entity (string).
<code>Journal</code>	The logical journal to associate this entry (string).

The following is an example of a new `EntityJournalingPolicy`:

```
{
    EntityFullName = "Sitecore.Commerce.Plugin.Orders.Order",
    Journal = "OrdersJournal"
}
```

The `EntityPersistenceJournalingBlock` checks the `EntityJournalingPolicy` for rules on how to journalize the entity. An entity can easily have journaling added or removed by changing the policy. If journaling is disabled for an entity where it was previously enabled, the journal entries are not removed; it will simply no longer create new ones.

3.8 Sitecore Commerce Service API

Sitecore Commerce has been designed to interact with other external entities as part of ongoing operations. It achieves this through the Service API. The primary focus of the Service API is to provide the ability to execute Sitecore Commerce commands through the service endpoint.

Commands can be short running or long running. Long running commands return a token that can be used to check the status of a command. Commands can be added through a Sitecore Commerce plugin. Commands are request/response oriented.

The Sitecore Commerce ServiceAPI is further segregated into role-oriented APIs to allow targeting of specific entities, actions, and responses to meet specific business role needs.

The following table describes role-oriented APIs within the Service API:

Service API	Description
<code>CommerceAuthoring API</code>	Surfaces artifacts and methods targeted toward business user experiences, for creating and updating content to be published.
<code>CommerceShops API</code>	Surfaces artifacts and methods targeted toward supporting an online shopping experience, such as a web storefront.
<code>CommerceOps API</code>	Surfaces artifacts and methods targeted toward a DevOps role in managing a Sitecore Commerce implementation. This includes methods to create and manage environments and global policies.

Odata Compliance

The Service API is Odata compliant. Odata provides metadata that allows external systems to discover capabilities and data structures of Sitecore Commerce. Odata also supports annotations on metadata, which provides basic validation requirements and enumerations so that smarter clients can pre-validate and provide drop-down support for enumerable properties.

Odata provides the option to generate a strongly typed client proxy to facilitate access to the capabilities in Sitecore Commerce.

Developer's Guide

Most Microsoft products can natively consume Odata. For example, Excel can connect to an Odata source allowing the use of Power Pivot or other Excel-based analytic tools to be easily integrated. Other Microsoft products, including Microsoft Dynamics AX, support Odata in their service layers.

Service Metadata

The Sitecore Commerce API is based on Odata and therefore provides all the built-in benefits that Odata includes, such as metadata discovery.

The following is a sample call to retrieve the Sitecore Commerce metadata:

```
http://{{ServiceHost}}/{{ShopsApi}}/$metadata
```

Service API Headers

Sitecore Commerce uses headers passed in through the service APIs to establish context for the call. These headers are available from within the `CommerceContext`, which is passed around on all calls.

There are standard known headers that are supported out-of-box and this is an extensibility point where additional headers can be established for specific plugins. Plugins can reference these headers when taking actions.

Some headers are only relevant in certain actions; however, it is not harmful to include a header even if it is not used or needed for the particular call.

The following headers are available:

Header Name	Description
ShopName	The name of the current shop being accessed (string).
ShopperId	A unique identifier for a shopper (normally a GUID represented as a string).
CustomerId	If the shopper is registered, then an additional identifier is passed in, representing the unique identifier of a registered customer (normally a GUID represented as a string).
Currency	The currency that is desired in the response (string).
Language	The language desired in the response (string).
EffectiveDate	The effective date to use during any date-based calculations. This allows scenarios where you want to see results as if the interactions occurred at dates and times not in the present. If an <code>EffectiveDate</code> is not passed in, then <code>DateTime.UtcNow</code> is used.
Latitude	The current latitude of the shopper (string).
Longitude	The current longitude of the shopper (string).
IpAddress	The IP address of the shopper (string).
Roles	A " " delimited list of roles for the caller. This can influence what actions are allowed and what information is returned in queries, or whether the query can be performed at all (string).
IsRegistered	An indicator whether the current caller is registered or not (boolean represented as a string).

Security

General security is enforced at the service endpoint to determine whether a remote party can connect at all. Security is based on certificates or on specific authenticated identity(ies).

Security is enforced at the command level. Individuals are able to either execute specific commands or not. This is managed within the Sitecore Experience Platform's management tooling. The service API resolves and passes in claims that Sitecore Commerce uses to enforce this security.

3.9 Localization

Messages returned from the Sitecore Commerce Engine are keyed and stored in the Sitecore Experience Platform to prepare them for localization.

Localization is governed by the following policy:

Policy	Description
LocalizedMessagesPolicy	Defines localization settings: <ul style="list-style-type: none">• AllowCaching - the default is true• AllowCaching - the default is true• MaxMessageSize - the default is 1024

Chapter 4 Commerce Views Service

The Commerce Views service is provided by the **Sitecore.Commerce.Plugin.Views** plugin.

The Sitecore Commerce Views plugin provides a data-driven mechanism for servicing a dynamic business experience. The views provide a mechanism for narrowing or translating data from core entity storage into a dynamic API that can be directly leveraged by the business experience.

Typical model-view-controller architectures have a view layer that translates potentially complex internal storage artifacts into a simpler form meant to be consumed by a user experience. In many cases, each individual client type (web, mobile, and so on) has its own view layer in which they transform data into a viewable form.

In the Sitecore Commerce architecture, this view layer is provided on the server side, instead of the client side. This allows standard Sitecore Commerce plugins to extend existing views and add new views.

Sitecore Commerce views are also dynamic, which means they are progressively built up by plugins in a pipeline, which means the solution developer can develop plugins that extend views without breaking the API.

Sitecore Commerce views are also designed to directly feed a dynamics data-driven business tool experience. Functionality in the business tools can be lit up, extended, or removed by custom-developed plugins, without having to modify the out-of-box plugins or modify the source code of the business tools themselves. This approach dramatically improves upgradability.

4.1 Authoring API

Sitecore Commerce views are surfaced via the Authoring API. This API is based on Odata; however, it is significantly simplified and much more dynamically structured to support a data-driven user experience.

The Authoring API is targeted at authoring/management experiences, whereas the Shops API is targeted at runtime experiences for a shop or storefront. The Authoring API is logically separated from the Shops API to allow more direct control over the availability of functionality in different engine roles.

For example, a solution developer may not want the Authoring API to be callable in the engine that is serving storefronts, but only on an internal instance of the engine.

4.2 EntityViews

The core artifact in Sitecore supporting views is the EntityView. An EntityView represents a flattened, dynamic service response focused on supporting a dynamic user experience.

The EntityView provides a business user experience that is completely customizable and extensible and can dynamically represent and allow the business user to take actions, without requiring significant intelligence or customization of the user experience itself.

The EntityView is a simple POCO class that provides a property bag. It can have child EntityViews. Using these simple concepts, more complex views can be progressively built up as needed.

4.3 Composite EntityViews

To understand a Composite EntityView, consider a page in the user interface, for example, the page where the business user views an order. The order page is further broken up into sections (details, shipments, payments, entitlements, and so on). The Composite EntityView represents how that page is organized and

what actions are available so that the user interface can simply render the view to form a usable, extensible page without having any custom logic. A Composite EntityView includes a master view and child views.

The master view represents the overall page itself. It normally does not contain properties, but could. The master view contains a list of child views that each represent a section of the user interface. It also contains a list of actions that can be executed against the entity. An example of a child view is the details section of a user interface. For example, on an order page, the child view is the section that presents the order confirmation identifier, the date, and so on.

The view represents what should be displayed in the section, so it could be extended by plugins to display additional information. Or, properties could vary based on some aspect of the entity, such as the type of order. It is also possible to display some properties to some users and not others; this is a form of security trimming.

If an order does not need to have a section, it simply does not return one. For example, a digital product order would not have a shipments child view returned.

The view also provides an extensibility point, so the solution developer can deploy a plugin that ties into the pipelines and return additional child views, or augment existing views with new properties.

Using the master view allows the entire web page to be returned in one call, promoting efficiency. It is also possible to call an individual child view separately. This might be required, for example, when the user wants to refresh a single section without rebuilding the entire page.

4.4 EntityView Properties

The following table describes the EntityView properties:

Property	Description
Name	Logical name of the view (string).
DisplayName	Localized name of the view (string).
EntityId	Most views are centered on a particular underlying CommerceEntity. This EntityId represents the ID of that Commerce Entity (string).
Action	Represents an action to be performed. In some cases, EntityViews are retrieved, modified, and then submitted back as an action. This provides a task-based API for the Business User tooling (string).
ItemId	In some cases, an EntityView represents a single row among many in a list view. In the case where another level of ID is needed to further target a particular row, ItemId is used for this (string).
Properties	A list of ViewProperty objects. This represents individual properties that will display in the view. The Business Tool has generic functionality to create a display based on the properties of the View (list<ViewProperty>). See the following section for a further description of the EntityView <code>Property</code> property.
ChildViews	Views are progressively built up as a main view and a set of child views. This can be used to build more complex displays.
DisplayRank	In some cases, you might want to fine-tune the order of display. You can use DisplayRank to influence the order of display. All EntityViews default to a rank of 500.
UiHint	A UiHint is intended to be a design-agnostic hint to the UX on how to display the view. The view can be displayed as a flat list of properties, or it can designate that underlying child views represent a list to be displayed using a listing metaphor.

Property

The EntityView `Property` property is a list that facilitates the display of properties in a dynamic user interface. Each property has a set of values that defines what to display, as well as hints for the user interface to allow it to choose the proper control for the display of the property.

Developer's Guide

Any plugin can influence what is returned by adding, removing, or modifying properties before the view is returned. This enables easy customization of the dynamic business user experience without modifying the client itself. It provides an opt-in complexity philosophy by enabling properties to be present only when they are valid for the scenario.

The following table describes `Property` collection properties:

Property	Description
Name	The common name of the property. This can be shared among views and is used to localize the <code>DisplayName</code> (string).
DisplayName	The localized name of the property. This is currently not localized so it is the same as the Name field until localization is in place (string).
RawValue	This is the direct value from the mapping with no formatting. It is the value formatted using policy-based formatting (string).
Policies	This contains any constraints, closed vocabularies, and so on, as policies for actions (string).
Policy<T>	Retrieves or creates the policy instance that you are looking for (string).

4.5 EntityView Actions, Commands, Pipelines

The following tables describes the EntityView actions, commands, and pipelines:

Action/Command/Pipeline	Description
<code>GetEntityView</code> <code>GetEntityViewCommand</code> <code>GetEntityViewPipeline</code>	Retrieves an EntityView. The parameters are: <ul style="list-style-type: none">• <code>entityId</code> – the fully qualified identifier of the entity (string).• <code>viewName</code> – the name of the view (Summary, Preview, and so on) (string).• <code>forAction</code> – indicates whether a view can be edited and returned. This returns additional metadata, such as constraints, closed vocabularies, and so on (string).• <code>itemId</code> – the identifier of the item (string).

4.6 EntityView Samples

This following sections list some sample EntityViews.

4.6.1 EntityView Sample – Order Preview

Properties added:

Provided by the Orders plugin:

```
OrderConfirmationId<=order.OrderConfirmationId
OrderPlacedDate<=order.OrderPlacedDate
Status<=order.Status
PaymentStatus<=order.PaymentStatus
ShopName<=order.ShopName
CustomerEmail<=order.GetComponent<ContactComponent>().Email
OrderTotal<=order.Totals.GrandTotal
```

Sample call to retrieve EntityView:

```
http://{{ServiceHost}}/{{ShopsApi}}/GetEntityView()
```

Body:

```
{
  "entityId": "006fa25e-d3ca-4e05-97fb-55afd8568e2e",
  "viewName": "Preview",
  "forAction": ""
}
```

Sample results from the call:

```
{
  "@odata.context":
"http://localhost:5000/Api/$metadata#Sitecore.Commerce.EntityViews.EntityView",
  "@odata.type": "#Sitecore.Commerce.EntityViews.EntityView",
  "Name": "Preview",
  "Policies": [],
  "DisplayName": "Preview",
  "EntityId": "{006fa25e-d3ca-4e05-97fb-55afd8568e2e}",
  "Action": "",
  "ItemId": "",
  "Properties": [
    {
      "Name": "OrderConfirmationId",
      "Policies": [],
      "DisplayName": "Order Confirmation Id",
      "Value": "{006fa25e-d3ca-4e05-97fb-55afd8568e2e}",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": false,
      "UiType": ""
    },
    {
      "Name": "OrderPlacedDate",
      "Policies": [],
      "DisplayName": "Placed Date",
      "Value": "8/19/2016 9:52:02 PM",
      "IsHidden": false,
      "OriginalType": "DateTime",
      "IsReadOnly": false,
      "UiType": ""
    },
    {
      "Name": "Status",
      "Policies": [],
      "DisplayName": "Status",
      "Value": "Completed",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": false,
      "UiType": ""
    },
    {
      "Name": "PaymentStatus",
      "Policies": [],
      "DisplayName": "Payment Status",
      "Value": "Paid",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": false,
      "UiType": ""
    },
    {
      "Name": "ShopName",
      "Policies": [],
      "DisplayName": "Shop Name",
      "Value": "Storefront",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": false,
      "UiType": ""
    },
    {
      "Name": "OrderTotal",
      "Policies": [],
      "DisplayName": "Order Total",
      "Value": "USD1,034.00",
      "IsHidden": false,
      "OriginalType": "Money",
      "IsReadOnly": false,
      "UiType": ""
    }
  ],
  {
```

```
        "Name": "CustomerEmail",
        "Policies": [],
        "DisplayName": "Customer Email",
        "Value": "email@domain.com",
        "IsHidden": false,
        "OriginalType": "String",
        "IsReadOnly": false,
        "UiType": ""
    }
],
"ChildViews": [],
"DisplayRank": 500,
"UiHint": "Flat"
}
```

4.6.2 EntityView Sample – Order Summary

Properties added:

Provided by the Orders plugin:

```
OrderConfirmationId<=order.OrderConfirmationId
OrderPlacedDate<=order.OrderPlacedDate
DateUpdated<=order.DateUpdated
Status<=order.Status
PaymentStatus<=order.PaymentStatus
ShopName<=order.ShopName
CustomerEmail<=order.GetComponent<ContactComponent>().Email
OrderSubTotal<=order.Totals.SubTotal
OrderAdjustmentsTotal<=order.Totals.AdjustmentsTotal
OrderGrandTotal<=order.Totals.GrandTotal
OrderPaymentsTotal<=order.Totals.PaymentsTotal
```

Provided by the GiftCards plugin, for example, if the order is purchasing a gift card:

```
GiftCardPurchased<=line.GetComponent<GiftCardComponent>().GiftCardCode
GiftCardAmount<=line.GetComponent<GiftCardComponent>().GiftCardAmount
```

Provided by the AdventureWorks plugin. This is a sample customization where the order uses a coupon:

```
CouponUsed<=line.GetComponent<CartCouponsComponent>().Name
```

Sample call to retrieve EntityView:

```
http://{{ServiceHost}}/{{ShopsApi}}/GetEntityView()
```

Body:

```
{
  "entityId": "{006fa25e-d3ca-4e05-97fb-55afd8568e2e}",
  "viewName": "Summary",
  "forAction": ""
}
```

Sample results from the call:

```
{
  "@odata.context":
"http://localhost:5000/Api/$metadata#Sitecore.Commerce.EntityViews.EntityView",
  "@odata.type": "#Sitecore.Commerce.EntityViews.EntityView",
  "Name": "Summary",
  "Policies": [],
  "DisplayName": "Summary",
  "EntityId": "{006fa25e-d3ca-4e05-97fb-55afd8568e2e}",
  "Action": "",
  "ItemId": "",
  "Properties": [
    {
      "Name": "OrderConfirmationId",
      "Policies": [],
      "DisplayName": "Order Confirmation Id",
      "Value": "{006fa25e-d3ca-4e05-97fb-55afd8568e2e}",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": true,
      "UiType": ""
    }
  ]
}
```

```

    },
    {
      "Name": "OrderPlacedDate",
      "Policies": [],
      "DisplayName": "Placed Date",
      "Value": "8/19/2016 9:52:02 PM",
      "IsHidden": false,
      "OriginalType": "DateTime",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "DateUpdated",
      "Policies": [],
      "DisplayName": "Date Updated",
      "Value": "8/19/2016 9:52:09 PM",
      "IsHidden": false,
      "OriginalType": "DateTime",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "Status",
      "Policies": [],
      "DisplayName": "Status",
      "Value": "Completed",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "PaymentStatus",
      "Policies": [],
      "DisplayName": "Payment Status",
      "Value": "Paid",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "ShopName",
      "Policies": [],
      "DisplayName": "Shop Name",
      "Value": "Storefront",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "CustomerEmail",
      "Policies": [],
      "DisplayName": "Customer Email",
      "Value": "email@domain.com",
      "IsHidden": false,
      "OriginalType": "String",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "OrderSubTotal",
      "Policies": [],
      "DisplayName": "Order Sub Total",
      "Value": "USD940.00",
      "IsHidden": false,
      "OriginalType": "Money",
      "IsReadOnly": true,
      "UiType": ""
    },
    {
      "Name": "OrderAdjustmentsTotal",
      "Policies": [],

```



```
        "DisplayName": "Order Adjustments Total",
        "Value": "USD94.00",
        "IsHidden": false,
        "OriginalType": "Money",
        "IsReadOnly": true,
        "UiType": ""
    },
    {
        "Name": "OrderGrandTotal",
        "Policies": [],
        "DisplayName": "Order Grand Total",
        "Value": "USD1,034.00",
        "IsHidden": false,
        "OriginalType": "Money",
        "IsReadOnly": true,
        "UiType": ""
    },
    {
        "Name": "OrderPaymentsTotal",
        "Policies": [],
        "DisplayName": "Order Payments Total",
        "Value": "USD413.60",
        "IsHidden": false,
        "OriginalType": "Money",
        "IsReadOnly": true,
        "UiType": ""
    }
],
"ChildViews": [],
"DisplayRank": 500,
"UiHint": "Flat"
}
```

4.7 EntityActions

An **EntityAction** provides a simple mechanism to extend an **EntityView**. It allows a user to edit the values in an **EntityView**, then resubmit it for implementation. **EntityActions** can apply globally on the **EntityView**, and also at the **ChildView** level.

An example action is one that requires the user to select an entry in a list in order to focus an action, for example, selecting a particular line in a list of order lines, in order to take action on that particular line.

Action Process

When the user clicks on a button indicating they want to take an action, the button supplies the name of the view and the action name. The view the button provides represents a set of properties that must be populated in order for the action to occur.

The user interface then calls for the view, which provides the **ForAction** property. The **ForAction** property indicates that the user intends to edit the view, so additional policies are returned to support editing (for example, constraints, closed vocabulary list, and so on).

Viewless Actions

It is possible that there is no view required for an action, for example, when deleting an entity. It already has everything it needs, so there is no additional information to be collected from the user to take the action.

The ***RequiresConfirmation*** property indicates the user interface should supply a dialog box to confirm if the user is sure of the action before allowing it, or if the action should just happen after the user clicks the button.

If there is a view, then the a dialog box opens and displays properties from the view in an editable display. When the user clicks OK, the view is sent back to the Commerce Engine. The engine now has the properties that are populated and the desired action, so the action can be processed to return a normal command result.

Multistep Actions

Some action executions require multiple steps. For example, when entering an address the user first selects the country, then the county-specific properties and close vocabulary lists are presented. For example, the ***StateProvince*** property provides the list of states or provinces depending on what applies to the selected country.

Chapter 5 Rules

The Rules service is provided by the **Sitecore.Commerce.Plugin.Rules** plugin.

The Sitecore Commerce Rules service is a generic infrastructure for applying rules in a Sitecore Commerce service. In the current Sitecore Commerce release, the Rules service is used with the Promotions service to assist in evaluating the qualifications for a promotion.

The Rules service could be applied in other scenarios, for example to customize or extend other services, or in future Sitecore Commerce releases.

The following Rules entity is provided:

Entity
RuleSet

5.1 Rules Commands and Pipelines

The following table describes the rules commands and pipelines:

Command/Pipeline	Description
BuildRuleSetCommand BuildRuleSetPipeline	Builds a rule set based on a set of rule models.
GetActionsCommand GetActionsPipeline	Retrieves the Actions of a specified type, or if no type is specified it retrieves all Actions.
GetConditionsCommand GetConditionsPipeline	Retrieves the Conditions of a specified type, or if no type is specified it retrieves all Conditions.
GetOperatorsCommand GetOperatorsPipeline	Retrieves the Operators of a specified type, or if no type is specified it retrieves all Operators.
RunRuleSetCommand RunRuleSetPipeline	Runs a rule set. Returns true if all the rules are evaluated to true, otherwise returns false.

5.2 Rules Models

The following table describes the rules models:

Model	Description
ActionModel	Defines an action within a rule
ConditionModel	Defines a condition within a rule
OperatorModel	Defines an operator within an action or condition
PropertyModel	Defines a property within an action or condition
RuleModel	Defines a rule

Chapter 6 Orders Service

The Orders service is provided by the **Sitecore.Commerce.Plugin.Orders** plugin.

The Orders service manages all aspects of the order lifecycle and ensures seamless control of customer orders, from order receipt to financial settlement. A key component of the orders service is the Shopping Cart service, which forms the initial step in the order workflow. The Shopping Cart service is described in the following chapter.

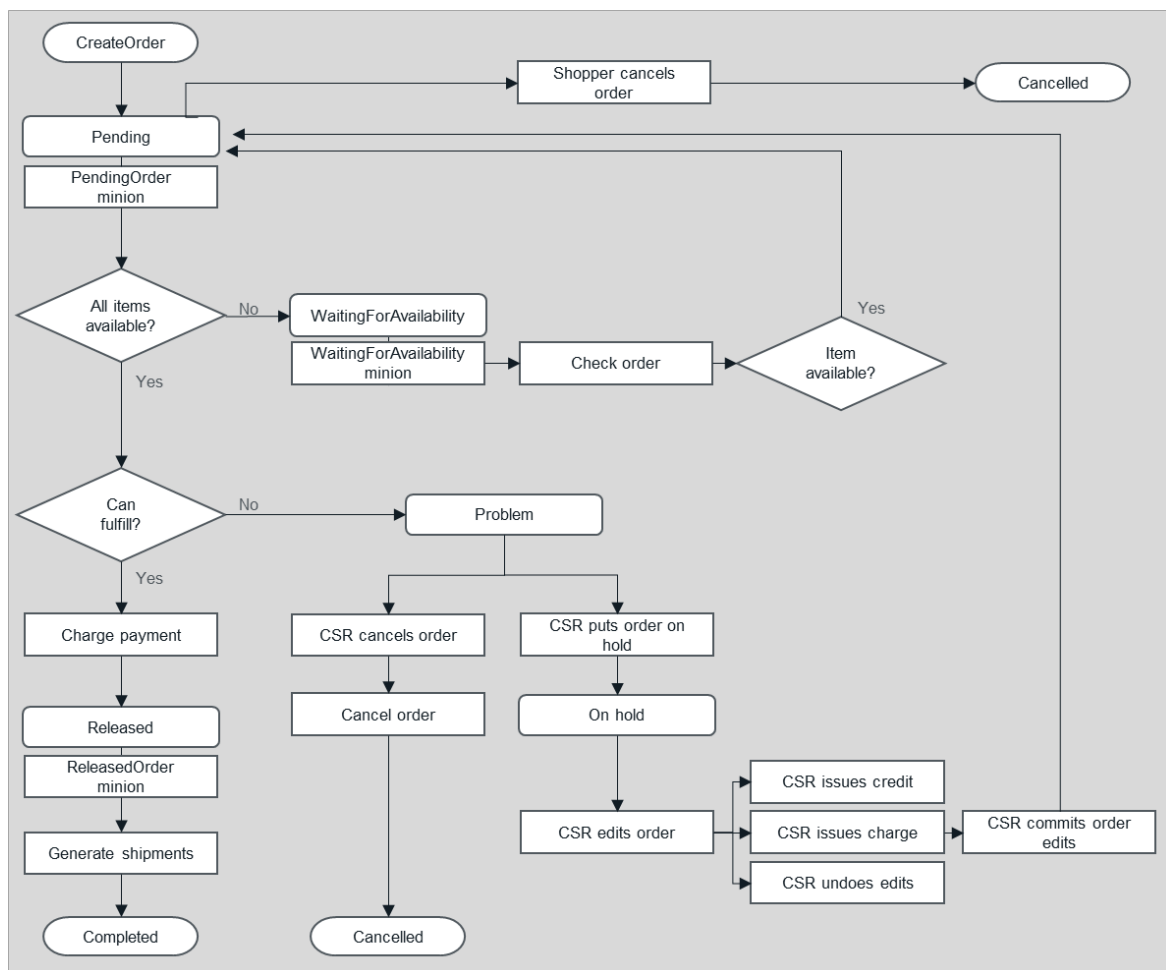
Orders capabilities include:

- Shopping Cart - manage interactions with a Shopping Cart
- Checkout - accept additional information during checkout
- Order Capture - capture an Order from a Shopping Cart after Checkout
- Post Order Capture Processing – support for capturing data after the order
- Shops Service – an interactive service to support shopping/Integration experiences
- Authoring Service – an interactive service to support Order Management by CSRs
- Refund & RMA Processing - manage customer merchandise returns
- Pluggable Tax Integration - enable integration with a third-party tax provider; basic tax calculation functionality is provided out-of-box for testing. You will need to customize the tax calculation code logic to reflect your own business and tax regulation requirements.
- Pluggable Payments Integration - enable integration with a third-party payment provider
- Pluggable Fulfillment Integration – enable integration with a third-party fulfillment provider, basic fulfillment functionality is provided out-of-box for testing
- Pluggable Inventory Integration – enable integration with a third-party inventory system; currently shipped with out-of-box integration with the Commerce Server inventory system.
- Pluggable Integration with External Systems - enable pluggable integration with other external systems such as ERP and fraud check

6.1 Orders Concepts

An order is created when a customer completes purchase of a product or service from the storefront web site. The order contains all the information necessary to process the order, such as customer information, date purchased, currency used, tax information, and more.

The following figure summarizes the Order flow:

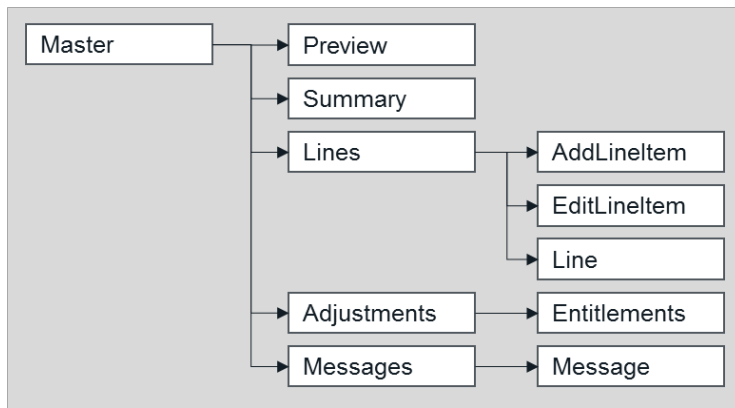


The following table lists the orders entities and component:

Entity	Components
Order	OnHoldOrder
Sales Activity	

6.2 Orders Views

The following figure displays the hierarchy of order views:



6.3 Orders Actions, Commands and Pipelines

The following table describes the orders actions, commands and pipelines pertaining to the Order entity:

Action/Command/Pipeline	Description
Orders GetOrderCommand GetOrderPipeline	Retrieves an order. Parameter: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order.
CreateOrder CreateOrderCommand CreateOrderPipeline And the following dependent pipelines: ItemOrderedPipeline OrderPlacedPipeline	Creates an order. This action is performed by a shopper from a storefront web site. Parameter: <ul style="list-style-type: none"> <code>id</code> – identifier of a Cart is passed into the order.
HoldOrder HoldOrderCommand HoldOrderPipeline	Places an order into on hold status so that no further processing can occur, for example temporarily while changes are being made to the order. This action is performed by a customer service rep. using the business tools. Parameter: <code>orderId</code> – the identifier of the order.
UndoOnHoldOrder UndoOnHoldOrderCommand UndoOnHoldOrderPipeline	Undoes the changes made to an order while the order is on hold. Parameter: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order.
CancelOnHoldOrder CancelOnHoldOrderCommand CancelOnHoldOrderPipeline	Cancels the order that is on hold. This is performed by a customer service rep., for example after having placed the order on hold, then decides to cancel it. Parameter: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order.
CancelOrder CancelOrderCommand CancelOrderPipeline	Cancels an order. This action is performed by a shopper from a storefront web site, or by a customer service rep. using business tools. The order must be in pending state, after which the order has been processed and needs to be canceled by an order return. Parameter: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order.
ReleaseOrder ReleaseOnHoldOrderCommand ReleaseOnHoldOrderPipeline	Releases an order that is on hold. Parameter: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order.
SetOrderStatus SetOrderStatusCommand SetOrderStatusPipeline	Sets the status of an order. Parameters: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order. <code>status</code> – the status to be set.

The following table describes the orders actions, commands and pipelines pertaining to the Sales Activity entity:

Action/Command/Pipeline	Description
SalesActivities	Retrieves one or all sales activities. If no parameter is passed, all available sales activities are returned. Parameter: <ul style="list-style-type: none"> SalesActivityId – the identifier of the sales activity
GetOrderSalesActivities GetOrderSalesActivityCommand	Retrieves sales order activities pertaining to an order. Parameter: <ul style="list-style-type: none"> orderId – the identifier of the order.
SettleSalesActivityPipeline	

6.4 Orders Models

The following orders models are provided:

Model
CreatedOrder
TemporaryCartCreated

6.5 Orders Policies

The following orders policies are provided:

Policy	Description
CancelOrdersPolicy	Defines whether orders are allowed to be cancelled.
GlobalCheckoutPolicy	Defines overall checkout settings including: <ul style="list-style-type: none"> Enable quick checkout Enable guest checkout Terms and conditions Minimum order quantity
GlobalOrderPolicy	Defined overall order settings including: <ul style="list-style-type: none"> Invoice prefix Invoice suffix Allow order cancel Submitted order list Completed order list Created order status
KnownOrderActionsPolicy	Provides the ability to change the default order action names.
KnownOrderListsPolicy	Provides the ability to change the default order lists names.
KnownOrderStatusPolicy	Provides the ability to change the default order status names.
KnownOrderViewsPolicy	Provides the ability to change the default order views names.
KnownSalesActivityStatusesPolicy	Provides the ability to change the default sales activity status names.
OnHoldOrdersPolicy	Defines whether orders are allowed to be placed on hold.

Chapter 7 Orders Service – Shopping Cart

The Shopping Cart service is provided by the **Sitecore.Commerce.Plugin.Carts** plugin.

The Shopping Cart (or Cart) is a virtual container on a web site that holds the products and services that a customer wants to purchase. It forms the initial step of the Order service.

A Cart has one or more line items that each present a **SellableItem**, a **Quantity** and a **SellPrice**.

The Cart is accessed from the Service API, and is not directly exposed to the user applications. Therefore, there are not Cart views or actions.

The following Cart entities are provided:

Entity
Cart
Cart Line

7.1 Cart Actions, Commands and Pipelines

The following table describes the cart commands and pipelines pertaining to the Cart entity:

Command/Pipeline	Description
GetCartCommand GetCartPipeline	Retrieves the cart record. If the cart does not exist it returns an empty cart. Parameter: <ul style="list-style-type: none"> <code>cartId</code> – the cart identifier (string)
MergeCartsCommand MergedCartsPipeline	Merges two cart records; for example a customer initially created a cart as an anonymous user, then authenticates and creates another cart. This action merges the previous anonymous cart into the authenticated cart. Recalculates the cart. Parameters: <ul style="list-style-type: none"> <code>fromCartId</code> – cart record that data is merged from (string) <code>toCartId</code> – cart record that data is merged to (string)

The following table describes the cart commands and pipelines pertaining to the Cart Line entity:

Command/Pipeline	Description
AddCartLineCommand AddCartLinePipeline	Creates a new cart line in the cart. Recalculates the cart. Parameters: <ul style="list-style-type: none"> <code>cartId</code> – the cart identifier (string) <code>itemId</code> – the cart line identifier <code>quantity</code> – the quantity of cart line items to add (decimal)
UpdateCartLineCommand UpdateCartLinePipeline	Updates a cart line within a Cart, for example updates its identifier or quantity. Recalculates the cart. Parameters: <ul style="list-style-type: none"> <code>cartId</code> – the cart identifier (string) <code>cartLineId</code> – the cart line identifier (string) <code>quantity</code> – the quantity of cart line items to add (decimal)
RemoveCartLineCommand RemoveCartLinePipeline	Removes a single cart line item from a Cart. Recalculates the cart. Parameters:

Command/Pipeline	Description
	<ul style="list-style-type: none"> <code>cartId</code> – the cart identifier (string) <code>cartLineId</code> – the cart line identifier (string)

The following table describes actions that are performed on a cart in the special case when an order is put on hold. When an order is put on hold and modified, the Commerce Engine in fact modifies the cart that created the order.

Action	Description
AddLineItem	<p>Adds a cart line item to an order that is on hold.</p> <p>Parameters:</p> <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order. <code>itemId</code> – the identifier of the cart line item to add to the order. <code>Quantity</code> – quantity of the line item to add.
EditLineItem	<p>Edits a cart line item of an order that is on hold.</p> <p>Parameters:</p> <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order. <code>itemId</code> – the identifier of the cart line item to edit. <code>Quantity</code> – quantity of the line item to edit.
DeleteLineItem	<p>Deletes a cart line item of an order that is on hold.</p> <p>Parameters:</p> <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order. <code>itemId</code> – the identifier of the cart line item to delete.

The following are general pipelines that extend other cart pipelines:

Action
CalculateCartPipeline
CalculateCartLinesPipeline
GetCartListPipeline
PopulateLineItemPipeline

7.2 Cart Models

The following cart models are provided:

Model
CartTotals
LineAdded
LineUpdated
Totals

7.3 Cart Policies

The following cart policies are provided:

Policy
GlobalCartPolicy
KnownCartAdjustmentTypesPolicy
LineQuantityPolicy
RollupCartLinesPolicy

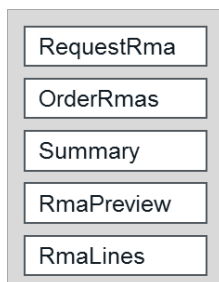
Chapter 8 Orders Service – Returns

The Returns service is provided by the **Sitecore.Commerce.Plugin>Returns** plugin.

The Returns service provides the ability to initiate a product return by requesting a Return Merchandise Authorization (RMA) and process the returned product.

8.1 Returns Views

The following figure displays the returns views:



8.2 Returns Actions, Commands and Pipelines

The following table describes the returns actions, commands and pipelines:

Action/Command/Pipeline	Description
RequestRma RequestRmaCommand RequestRmaPipeline	Initiates a request for a return merchandise authorization (RMA). Parameters: <ul style="list-style-type: none"> <code>orderId</code> – the identifier of the order that contains the returned item. <code>LineId</code> – the identifier of the order line item being returned. <code>Reason</code> – the reason the shopper has entered, for the return. <code>Quantity</code> – quantity of the line items being returned.
ValidateRmaRequestCommand ValidateRmaRequestPipeline	Validates the return request.
ReturnedItemReceived ReturnedItemReceivedCommand ReturnedItemReceivedPipeline	Initiates the acknowledgement that the returned items have been received, and that the processing of the refund can proceed. Parameters: <ul style="list-style-type: none"> <code>rmaId</code> – the identifier of the RMA record for the returned item(s). <code>refundPaymentId</code> – the identifier of the refund payment record.

Note

`StartOrderReturn` is not used.

8.3 Returns Models

The following returns models are provided:

Model
RmaAdded
RmaReason
RmaRequestValid

8.4 Returns Policies

The following returns policies are provided:

Policy	Description
KnownReturnsActionsPolicy	Provides the ability to change the default returns actions names.
KnownReturnsListsPolicy	Provides the ability to change the default returns lists names.
KnownReturnsStatusPolicy	Provides the ability to change the default returns status names.
KnownReturnsViewsPolicy	Provides the ability to change the default returns views names.
RequestRmaReasonsPolicy	Provides the ability to allow only specific reasons for the RMA, for example if the user interface presented a drop list where the shopper would be required to select one.

Chapter 9 Pricing Service

The Pricing service is provided by the **Sitecore.Commerce.Plugin.Pricing** plugin.

The Sitecore Commerce Pricing Service enables flexible, extensible, and dynamic pricing scenarios based on date, tiers, or aspects of the customer's environment. For example, the service actively calculates prices as the customer views products or services during an order.

Pricing capabilities include:

- Dynamic pricing – actively calculates prices in real time based on input from customers, for example, based on actions on the storefront. Actively calculates the List Price and Sell Price available for the customer, based on a set of conditions and policies.
- Is/Was pricing – calculates and displays to customers both the List Price (for example, MSRP) and a Sell Price (for example, a special discounted price) based on a set of conditions. A List Price could be displayed to the customer in struck-out font, to emphasize that the customer is getting a good price. Some regions or countries have laws regulating the display of regular versus sale prices, for example, preventing the elevation of the regular price; Sitecore Commerce software does not enforce such laws.

9.1 Pricing Concepts

In the current release, the catalog in Commerce Server is related to the Commerce Engine as follows:

Commerce Server	Commerce Engine
Catalog	Price Book Price Card

The following table presents the hierarchy of Commerce Engine pricing entities and components:

Entity	Components
Price Book	
Price Card	Snapshot Tier Tag

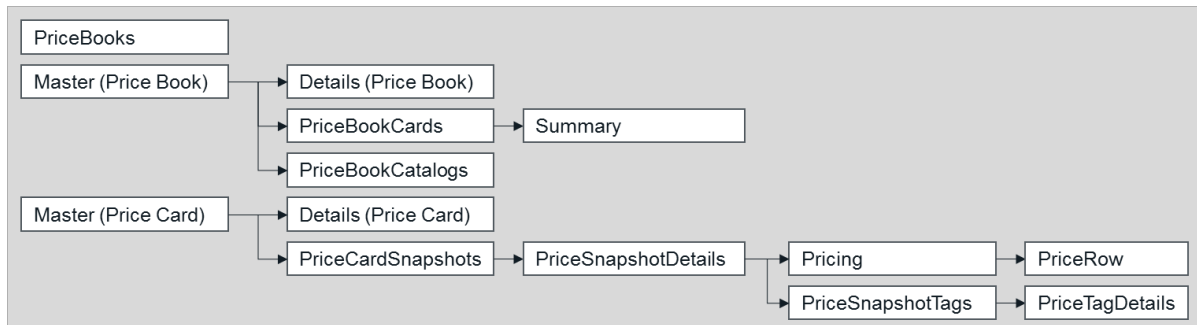
The following table describes general pricing terminology:

Pricing Term	Description
Catalog	An organization of products and services to sell. The Catalog is defined in groups of categories and products. The Catalog contains products and description data.
Price Book	A container for one or more Price Cards. Can be related to one or more Catalogs.
Price Card	An optional mechanism to define a Sell Price, which is normally a discounted price as compared to the List Price. Price Cards also make it possible to apply common pricing across multiple SellableItems without repeating the same pricing definition in each one. A Price Card can be a single application (for example, a reduced item price) or can contain multiple applications (for example, multiple promotions impacting a price at a particular time).
Snapshot	A calculated price that is defined to apply at a specified date/time (for example, at a snapshot in time). Snapshots are used to vary prices by date/time.

Pricing Term	Description
Tier	A categorization that can be used to present tier-based pricing, for example, by pricing by number of units bought, or by currency.
Tag	An indicator used to efficiently relate prices to particular SellableItems. By adding tags to a snapshot, you link a Price Card (at a particular date/time) to items that have the same tags applied.

9.2 Pricing Views

The following figure displays the hierarchy of pricing views:



The following table describes pricing views terminology, grouped by pricing entities and components:

Price Book Views	Description
PriceBooks	A list of all the price books.
Master	Displays the price book information.
Details	Displays the price book details.
PriceBookCards	A list of price book cards within the price book.
PriceBookCatalogs	A list of catalogs indicating whether they are associated or not with the price book.
Price Card Views	Description
Master	Displays the Price Card information.
Details	Displays the Price Card details.
Summary	Displays a subset of the Price Card's Details view.
Snapshot Views	Description
PriceCardSnapshots	A list of the Price Card's Snapshots.
PriceSnapshotDetails	Displays a Price Card's Snapshot details.
Tier Views	Description
Pricing	A list of the Snapshot's tiers.
PriceRow	Displays a Snapshot's Tier details.
Tag Views	Description
PriceSnapshotTags	A list of the Snapshot's Tags.
PriceTagDetails	Displays a Snapshot Tag's details.

Note

PriceSnapshotTiers and PriceTierDetails are not used.

9.3 Pricing Actions, Commands, Pipelines

The following table describes the pricing actions, commands, and pipelines pertaining to the Price Book entity:

Action/Command/Pipeline	Description
AddPriceBook AddPriceBookCommand AddPriceBookPipeline	Adds a price book. The price book name must be unique in the system.
EditPriceBook EditPriceBookCommand EditPriceBookPipeline	Edits a price book.
AssociateCatalog AssociateCatalogToBookCommand AssociateCatalogToBookPipeline	Associates a price book to a specific catalog.
DisassociateCatalog DisassociateCatalogFromBookCommand DisassociateCatalogFromBookPipeline	Disassociates a price book from a specific catalog.

The following table describes the pricing actions, commands, and pipelines pertaining to the Price Card entity:

Action/Command/Pipeline	Description
AddPriceCard AddPriceCardCommand AddPriceCardPipeline	Adds a price card to a price book. Requires a name and the parent's price book name. The price card name must be unique within the price book parameters.
EditPriceCard EditPriceCardCommand EditPriceCardPipeline	Edits a price card.
DeletePriceCard DeletePriceCardCommand DeletePriceCardPipeline	Deletes a price card. A price card can only be deleted if it does not contain approved snapshots.
DuplicatePriceCard DuplicatePriceCardCommand DuplicatePriceCardPipeline	Duplicates a price card and its snapshots.

The following table describes the pricing actions, commands, and pipelines pertaining to the Snapshot component:

Action/Command/Pipeline	Description
AddPriceSnapshot AddPriceSnapshotCommand AddPriceSnapshotPipeline	Adds a snapshot to a price card. Requires the snapshot's start date, which must be equal to or earlier than the latest approved snapshot date within the price card.
EditPriceSnapshot EditPriceSnapshotCommand EditPriceSnapshotPipeline	Edits a snapshot. The snapshot's approval status must be Draft.
RequestSnapshotApproval SetPriceSnapshotApprovalStatusCommand	Changes a snapshot's approval status from Draft to ReadyForApproval.
ApproveSnapshot SetPriceSnapshotApprovalStatusCommand	Changes a snapshot's approval status from ReadyForApproval to Approve.
RejectSnapshot SetPriceSnapshotApprovalStatusCommand	Changes a snapshot's approval status from ReadyForApproval to Draft.
RetractSnapshot SetPriceSnapshotApprovalStatusCommand	Changes a snapshot's approval status from Approved back to Draft. Applicable only before the snapshot is made fully active.
RemovePriceSnapshot RemovePriceSnapshotCommand RemovePriceSnapshotPipeline	Removes a price snapshot from a price card. The Snapshot's approval status must be Draft.
(internal - no action or command) ResolveActivePriceSnapshotByCardPipeline	Resolves the active snapshot within a price card. The active snapshot is an approved snapshot that

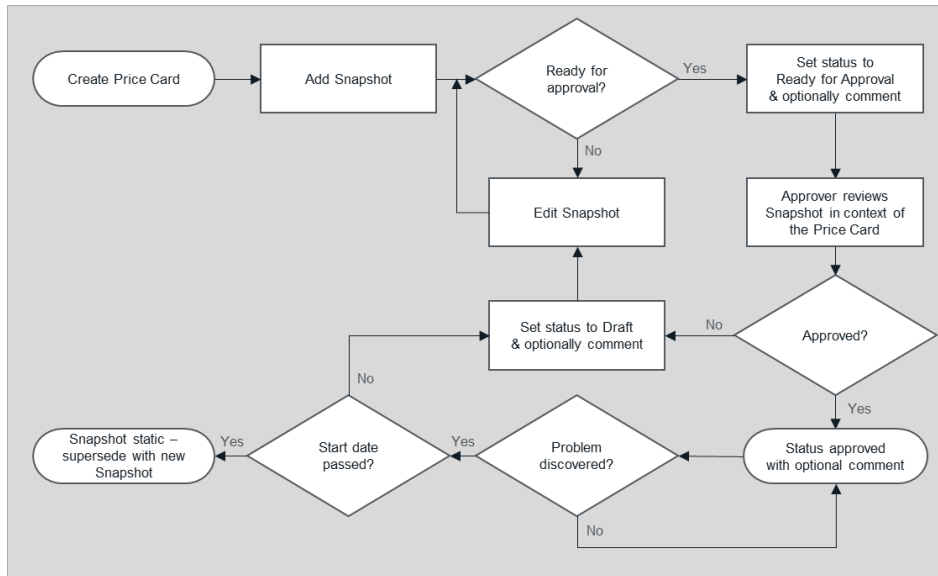
Action/Command/Pipeline	Description
	has a start date closest to the request's EffectiveDate.
(internal - no action or command) ResolveActivePriceSnapshotByTagsPipeline	Resolves the active snapshot within a price book by tags. The active snapshot is the approved snapshot that has start date closest to the request's EffectiveDate and matches the highest number of tags.

The following table describes the pricing actions, commands, and pipelines pertaining to the Tier component:

Action/Command/Pipeline	Description
AddCurrency AddPriceTierCommand AddPriceTierPipeline	Adds a price tier for a specific currency to a snapshot. The combination of a tier's quantity and currency must be unique within a snapshot. The snapshot's approval status must be set to Draft to add a tier.
EditCurrency EditPriceTierCommand EditPriceTierPipeline	Edits a snapshot's price tier for a specific currency. The snapshot's approval status must be set to Draft to edit a tier.
RemoveCurrency RemovePriceTierCommand RemovePriceTierPipeline	Removes a price tier from a snapshot for a specific currency. The snapshot's approval status must be set to Draft to remove a tier.
AddPriceTag AddPriceSnapshotTagCommand AddPriceSnapshotTagPipeline	Adds a tag to a snapshot. A tag must be unique within a snapshot. The snapshot's approval status must be set to draft to add a tag.
RemovePriceTag RemovePriceSnapshotTagCommand RemovePriceSnapshotTagCommand	Removes a tag from a snapshot. The snapshot's approval status must be set to Draft to remove a tag.

Pricing Approval Process

The following figure illustrates the pricing approval process:



9.4 Pricing Models

The following pricing models are provided:

Model
AssociatedCatalogModel
ItemPricing
PriceBookAdded
PriceCardAdded
PriceSnapshotAdded
PriceTier
PriceTierAdded

9.5 Pricing Policies

GlobalPricingPolicy

The `GlobalPricingPolicy` policy provides the ability to specify standard price policies. It applies in the Commerce Shops service. The following sample shows the `GlobalPricingPolicy` returned on an environment policies collection. It can be overwritten for a particular shop by being added to the policies collection of that shop.

```

{
  "@odata.type": "#Sitecore.Commerce.Plugin.Pricing.GlobalPricingPolicy",
  "PolicyId": "54a2e0ab01574526a4d0adc9abf4b90e",
  "Models": [],
  "RuleSet": null,
  "ShouldRoundPriceCalc": true,
  "RoundDigits": 2,
  "MidPointRoundUp": true,
  "MaxPriceCardNameLength": 30,
  "MaxPriceBookNameLength": 30,
  "MinimumPrice": 0,
  "MinimumPricingQuantity": 0
}

```

KnownPricingActionsPolicy

The `KnownPricingActionsPolicy` policy provides the ability to change the default pricing action names.

KnownPricingListsPolicy

The `KnownPricingListsPolicy` policy provides the ability to change the default Pricing List names:

List Name	Default Value
PriceBookCards	PriceBooks-{o}-PriceCards

KnownPricingViewsPolicy

The `KnownPricingViewsPolicy` policy provides the ability to change the default Pricing View names.

ListPricingPolicy

The `ListPricingPolicy` policy provides the ability to specify a List Price in multiple currencies. The following sample shows the `ListPricingPolicy` returned on a `SellableItem` policies collection or one of its variants:

```
{
  "@odata.type": "#Sitecore.Commerce.Plugin.Pricing.ListPricingPolicy",
  "PolicyId": "db13def722dc488f86a0673b83eb85e8",
  "Models": [],
  "RuleSet": null,
  "Prices": [
    {
      "CurrencyCode": "USD",
      "Amount": 44.75
    },
    {
      "CurrencyCode": "CAD",
      "Amount": 45.75
    }
  ]
}
```

OnSalePolicy

The `OnSalePolicy` policy provides the ability to specify that a `SellableItem` is on sale. The following sample shows the `OnSalePolicy` returned on a `SellableItem` policies collection or in the policies collection of one of its variants:

```
{
  "@odata.type": "#Sitecore.Commerce.Plugin.Pricing.OnSalePolicy",
  "PolicyId": "b70814f6f15b4a0aaeea74e83af63c20",
  "Models": [],
  "RuleSet": null,
  "OnSale": true,
  "ShowInCart": false,
  "CartDisplay": "text"
}
```

PriceCardPolicy

The `PriceCardPolicy` policy provides the ability to specify a Price Card for the `SellableItem`. This allows the `SellableItem` to share a pricing definition with multiple other `SellableItems`. The same logical PriceCard name can be represented in multiple Price Books, providing a flexible method for specifying pricing. The following sample shows the `PriceCardPolicy` returned on a `SellableItem` policies collection or in the policies collection of one of its variants. It indicates that a common price definition is being referenced:


```
{
  "@odata.type": "#Sitecore.Commerce.Plugin.Pricing.PriceCardPolicy",
  "PolicyId": "2f770e056b9149088c8f57a84d9343ed",
  "Models": [],
  "RuleSet": null,
  "PriceCardName": "HabitatPriceCard"
}
```

PurchaseOptionMoneyPolicy

The `PurchaseOptionMoneyPolicy` policy provides the ability to specify a calculated Sell Price in multiple currencies. The following sample shows the `PurchaseOptionMoneyPolicy` returned on a `SellableItem` policies collection or in the policies collection of one of its variants. It represents what you would actually sell an item for based on the context of the call. This could be a dynamically calculated price based on environment factors, for example, the Date/Time or the quantity the customer intends to purchase based on what they have in the cart.

```
{
  "@odata.type": "#Sitecore.Commerce.Plugin.Pricing.PurchaseOptionMoneyPolicy",
  "PolicyId": "b70814f6f15b4a0aaeea74e83af63c20",
  "Models": [],
  "RuleSet": null,
  "SellPrice": {
    "CurrencyCode": "USD",
    "Amount": 10
  }
}
```

9.6 Pricing Transparency

Price transparency ensures that the mechanism for calculating pricing is testable, auditable, and reproducible. Price transparency is for internal use, and is not presented to shoppers. Transparency is achieved during price calculation, by adding pricing messages into the artifacts receiving the pricing. Messages convey information on pricing strategy in a succinct and traceable way, for example, how a particular price was calculated.

Pricing messages can be added to a `MessagesComponent` during price calculation. A `MessageModel` uses the `Name` property to designate the type of message. These messages are named `Pricing`, which allows subsequent processing to filter on them. The `MessagesComponent` has an `AddMessage` method that allows a message to easily be added by passing in the message type and text for the message. In some cases, multiple messages are written as pricing rules successively cause an overwrite of a previous calculated price.

During calculation, pricing messages are added to a:

- `SellableItem` – messaging is inserted into to a `MessagesComponent` in the `SellableItem`. This is first calculated at the item level and then calculated at each variant level for price overrides.
- `Price Cart` – messaging inserted into a `CartLineComponent` as part of calculating the price cart.

At the end of the pricing calculation a reconciliation step occurs, which ensures that there is both a `ListPrice` and a `SellPrice` for each item and its variation.

Calculating Sellable Item List Price

The sellable item list price is calculated in the `Plugin.Catalog.Cs.CalculateSellableItemListPriceBlock` plugin.

The list price is written into the `SellableItem` property `ListPrice`, which is of the type `Money`.

The following messages are generated when calculating the list price:

Message	Circumstance
ListPrice<=PricingPolicy: Price={listPrice.AsCurrency() }	Item has a <code>ListPricingPolicy</code> and that policy has an entry for the desired currency. Sample Item AW051-14 (no variation) Sample Item AW055-01,33 (with variation)

Message	Circumstance
ListPrice<FirstVariation.PricingPolicy: Price={listPrice.AsCurrency()} FirstVariation={variation.Id}	No matching item level ListPricingPolicy but item has variations. The system uses the first ListPricingPolicy that it finds with a matching currency.
ListPrice<=Template.PricingPolicy: Template={_itemTemplateByDefinition.Id}	There is no ListPrice in the item but the item has a template and is using the ListPricePolicy from the template.
ListPrice<=SellPrice: Price={SellPrice}	There is no ListPrice but there is a SellPrice, so the SellPrice is copied to the ListPrice.

Calculating Sellable Item Variation List Price

The sellable item variation list price is calculated in the **Plugin.Catalog.Cs.CalculateSellableItemListPriceBlock** plugin (CalculateVariationsListPrice).

Normally a variation inherits the parent item's ListPrice and only needs to specify its own pricing if it is different than the parent. These messages are added into the SellableItem variation level.

The following messages are generated when calculating variation list price:

Message	Circumstance
Variation.ListPrice<=Variation.PricePolicy: Variation={variation.Id} Price={listPrice.AsCurrency()}	The variation has a ListPricingPolicy and that policy has an entry for the desired currency.
Variation.ListPrice<=Item.PricingPolicy: Variation={variation.Id} Price={listPrice.AsCurrency()}	No variation-specific ListPricingPolicy; the ListPricingPolicy is retrieved from the parent item.
Variation.ListPrice<=Template.PricingPolicy: Variation={variation.Id} Price={listPrice.AsCurrency()}	There is no ListPrice at variation or item level; the ListPrice is retrieved from the item template.
Variation.ListPrice<=Variation.SellPrice: Price={Variation.SellPrice} Variation={variationid}	The variation does not have a ListPrice but has a SellPrice, so the SellPrice is copied to the ListPrice.

Calculating Sellable Item Sell Price

The sellable item sell price is calculated in the **Plugin.Catalog.Cs.CalculateSellableItemSellPriceBlock** plugin.

To calculate the sell price:

1. Find the PriceCardPolicy (new Policy) in the SellableItem policies list.
The PriceCardPolicy must contain the logical name of a PriceCard (note that the same logically named price card can exist in multiple price books).
2. If there is no PriceCardPolicy, then look for a PriceCard using the tagging.
3. Resolve the PriceBook to use by loading the catalog from Commerce Server. The identifier for the catalog is passed in to the call (new Pipeline - GetCatalog).
4. Check that there is a new attribute on the catalog (PriceBook - string with Id fragment to PriceBook).
5. Load the PriceCard instance using the PriceBook and the logical PriceCard name.
6. Resolve the proper PriceSnapshot to use by filtering out those with dates that are superseded by other PriceSnapshots to find the single PriceSnapshot that should apply.
7. In the PriceSnapshot, resolve the available PriceTiers by filtering on the currency passed into the call.
8. Calculate the SellPrice (purchaseOption.SellPrice) using an assumed quantity of 1.
9. Copy the filtered PriceSnapshot and PriceTiers into the SellableItem so it can be used in later calculations.

Developer's Guide

The following messages are generated when calculating the sell price:

Message	Circumstance
<code>SellPrice<=PriceCard.Snapshot: Price={sellPrice.AsCurrency()} Qty={tier. Quantity} PriceCard={priceCardPolicy.Pric eCardName}</code>	The SellableItem has a PriceCardPolicy and a valid PriceSnapshot and PriceTier can be determined. Sample Item AW051-14 (no variation) Sample Item AW055-01,33 (with variation)
<code>SellPrice<=Tags.Snapshot: Price={sellPrice.AsCurrency()} Qty={tier. Quantity} Tags='{string.Join(", ", snapshot.Tags.Select(c => c.Name))}'</code>	The SellableItem does not have a PriceCardPolicy but it has tags.
<code>SellPrice<=Catalog.Product.ListPrice: Price={sellPrice.AsCurrency()} Product={p roduct.ProductId} Catalog={product.Catalo gName}</code>	The SellableItem has no PriceCardPolicy and matches Commerce Server Catalog currency, therefore it uses the Commerce Server List Price as the SellPrice.
<code>SellPrice<=ListPrice: Price={ListPrice}</code>	The SellableItem has no SellPrice but has a valid ListPrice, so the ListPrice is copied to the SellPrice.

Calculating Sellable Item Variation Sell Price

The sellable item variation sell price is calculated in the **Plugin.Catalog.Cs.CalculateSellableItemSellPriceBlock** plugin (CalculateVariationsSellPrice).

Variation sell price messages appear in a MessageComponent at the variation level. The SellPrice is created in the variation as a PurchaseOptionMoneyPolicy at the variant level. There must be no PurchaseOptionMoneyPolicy at the variant level if the variant has the same sell pricing as the parent SellableItem.

The following messages are generated when calculating the variation sell price:

Message	Circumstance
Variation.SellPrice<=Variation.PriceCard.Snapshot: Price={variationSellPrice.AsCurrency()} Qty={tier.Quantity} Variation={variation.Id} PriceCard={variationPriceCardPolicy.PriceCardName}	The variation has a PriceCardPolicy, or if not present, uses the parent calculated PriceCardPolicy.
Variation.SellPrice<=Variation.Tags.Snapshot: Price={variationSellPrice.AsCurrency()} Variation={variation.Id} Qty={tier.Quantity} Tags='{string.Join(", ", snapshot.Tags.Select(c => c.Name))}'	The variation has no PriceCardPolicy but has tags, or the parent SellableItem has tags.
Variation.SellPrice<=Catalog.Product.Variation.ListPrice: Price={variationSellPrice.AsCurrency()} Variation={variation.Id}	If there is no PriceCardPolicy and no parent SellableItem PriceCardPolicy and there are matches for the Commerce Server catalog currency, use the list price from the Commerce Server (using the ProductFamily lookup).
Variation.SellPrice<=Variation.ListPrice: Price={ListPrice} Variation={variationid}	The variation does not have a SellPrice but has a ListPrice, so the ListPrice is copied to the SellPrice.

Calculating Cart Line List Price

The cart line list price is calculated in the **Plugin.Catalog.Cs.CalculateCartLinesPriceBlock** plugin.

The following messages are generated when calculating the cart line list price:

Message	Circumstance
All from SellableItem	All pricing messages are copied from the SellableItem.
CartItem.ListPrice<=SellableItem.ListPrice: Price={line.UnitListPrice.AsCurrency() }	The list price is retrieved from the SellableItem.
CartItem.ListPrice<=SellableItem.Variation.ListPrice: Price={line.UnitListPrice.AsCurrency() }	The SellableItem has a variation and the list pricing is retrieved from the variation.
All Price messaging from Variation	All pricing messages from the SellableItem variation are copied into the line MessageComponent.

Calculating Cart Line Sell Price

The cart line sell price is calculated in the **Plugin.Catalog.Cs.CalculateCartLinesPriceBlock** plugin.

The following messages are generated when calculating the cart line sell price:

Message	Circumstance
All from SellableItem	All pricing messages are copied from the SellableItem.

Message	Circumstance
<code>CartItem.SellPrice<=SellableItem.SellPrice: Price={purchaseOptionPolicy.SellPrice.AsCurrency() }</code>	The <code>SellableItem</code> has a <code>PurchaseOptionMoneyPolicy</code> (indicating it can be sold in the currency desired).
<code>CartItem.SellPrice<=SellableItem.Variation.SellPrice: Price={purchaseOptionPolicy.SellPrice.AsCurrency() }</code>	The item has a variation and the variation has a <code>PurchaseOptionMoneyPolicy</code> .
<code>CartItem.SellPrice<=PriceCard.ActiveSnapshot: Price={purchaseOptionPolicy.SellPrice.AsCurrency() } Qty={tier.Quantity}</code>	The <code>SellableItem</code> has a calculated <code>PriceSnapshotComponent</code> and <code>line.Quantity <= tier.quantity</code> (takes first in descending order).

Chapter 10 Promotions Service

The Promotions service is provided by the **Sitecore.Commerce.Plugin.Promotions** plugin.

The Sitecore Commerce Promotion Service provides the underlying infrastructure and functionality for defining, evaluating, and applying promotions to products, at defined levels of granularity.

A promotion is an artifact that defines a set of qualifications for awarding a benefit, or a collection of benefits. Promotions can be configured to apply in a number of forms. For example: percent off, amount off depending on the currency, free shipping, and access to a service. Promotions can be applied to a product, or at the cart level. When a promotion is applied to the cart, the promotion can be calculated in real-time as part of the cart calculation.

Promotions are organized into promotion books. A promotion book is an entity that acts as a collection of individual promotions, which enables promotions to be stacked. A promotion book can apply to products in one or more catalogs, which enables promotions to be expressed for particular customer segments.

Promotions capabilities include:

- Flexible application – tie a promotion to a product, or a collection of products using tags.
- Promotion Books – flexible application of promotions using promotion books.
- Qualifications infrastructure – qualify for a promotion by rules-based expressions (for example, channel, date/time, catalog, shop, customer order history); apply multiple qualifications to a promotion.
- Flexible application of benefits awarded by a promotion – award a benefit to the item or to another item, award via a shopping cart level adjustment, award an entitlement or action on entitlement, award a fee adjustment, or award a gift into a shopping cart.
- Coupon management – promotions qualified using coupons, applied using public coupons (for example, multiuse, named coupons) or privately (for example, single use, single customer); allocate to specific uses in batches (for example, campaign mailings), and export to CSV for external marketing.
- Real-time promotion calculation – calculate and present awards in a shopping cart.
- Real-time bulk promotion calculation – retrieve possible promotions for an item to display in a Product Detail page.
- Promotion authoring – manage promotions from a business tool, through an Odata service.
- Centralized promotions storage – stored in a centralized repository, so that changes can be implemented without the need to synchronize data with other systems.
- Transparent – the process for how a promotion is derived can be tracked and audited.

10.1 Promotions Concepts

The following table describes the hierarchy of promotions entities and components:

Entity	Components
Promotion Book	
Promotion	Qualification Benefit Item

The following table describes general promotion terminology:

Promotion Term	Description
Promotion Book	A container for one or more promotions. May be related to one or more catalogs.
Promotion	A singled defined instance of a set of qualifications that result in one or a set of benefits.
Qualification	A set of rules to be satisfied in order to trigger the award of a benefit(s).
Benefit	A financial advantage awarded to a customer who's buying activities meet the qualification(s).
Item	A product or its variant, defined in the catalog.

10.2 Promotions – Qualifications

The following promotion qualifications are supported, and available in the Pricing and Promotions Manager Tool.

Product qualifications – related to the product inventory:

- Inventory Item Stock Count in [specific] Location [compares] to [specific value]?
- Is Item in Stock?
- Is Item in Stock in [specific] Location?
- Is Item Out of Stock?
- Is Item Out of Stock in [specific] Location?
- Is Item Pre-orderable?
- Is Item Pre-orderable in [specific] Location?
- Is Item Back-orderable?
- Is Item Back-orderable in [specific] Location?

Cart qualifications – applied at the cart level:

- Is Cart Item Available?
- Is Cart Item [specific] Quantity Available?
- Cart Has Items?
- Cart Any Item Has Template [compares] to [specific] Product Template?
- Cart Any Item Subtotal [compares] to [specific value]?
- Cart Subtotal [compares] to [specific value]?
- Cart Has Fulfillment?

Cart line qualifications – applied at the cart line level:

- Cart Item has [specific] tag?
- Cart Item Quantity [compares] to [specific value]?
- Cart Item Quantity is in [min] [max] Range?
- Cart Item Subtotal [compares] to [specific value]?

Shopper qualifications – related to an existing registered shopper:

- Is Cart Contact Registered?
- Is Cart Contact Currency [specific value]?
- Is Cart Contact Customer Id [specific value]?
- Is Cart Contact Language [specific value]?

Shopper history qualifications – related to the transaction history of the current shopper:

- Current Customer Has Purchased [specific] Item?
- Current Customer Has Purchased Item with [specific] Tag?
- Current Customer Orders Count [compares] to [specific value]?
- Current Customer Orders Total [compares] to [specific value]?

Environment qualifications – related to the current date:

- Current Date Has Passed?

- Is Current Day?
- Is Current Month?

Shop context qualifications – related to a particular shop:

- Is Shop Currency [specific value]?
- Is Shop Language [specific value]?
- Is Shop Name [specific value]?

10.3 Promotions – Benefits

The following promotion benefits are supported, and available in the Pricing and Promotions Manager Tool.

Cart adjustments:

- Get Cart Any Item Subtotal [specific] Amount Off
- Get Cart Any Item Subtotal [specific] Percent Off
- Get Cart Subtotal [specific] Amount Off
- Get Cart Subtotal [specific] Percent Off

Cart line adjustments:

- Get Cart Item Subtotal [specific] Amount Off
- Get Cart Item Subtotal [specific] Percent Off

Fulfillment adjustment:

- Get Free Shipping

Note

The Commerce Engine does not support promotions that rely on a cart subtotal to apply a benefit on a line item.

The Commerce Engine calculates the subtotals, taxes and fulfillment costs (including promotions) on line items before it performs those calculations at the cart level. A promotion that benefits a line item is evaluated first, but if its qualification depends on a cart subtotal, the condition cannot be met because cart totals are calculated by the next pipeline. In such cases, the promotion is not applied.

10.4 Promotions Samples

Sitecore Commerce is shipped with the following configured sample promotions, applicable to the Habitat and Adventure Works sample catalogs:

The following table describes the Habitat sample catalog:

Promotion Name	Promotion Description	Promo Type
CartFreeShippingPromotion	Free shipping when Cart subtotal of \$100 or more	automatic
Cart5OffExclusiveCouponPromotion	\$5 off Cart with subtotal of \$10 or more	exclusive coupon
Cart5PctOffExclusiveCouponPromotion	5% off Cart with subtotal of \$10 or more	exclusive coupon
Cart10OffCouponPromotion	\$10 off Cart with subtotal of \$50 or more	coupon
Cart10PctOffCouponPromotion	10% off Cart with subtotal of \$50 or more	coupon
Cart15PctOffCouponPromotion	15% off Cart with subtotal of \$50 or more	coupon
CartOptixCameraExclusivePromotion	50% off Cart when buying Optix Camera	exclusive auto.
Line5OffCouponPromotion	\$5 off any item with subtotal of \$10 or more	coupon
Line5PctOffCouponPromotion	5% off any item with subtotal of 10\$ or more	coupon
Line20OffExclusiveCouponPromotion	\$20 off any item with subtotal of \$50 or more	exclusive coupon
Line20PctOffExclusiveCouponPromotion	20% off any item with subtotal of \$50 or more	exclusive coupon
LineHabitat34withTouchScreen5OffPromotion	\$5 off the Habitat 34.0 Cubic Refrigerator with Touchscreen item	automatic
LineHabitat34withTouchScreenPromotion	50% off the Habitat 34.0 Cubic Refrigerator with Touchscreen item	automatic

Promotion Name	Promotion Description	Promo Type
LineMiraLaptopExclusivePromotion	50% off the Mira Laptop	exclusive auto.

The following table describes the AdventureWorks sample catalog:

Promotion Name	Promotion Description	Promo Type
CartFreeShippingPromotion	Free shipping when Cart subtotal of \$100 or more	automatic
Cart5OffExclusiveCouponPromotion	\$5 off Cart with subtotal of \$10 or more	exclusive coupon
Cart5PctOffExclusiveCouponPromotion	5% off Cart with subtotal of \$10 or more	exclusive coupon
Cart10OffCouponPromotion	\$10 off Cart with subtotal of \$50 or more	coupon
Cart10PctOffCouponPromotion	10% off Cart with subtotal of \$50 or more	coupon
Cart15PctOffCouponPromotion	15% off Cart with subtotal of \$50 or more	coupon
CartGalaxyTentExclusivePromotion	50% off Cart when buying Galaxy Tent	exclusive auto.
Line5OffCouponPromotion	\$5 off any item with subtotal of \$10 or more	coupon
Line5PctOffCouponPromotion	5% off any item with subtotal of \$10 or more	coupon
Line20OffExclusiveCouponPromotion	\$20 off any item with subtotal of \$50 or more	exclusive coupon
Line20PctOffExclusiveCouponPromotion	20% off any item with subtotal of \$50 or more	exclusive coupon
LineSaharaJacket5OffPromotion	\$5 off the Sahara Jacket item	automatic
LineSaharaJacketPromotion	50% off the Sahara Jacket item	automatic
LineAlpineParkaExclusivePromotion	50% off the Alpine Parka item	exclusive auto.

10.5 Calculating Promotions

The following filters are used to prequalify promotions before they are evaluated, to reduce the evaluation workload:

- Approval status – the promotion approval status is equal to Approved, and has not been Disabled.
- Date range – the promotion's from and to date range is valid.
- Catalog – the promotion book is associated with the same catalog as items in the user's cart.
- Benefits type – the promotion's benefits are of the same type, cart, or cart line. For example, when calculating cart line promotions, no cart promotions are evaluated.
- Items – the promotion's included items match items in the user's cart. The promotion's excluded items exclude the promotion from evaluation if matching any items in the user's cart.
- Cart exclusivity – an exclusive cart promotion excludes all other promotions in the cart.
- Cart line exclusivity – an exclusive cart line promotion excludes all other promotions in all of the cart lines within the cart.

The filters apply in a particular order. The following describes the order in which promotions are calculated:

- Promotions at the cart line level are applied before promotions at the cart level. Promotions (including automatic, coupon, exclusive) are calculated for each cart line, then promotions (including automatic, coupon, exclusive) are calculated for the cart.
- Automatic promotions are applied before coupon promotions, for example within each cart line, or within the cart.
- Multiple automatic promotions are applied in order by valid-from date (oldest first).
- Multiple coupons are applied in the order they were added to the cart.
- If multiple exclusive promotions apply to a particular cart line or cart, then the first one found is applied (found in order by valid-from date).

Calculation examples:

Example 1: Applicable promotions, listed in the order they are calculated:

- 1) Cart Line 1 automatic 10% off list price, valid from 2017-01-23
- 2) Cart Line 1 automatic 5% off list price, valid from 2017-01-27
- 3) Cart Line 2 automatic 15% off list price
- 4) Cart automatic free shipping for cart prices over \$100 promotion.

Results: 15% off Cart Line 1 list price
 15% off Cart Line 2 list price
 Shipping is free (if the cart's calculated sell price exceeds \$100)

Example 2: Applicable promotions, listed in the order they are calculated:

- 1) Cart Line 1 automatic 10% off list price, valid from 2017-01-23

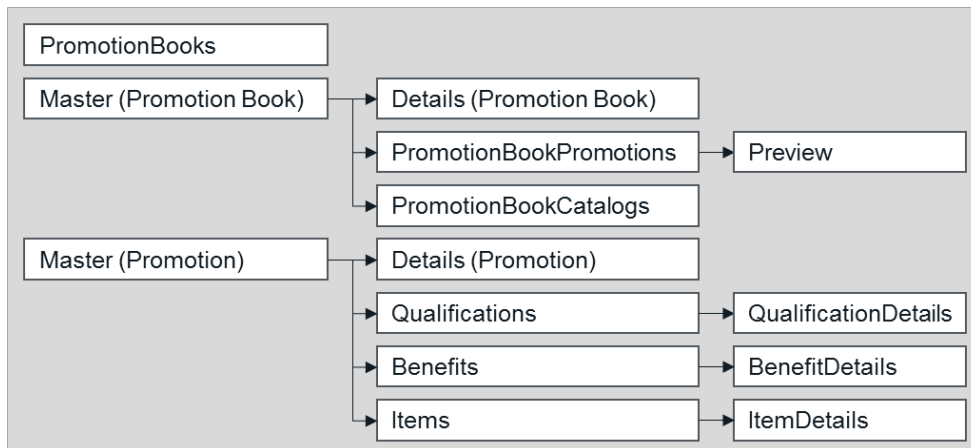
- 2) Cart Line 1 automatic 5% off list price, valid from 2017-01-27
- 3) Cart Line 2 automatic 15% off list price
- 4) Customer entered exclusive Cart Line 2 coupon for 20% off list price
- 5) Cart automatic free shipping for cart prices over \$100 promotion.

Results: 20% off Cart Line 2 list price

Shipping is free (if the cart's calculated sell price exceeds \$100)

10.6 Promotions Views

The following figure displays the hierarchy of promotions views:



The following table describes promotions views terminology, grouped by promotions entities and components:

Promotion Book Views	Description
PromotionBooks	A list of all the promotion books.
Master	Displays the promotion book information.
Details	Displays the promotion book details.
PromotionBookPromotions	A list of the promotions within the promotion book.
PromotionBookCatalogs	A list of the catalogs indicating if they are or are not associated with the promotion book.
Promotion Views	Description
Master	Displays the promotion information.
Details	Displays the promotion details.
Preview	Displays a subset of the promotion's details view.
Qualification Views	Description
Qualifications	A list of the promotion's qualifications.
QualificationDetails	Displays the promotion's qualification details.
Benefit Views	Description
Benefits	A list of the promotion's benefits.
BenefitDetails	Displays the promotion's benefit details.
Item Views	Description
Items	A list of items (for example, products or variants) the promotion applies to.
ItemDetails	Displays the promotion item's details.

10.7 Promotions Actions, Commands, Pipelines

The following table describes the promotions actions, commands, and pipelines pertaining to the Promotion Book entity:

Action/Command/Pipeline	Description
AddPromotionBook AddPromotionBookCommand AddPromotionBookPipeline	Adds a promotion book. Requires a unique promotion book name.
EditPromotionBook EditPromotionBookCommand EditPromotionBookPipeline	Edits a promotion book.
AssociateCatalog AssociateCatalogToBookCommand AssociateCatalogToBookPipeline	Associates a promotion book to a specific catalog.

Action/Command/Pipeline	Description
DisassociateCatalog DisassociateCatalogFromBookCommand DisassociateCatalogFromBookPipeline	Disassociates a promotion book from a specific catalog.
GetPromotionBookAssociatedCatalogs GetBookAssociatedCatalogsCommand GetBookAssociatedCatalogsPipeline	Retrieves all catalogs associated with a promotion book.

The following table describes the promotions actions, commands, and pipelines pertaining to the Promotion entity:

Action/Command/Pipeline	Description
AddPromotion AddPromotionCommand AddPromotionPipeline	Adds a promotion. Requires the parent promotion book name and a unique promotion name.
EditPromotion EditPromotionCommand EditPromotionPipeline	Edits a promotion. The promotion's approval status must not be Approved.
DuplicatePromotion DuplicatePromotionCommand DuplicatePromotionPipeline	Duplicates a promotion.
RequestPromotionApproval SetApprovalStatusCommand	Changes a promotion's approval status from Draft to ReadyForApproval.
ApprovePromotion SetApprovalStatusCommand	Changes a promotion's approval status from ReadyForApproval to Approve.
RejectPromotion SetApprovalStatusCommand	Changes a promotion's approval status from ReadyForApproval to Draft.
RetractPromotion SetApprovalStatusCommand	Changes a promotion's approval status from Approved back to Draft. Applicable only before the promotion is made fully active.
DisablePromotion DisablePromotionCommand DisablePromotionPipeline	Disables an approved promotion. Applicable when the promotion is approved and active.
DeletePromotion DeletePromotionCommand DeletePromotionPipeline	Deletes a promotion. The promotion's approval status must not be Approved.
EvaluatePromotionsQualificationsCommand EvaluatePromotionsQualificationsPipeline	These are used as part during the runtime of the promotions service.
DiscoverPromotionsPipeline	This is used as part during the runtime of the promotions service.
FilterQualifyingPromotionsPipeline	Searches for promotions in the system.

Note

ApplyPromotionsBenefitsCommand and corresponding pipeline are not used:

The following table describes the promotions actions, commands, and pipelines pertaining to the Qualification component:

Action/Command/Pipeline	Description
SelectQualification GetOperatorsCommand and GetConditionsCommand	Selects a qualification from all the qualifications available in the system. This action is implemented together with the AddQualification.
AddQualification AddQualificationCommand AddQualificationPipeline	Adds the selected qualification to a promotion. The promotion's approval status must be Draft.
EditQualification EditQualificationCommand EditQualificationPipeline	Edits a promotion's qualification. The promotion's approval status must be Draft.

Developer's Guide

Action/Command/Pipeline	Description
DeleteQualification DeleteQualificationCommand DeleteQualificationPipeline	Removes a qualification from a promotion. The promotion's approval status must be Draft. All qualifications can be deleted from a promotion only if the promotion has no benefits.

The following table describes the promotions actions, commands, and pipelines pertaining to the Benefit component:

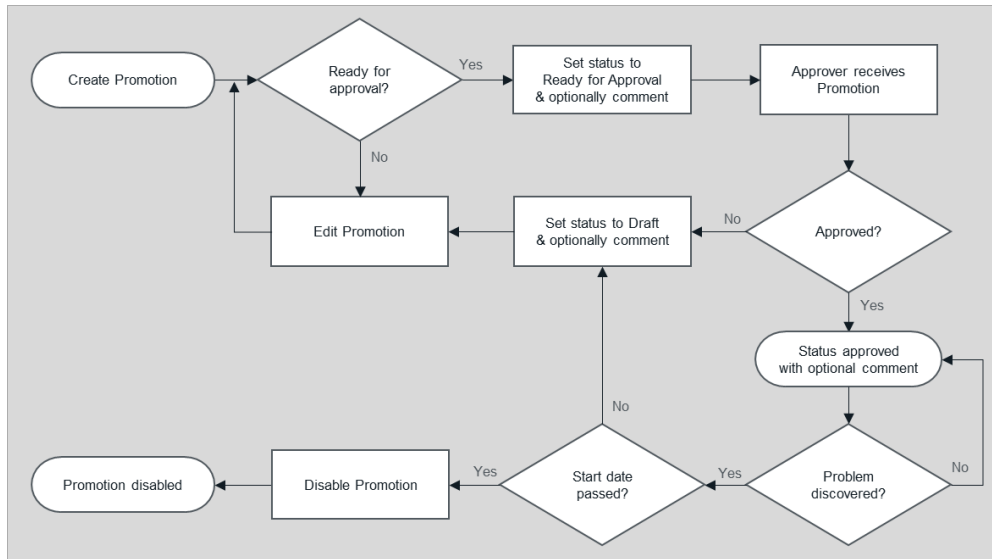
Action/Command/Pipeline	Description
SelectBenefit GetOperatorsCommand and GetActionsCommand	Selects a benefit from all the benefits available in the system.
AddBenefit AddBenefitCommand AddBenefitPipeline	Adds the selected benefit to a promotion. The promotion's approval status must be Draft. The promotion must have a qualification.
EditBenefit EditBenefitCommand AddBenefitPipeline	Edits a promotion's benefit. The promotion's approval status must be Draft.
DeleteBenefit DeleteBenefitCommand DeleteBenefitPipeline	Removes a benefit from a promotion. The promotion's approval status must be Draft.

The following table describes the promotions actions, commands, and pipelines pertaining to the Item component:

Action/Command/Pipeline	Description
AddItem AddPromotionItemCommand AddPromotionItemPipeline	Adds an Item to a promotion. An item must be unique within a promotion. The promotion's approval status must be Draft.
RemoveItem RemovePromotionItemCommand RemovePromotionItemPipeline	Removes an item from a promotion. The promotion's approval status must be Draft.

Promotions Approval Process

The following figure illustrates the promotions approval process:



10.8 Promotions Models

The following table lists the promotions models:

Model
AssociatedCatalogModel
PromotionBookAdded
PromotionAdded
PromotionChanged
QualificationAdded
BenefitAdded
PromotionItemAdded
PromotionItemModel

10.9 Promotions Policies

The following table describes the promotions policies:

Policy	Description
ExclusivePromotionPolicy	Identifies whether the promotion is exclusive. An exclusive promotion means that it is the only promotion that applies for a given cart line, or for a given cart. For example, a cart containing two cart lines could have up to three exclusive policies applied.
GlobalPromotionsPolicy	Provides the ability to specify some standard promotion policies.
KnownPromotionsActionsPolicy	Provides the ability to change the default promotions action names, those listed above.
KnownPromotionsListsPolicy	Provides the ability to change the default promotions list names
KnownPromotionsViewsPolicy	Provides the ability to change the default promotions view names.
PromotionBenefitsPolicy	A policy that is added to a promotion to define its benefits.
PromotionQualificationsPolicy	A policy that is added to a promotion to define its qualifications.

Chapter 11 Promotion Service – Coupons

As part of the Promotions service, the Coupons plugin provides the ability to generate, redeem, and track coupons. A coupon represents a customer-initiated request to qualify for a promotion. Types of coupons include:

- Public – multiuse coupons that can be used by anyone.
- Private – single-use coupons targeted at specific registered customers.

Coupons can apply at various layers:

- Cart Line – for example, money-off or a percentage-off an item or variant in a cart line.
- Cart – for example, money-off or a percentage-off the entire cart.
- Customer – for example, access to a benefit that applies across carts, for example, services, entitlements, a trial period.

Other capabilities of coupons include:

- Customer adds a coupon to a cart, by typing in the coupon code.
- Customer removes a coupon from the cart.
- System includes the coupon in its calculation of promotion benefits based on entered coupon.
- Validating the coupon when it is added to the cart.
- Presenting to the customer in the cart if the coupon will be applied, or if it will not be used (for example, is not qualified).
- Tracking the number of times a valid coupon is used in completed orders.

Business users in the Business Tools can:

- View coupons used in an order.
- View coupons used by a customer.
- View coupons allocated to a customer.
- Assign a coupon to a registered customer.
- Generate coupons linked to a promotion, or a batch of specified quantity each with a unique coupon code.
- Manage generated coupons, including allocating/unallocating coupons.

11.1 Coupons Concepts

The following table describes the hierarchy of coupon entities and components:

Entity	Components
Coupon	
CouponUsage	CartCoupons
PrivateCouponGroup	
PriceCard	

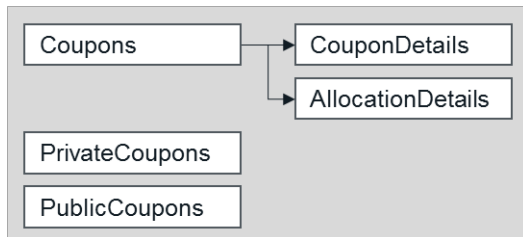
The following table describes general coupons terminology:

Coupon Term	Description
Coupon	A customer-initiated request to qualify for a promotion.
CouponUsage	Defines where a coupon is used.
PrivateCouponGroup	A function to keep track of private coupons associated with a promotion, identifying whether they are allocated or not.

Coupon Term	Description
PriceCard	A mechanism to apply common pricing across multiple SellableItems without repeating the same pricing definition in each one
CartCoupons	A component normally placed into a cart where any coupons that have been added to the cart are stored.

11.2 Coupons Views

The following figure displays the hierarchy of coupon views:



The following table describes the coupons views terminology:

Coupons Views	Description
Coupons	A list of all available coupons.
PrivateCoupons	A list of available private coupons.
PublicCoupons	A list of available public coupons.
CouponDetails	The details of a specified coupon.
AllocationDetails	The details of where a specified coupon is allocated.

11.3 Coupons Actions, Commands, Pipelines

The following table describes the coupons actions, commands, and pipelines:

Action/Command/Pipeline	Description
AddCoupon AddCouponCommand AddCouponToCartPipeline	<p>Adds a coupon to a cart using a coupon code. It has the following parameters:</p> <ul style="list-style-type: none"> cartId – the identifier of the cart (string). couponCode – the identifier of the coupon (string). <p>To add a coupon to a cart using a coupon code, this action, command and pipeline:</p> <ol style="list-style-type: none"> Validates the coupon by loading the coupon based on the coupon code. <ul style="list-style-type: none"> Coupon code length <= GlobalCouponsPolicy.MaxCouponLength Coupon code not already in cart Coupon with that coupon code actually exists Retrieves the CartCouponsComponent from the cart. Checks if this coupon code is already added. Creates a new CartCoupon object. Populates: IsValid = true, promotion = coupon.Promotion. Adds new CartCoupon object to CartCouponsComponent list property.
AddPrivateCoupon AddPrivateCouponCommand AddPrivateCouponPipeline	Generates a batch of private coupons, adding those coupons to a list of unallocated coupons associated to the promotion. Uses the

Action/Command/Pipeline	Description
	<p>prefix and suffix combined with a generated string to form a complete unique coupon code.</p> <p>It has the following parameters:</p> <ul style="list-style-type: none"> <code>PromotionId</code> – the identifier of a promotion the coupon qualifies for (string). <code>Prefix</code> – a prefix to apply to the generated codes (string). <code>Suffix</code> – a Suffix to apply to the generated codes (string). <code>Total</code> – the number of private coupons to generate (integer).
AddPublicCoupon AddPublicCouponCommand AddPublicCouponPipeline	<p>Creates a public coupon to be associated with a promotion.</p> <p>It has the following parameters:</p> <ul style="list-style-type: none"> <code>PromotionId</code> – the identifier of a promotion the coupon qualifies for (string). <code>CouponCode</code> - the identifier of the coupon code (string).
NewAllocation NewCouponAllocationCommand NewCouponAllocationPipeline	<p>Allocates a batch of coupons from an private coupon group associated with the promotion. This moves the list of coupons from the unallocated coupons list to the allocated coupons list. This returns a list of coupon codes that have been allocated.</p> <p>It has the following parameters:</p> <ul style="list-style-type: none"> <code>PromotionId</code> – the identifier of a promotion to allocate from (string). <code>PrivateCouponGroupId</code> - the identifier of the PrivateCouponGroup to allocate from (string). <code>Total</code> - the number of private coupons to allocate (integer).
RemoveCoupon RemoveCouponCommand RemoveCouponFromCartPipeline	<p>Removes a coupon from the cart.</p> <p>It has the following parameters:</p> <ul style="list-style-type: none"> <code>cartId</code> – the identifier of the cart (string). <code>couponCode</code> – the identifier of the coupon (string).

11.4 Coupons Models

The following table lists the coupons models:

Model
CartCoupon
PrivateCouponGroupAdded
PrivateCouponList
PublicCouponAdded

11.5 Coupons Policies

The following table describes the coupons policies:

Policy	Description
GlobalCouponsPolicy	<p>Defines coupon policies that apply globally:</p> <ul style="list-style-type: none"> <code>GeneratedCouponCodeLength</code> – defines number of characters that the coupon code shall have. <code>MaxCouponCodeLength</code> – maximum length of a coupon before it gets rejected by validation; default = 30 (integer). <code>MaxCouponPrefixLength</code> – maximum length of the coupon's prefix; default = 10 (integer). <code>MaxCouponSuffixLength</code> – maximum length of the coupon's suffix; default = 10 (integer).

Policy	Description
	<ul style="list-style-type: none"> • <code>MaxNumberOfPrivateCoupons</code> – maximum number of coupons to generate at one time; default = 1000 (integer). • <code>MinPrivateCouponTotal</code> – minimum number of private coupons; default = 1 (integer). • <code>MinAllocationCount</code> – minimum number of coupons to allocate; default = 1 (integer).
<code>CartCouponsPolicy</code>	Defines a policy for the <code>CartCoupons</code> component: <ul style="list-style-type: none"> • <code>MaxCouponsInCart</code> – maximum # of coupons allowed in the cart; default = 5 (integer).
<code>KnownCouponActionsPolicy</code>	Provides the ability to change the default coupon action names.
<code>KnownCouponViewsPolicy</code>	Provides the ability to change the default coupon view names.
<code>KnownCouponsListsPolicy</code>	Provides the ability to change the default coupon list names.

Chapter 12 Entitlements Service

The Entitlements service is provided by the **Sitecore.Commerce.Plugin.Entitlements** plugin.

An entitlement is an artifact representing a trackable unit of ownership or license with the ability to track unit quantities as they change due to customer or ambient activity. This allows a separation of concerns between a right to access a digital product or the tracking of activity on a customer's behalf.

Example entitlements includes:

- Gift card
- Warranty
- Installation service
- Digital product
- Loyalty membership

Entitlement capabilities include:

- The ability to have a separate provisioning process for an entitlement when an order is placed.
- The ability to separately track a customer's current entitlements from the original order that caused it to be provisioned.
- The ability to track a quantity on an entitlement that is meaningful to the entitlement, such as the current value of a gift card or the current number of loyalty points accumulated, and so on.
- The ability to remove an entitlement from a customer.

12.1 Entitlement Concepts

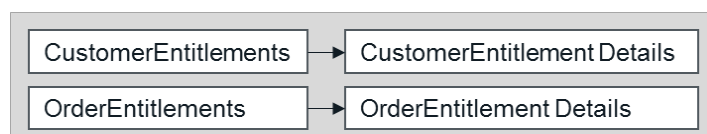
Entitlements represent digital rights. These are usually acquired through a purchase. For example, buying an online movie does not actually get you the movie. It only gives you the rights to view a movie. Often entitlements are only available for a limited time or other restrictions apply. For example, you can rent a movie online but you can only view that movie in certain countries, or for a certain period of time, or in certain formats. Entitlements provide the ability to store those rights and the policies that govern access in a generic way that can be extended to meet a solution developer's needs.

Provisioning entitlements: When an order containing digital items is released, the `ReleasedOrdersMinionPipeline` calls the `ProvisionEntitlementsPipeline`. This creates the order's entitlements and links them to the order and the order's customer (if the customer is authenticated).

Deleting entitlements: Entitlements are not actually deleted from the system. They are logically deleted by adding a `DeletedEntitlementComponent`, which indicates it has been deleted and no longer included in the views that return entitlements for orders or customers.

12.2 Entitlement Views

The following figure displays the hierarchy of entitlement views:



The following table describes entitlement views:

View	Description
CustomerEntitlements	Displays a customer's entitlements.
CustomerEntitlementsDetails	Displays the details of a customer's entitlements.
OrderEntitlements	Displays an order's entitlements.
OrderEntitlementDetails	Displays the details of an order's entitlements.

The following table lists entity blocks:

Entity Views
GetCustomerEntitlementsViewBlock
GetOrderEntitlementsViewBlock
PopulateCustomerEntitlementsViewActionsBlock
PopulateOrderEntitlementsViewActionsBlock
Entity Actions
DoActionDeleteEntitlementBlock

12.3 Entitlements Actions, Commands, Pipelines

The following table describes the entitlements actions, commands, and pipelines:

Action/Command/Pipeline	Description
Get]Api/Entitlements FindEntitiesInListCommand FindEntitiesInListPipeline	Returns all the entitlements in the system. Note This is computationally expensive and should be used with caution. Parameters: none
Get]Api/Entitlements(id) FindEntityCommand FindEntityPipeline	Returns a specific entitlement based on its identifier. It has the following parameter: <ul style="list-style-type: none"> Id – the identifier of the entitlement.
DeleteEntitlement DeleteEntitlementCommand DeleteEntitlementPipeline	Deletes an entitlement.

12.4 Entitlements Policies

The following table describes the entitlements policies:

Policy	Description
KnownEntitlementsActionsPolicy	Provides the ability to change the default entitlement action names.
KnownEntitlementsViewsPolicy	Provides the ability to change the default entitlement view names.

Chapter 13 Customer Service

The Customer service is provided by the **Sitecore.Commerce.Plugin.Customers** plugin.

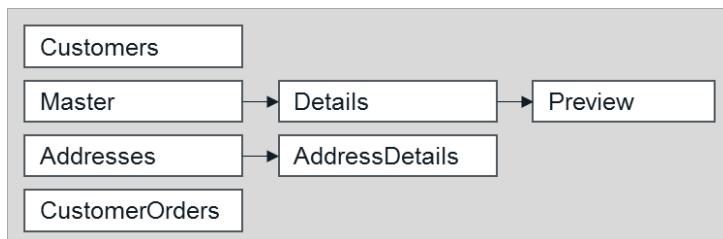
The Sitecore Commerce Customer Service provides integration with external profile systems. In this release, the out-of-box product is delivered with integration with the Commerce Server's profile system.

The following describes the hierarchy of customer entities and components:

Entity	Component
Customer	CustomerDetails Address

13.1 Customer Views

The following figure describes the hierarchy of customer views:



The following table describes the customer views:

View	Description
Customers	Lists customers.
Master	Displays customer information.
Details	Displays detailed customer information.
Preview	Displays a subset of the customer details information.
Addresses	Displays the customer addresses.
AddressDetails	Displays detailed address information.
CustomerOrders	Displays the orders associated with a customer.

The following table lists the entity blocks:

Entity Views
GetCustomerAddressDetailsViewForSelectBlock
GetCustomerAddressDetailsViewBlock
GetCustomerAddressesViewBlock
GetCustomerDetailsViewBlock
GetCustomerOrdersViewBlock
GetCustomerPreviewViewBlock
PopulateAddressDetailsViewActionsBlock
PopulateAddresseseViewActionsBlock
PopulateCustomersViewActionsBlock
PopulateEntityViewActionsBlock
PopulateEntityViewActionsMasterBlock
Entity Actions

DoActionAddressBlock
DoActionAddCustomerBlock
DoActionEditAddressBlock
DoActionEditCustomerBlock
DoActionGetCountryRegionsBlocks
DoActionRemoveAddressBlock
DoActionRemoveCustomerBlock

13.2 Customer Actions, Commands, Pipelines

The following table describes the customer actions, commands, and pipelines:

Action/Command/Pipeline	Description
GetCustomer GetCustomerCommand GetCustomerPipeline	Retrieves a specific customer.
GetCustomerAddress GetCustomerAddressCommand GetCustomerAddressPipeline	Retrieves the address of a specified customer.
AddCustomer CreateCustomerCommand CreateCustomerPipeline	Creates a new customer record.
AddAddress CreateCustomerAddressCommand CreateCustomerAddressPipeline	Creates a new address record for a customer.
EditCustomer UpdateCustomerDetailsCommand UpdateCustomerDetailsPipeline	Edits a customer record.
EditAddress UpdateCustomerAddressCommand UpdateCustomerAddressPipeline	Edits a customer address record.
RemoveCustomer DeleteCustomerCommand DeleteCustomerPipeline	Deletes a customer record.
RemoveAddress DeleteCustomerAddressCommand DeleteCustomerAddressPipeline	Deletes a customer address record.

13.3 Customer Models

The following table describes the customer models:

Model	Description
CustomerAdded	A return model providing the identifier of a customer that was added.
CustomerAddressAdded	A return model providing the identifiers of a customer and an address that were added.

13.4 Customer Policies

The following table describes the customer policies:

Policy	Description
KnownCustomerActionsPolicy	Provides the ability to change the default customer action names.
KnownCustomersListsPolicy	Provides the ability to change the default customer list names.

Policy	Description
KnownCustomerViewsPolicy	Provides the ability to change the default customer view names.
ProfilePropertiesMappingPolicy	<p>Provides mappings between a Commerce Server profile and a Commerce Engine customer.</p> <p>Defaults:</p> <ul style="list-style-type: none"> • <code>first_name</code> = <code>GeneralInfo.first_name</code> • <code>last_name</code> = <code>GeneralInfo.last_name</code> • <code>email_address</code> = <code>GeneralInfo.email_address</code> • <code>user_security_password</code> = <code>GeneralInfo.user_security_password</code> • <code>account_status</code> = <code>AccountInfo.account_status</code> • <code>language</code> = <code>GeneralInfo.language</code>
PartyProfilePropertiesMappingPolicy	<p>Provides mappings between a Commerce Server profile address and a Commerce Engine Party Model.</p> <p>Defaults:</p> <ul style="list-style-type: none"> • <code>address_name</code> = <code>AddressName</code> • <code>first_name</code> = <code>FirstName</code> • <code>last_name</code> = <code>LastName</code> • <code>country_name</code> = <code>Country</code> • <code>country_code</code> = <code>CountryCode</code> • <code>region_name</code> = <code>State</code> • <code>region_code</code> = <code>StateCode</code> • <code>city</code> = <code>City</code> • <code>address_line1</code> = <code>Address1</code> • <code>address_line2</code> = <code>Address2</code> • <code>postal_code</code> = <code>ZipPostalCode</code> • <code>tel_number</code> = <code>PhoneNumber</code>
ProfilePropertiesPolicy	<p>Provides mappings of standard profile properties between Commerce Server and the Commerce Engine.</p> <p>Defaults:</p> <ul style="list-style-type: none"> • <code>this.AddressType</code> = <code>Address</code> • <code>this.UserObjectType</code> = <code>UserObject</code> • <code>this.GeneralInfoPropertyGroup</code> = <code>GeneralInfo</code> • <code>this.AccountInfoPropertyGroup</code> = <code>AccountInfo</code> • <code>this.UserIdProperty</code> = <code>GeneralInfo.user_id</code> • <code>this.EmailAddressProperty</code> = <code>GeneralInfo.email_address</code> • <code>this.AddressListProperty</code> = <code>GeneralInfo.address_list</code> • <code>this.ExternalIdProperty</code> = <code>GeneralInfo.ExternalId</code> • <code>this.PasswordProperty</code> = <code>GeneralInfo.user_security_password</code> • <code>this.AddressIdProperty</code> = <code>GeneralInfo.address_id</code> • <code>this.AddressId</code> = <code>address_id</code> • <code>this.Country</code> = <code>Country</code> • <code>this.CountryCode</code> = <code>CountryCode</code> • <code>this.State</code> = <code>State</code> • <code>this.StateCode</code> = <code>StateCode</code> • <code>this.City</code> = <code>City</code> • <code>this.AccountNumber</code> = <code>AccountNumber</code> • <code>this.Email</code> = <code>Email</code> • <code>this.Password</code> = <code>"Password"</code>

Policy	Description
	<ul style="list-style-type: none"> • <code>this.Value = Value</code> • <code>this.UserTypeSiteTerm = UserType</code> • <code>this.AccountStatusSiteTerm = AccountStatus</code> • <code>this.Languages = Languages</code>
<code>SitecoreUserTermsPolicy</code>	<p>Provides the location in Sitecore XP where UserTerms are stored.</p> <p>Defaults:</p> <ul style="list-style-type: none"> • <code>this.AccountStatusPath = /sitecore/Commerce/Commerce Control Panel/Commerce Engine Settings/Commerce Terms/CS User Site Terms/Account Status</code> • <code>this.UserTypePath = /sitecore/Commerce/Commerce Control Panel/Commerce Engine Settings/Commerce Terms/CS User Site Terms/User Type</code> • <code>this.AllLanguagesPath = /sitecore/Commerce/Commerce Control Panel/Shared Settings/Language Sets/All Languages</code>
<code>ProfilesSqlPolicy</code>	Provides connectivity information to the Commerce Server Profile System.

Chapter 14 Catalog Service

The Catalog service is provided by the **Sitecore.Commerce.Plugin.Catalog** plugin and the **Sitecore.Commerce.Plugin.Catalog.Cs** plugin.

In the current release, the Catalog Service is managed by Commerce Server. Therefore, the role of the Commerce Engine is focused on managing and displaying catalog information (for example, products and carts), rather than directly managing catalogs.

Commerce Engine catalog service capabilities include:

- Retrieving a standardized catalog or list of catalogs.
- Providing a basic SellableItem concept for carts and orders.
- Calculating bulk pricing for a catalog item.
- Managing and storing list pricing for a SellableItem, in multiple currencies.

The two catalog plugins apply as follows:

- **Sitecore.Commerce.Plugin.Catalog** – provides general catalog functionality commands and pipelines.
- **Sitecore.Commerce.Plugin.Catalog.Cs** – provides integration with the Commerce Server Catalog system.

14.1 Catalog Actions, Commands, Pipelines

The following table describe the catalog actions, commands, and pipelines:

Action/Command/Pipeline	Description
Catalogs GetCatalogsCommand GetCatalogsPipeline	Retrieves all catalogs defined in the system. Retrieves the list of catalogs from Commerce Server and maps them to the generic object. Parameters: none
Catalogs (id) GetCatalogCommand GetCatalogPipeline	Retrieves a specific Catalog defined in the system. It has the following parameter: <ul style="list-style-type: none"> • Id – identifier of the catalog (string)
SellableItems FindEntitiesInListCommand FindEntitiesInListPipeline	Retrieves all the SellableItems in the system. Note This is computationally expensive when you have many SellableItems and must not be used. Parameters: none
SellableItems (id) GetSellableItemCommand GetSellableItemPipeline	Retrieves a specified SellableItem. It has the following parameter: <ul style="list-style-type: none"> • Id – identifier of the SellableItem (string).
GetBulkPrices GetBulkPricesCommand GetSellableItemPipeline	Provides bulk pricing capabilities. Retrieves a string of sellable item identifiers and returns a bulk pricing model for each item. Returns pricing on a particular variant if a variant identifier is supplied, otherwise returns pricing on all variants. Parameter: <ul style="list-style-type: none"> • itemIds – a delimited list of item identifiers; contains a Catalog, ProductId, and VariantId to fully describe a particular item (string).

Action/Command/Pipeline	Description
UpdateListPrices UpdateListPricesCommand UpdateListPricesPipeline	Retrieves and stores list prices for a SellableItem, for example, when updating the list pricing in the Merchandising Manager. It can contain pricing in multiple currencies. It has the following parameters: <ul style="list-style-type: none"> • <code>itemId</code> – identifier of the SellableItem (string) • <code>Prices</code> – prices for different currencies (delimited string)
RemoveListPrices RemoveListPricesCommand RemoveListPricesPipeline	Removes specified list prices from a SellableItem. Parameters: <ul style="list-style-type: none"> • <code>itemId</code> – identifier of the SellableItem (string) • <code>Prices</code> – prices for different currencies (delimited string)

14.2 Catalog Models

The following table describes the catalog models:

Model	Description
SellableItemPricing	A return model providing bulk pricing. Properties: <ul style="list-style-type: none"> • <code>Name</code> – name of SellableItem (string). • <code>ItemId</code> – identifier of the SellableItem (string). • <code>ListPrice</code> – the SellableItem's list price (string). • <code>SellPrice</code> – the SellableItem's sell price (string). • <code>Variations</code> – a list of pricing variations (list<string>).
VariationPricing	A return model providing bulk pricing at the variation level. Properties: <ul style="list-style-type: none"> • <code>Name</code> – name of SellableItem (string). • <code>ItemId</code> – identifier of the SellableItem (string). • <code>ListPrice</code> – the SellableItem's list price (string). • <code>SellPrice</code> – the SellableItem's sell price (string).

Note

There are no catalog policies.

Chapter 15 Availability Service

The Availability service is provided by the **Sitecore.Commerce.Plugin.Availability** plugin.

Availability is the generalized notion of conveying whether an item is available to be purchased. It is intended to apply in a simplistic, small-shop scenario where there is no real inventory tracking. Availability could be extended as required to address more comprehensive scenarios, for example, integration with an inventory system and concepts like back orders and preorders.

For integration with Commerce Server, the availability service calls a stored procedure in Commerce Server to reduce the quantity of the inventory stock keeping unit (sku).

Availability capabilities include:

- Supporting a basic products' availability separate from the concept of inventory.
- Tracking availability for non-physical items such as digital products.
- Being applicable where only basic product availability is needed, instead of a full inventory system.

Availability can be defined and returned as part of a Sellable Item. Or it can be injected as a component into a CartLineItem.

Note

There are no Availability Service views or models.

15.1 Availability Commands and Pipelines

The Availability Service does not have externally exposed Service APIs. Availability is supported by extending existing pipelines.

The following table describes the availability commands and pipelines:

Command/Pipeline	Description
UpdateItemAvailabilityCommand UpdateItemAvailabilityPipeline	Provides the ability to update availability on an item. This is performed during the sale of a product, through integration to reduce the inventory count in an external inventory system. It has the following parameters: <ul style="list-style-type: none"> • <code>catalogName</code> – name of the catalog of an item (string). • <code>productId</code> – the product identifier (string). • <code>variantId</code> – the product variant identifier (string). • <code>deltaQuantity</code> – the quantity to reduce the inventory by, or increase it in the event of a canceled order (decimal).

15.2 Availability Policies

The following table describes the availability policies:

Policy	Description
AvailabilityAlwaysPolicy	Indicates that inventory checks on a SellableItem do not need to be performed. This is the case for digital items that do not have a specific inventory.
DigitalItemTagsPolicy	Allows the specification of a list of tags that cause an item to be considered a digital item. Digital items are automatically considered

Policy	Description
	<p>always available and have no inventory. This is a simple way of flagging an item in the Merchandising Manager as digital.</p> <p>The following default tags are provided: entitlement, service, installation, subscription, digital subscription, warranty, onlinetraining, onlinelearning, giftcards</p> <p>You can use an environment policy to override these tags.</p>
GlobalAvailabilityPolicy	<p>Provides basic availability with a single property (AvailabilityExpires), indicating how long availability applies, in seconds.</p> <p>This allows more efficient functioning because the system does not need to check availability with an external system.</p>

Chapter 16 Inventory Service

The Inventory service is provided by the **Sitecore.Commerce.Plugin.Inventory** and the **Sitecore.Commerce.Plugin.Inventory.Cs** plugin.

The Sitecore Commerce inventory solution integrates with Commerce Server's inventory system. Inventory is provided by the Merchandising Manager business tool.

Inventory capabilities include retrieving:

- A Commerce Server inventory catalog.
- A Commerce Server inventory SKU to populate a generic inventory item.
- A specific Commerce Server inventory SKU.

Note

There are no Inventory Service views.

16.1 Inventory Commands and Pipelines

The inventory plugins do not currently have any externally exposed inventory actions. Functionality in the plugin integrates with existing pipelines to implement inventory. Commands are called during the execution of other pipelines.

The following table describes the inventory commands and pipelines:

Command/Pipeline	Description
<code>GetInventoryCatalogCommand</code>	Retrieves an inventory catalog. Parameters: none To retrieve an inventory catalog, the command: <ul style="list-style-type: none"> • Retrieves the Commerce Server inventory catalog. • Translates it to a generic inventory catalog.
<code>GetInventoryItemCommand</code>	Retrieves an inventory item. Parameters: none To retrieve an inventory item, the command: <ul style="list-style-type: none"> • Retrieves the Commerce Server inventory item. • Translates it to a generic inventory item.
<code>GetInventorySkuCommand</code>	Retrieves an inventory SKU. Parameters: none To retrieve an inventory SKU, the command: <ul style="list-style-type: none"> • Retrieves the Commerce Server inventory SKU. • Translates it to a generic inventory SKU.

16.2 Inventory Policies

The following table describes the inventory policies:

Policy	Description
<code>GlobalInventoryPolicy</code>	Defines global inventory settings, for example for stock level, and adjusting stock levels with orders.
<code>LoggingPolicy</code>	Defines system logging settings for the inventory service.

Chapter 17 Payment Service

The Payment service is provided by the **Sitecore.Commerce.Plugin.Payment** plugin.

The Payment plugin provides basic commands, pipelines, and policies to implement a payment collection through integration with a third party payment service.

Payment capabilities include:

- Resolving a collection of payment options for a cart. Payment options allow the selection of a type of payment (credit card, PayPal, and so on.).
- Resolving a collection of payment options for an individual line in a cart.
- Resolving a collection of payment methods for a cart, based on a previously selected cart payment option. Some payment options allow further selection of a payment method, for example, selecting the shipping (ground, next day, and so on).
- Resolving a collection of payment methods for an individual line in a cart, based on a previously selected cart line payment option.
- Sample integration with a third-party payment provider (BrainTree), integrated into a sample solution and provided as source code in the SDK.

17.1 Payment Concepts

Sitecore Commerce provides out-of-box support for collecting payments from gift cards, but does not directly support collecting and storing payment vehicles like credit cards. As such, this enables Sitecore Commerce to contribute to a PCI compliant commerce solution without itself mediating sensitive financial information.

The Payment plugin supports the concept of *federated payments*, which leverages a third-party payment provider to collect payment information using an iFrame on the website. The solution receives and stores payment tokens and masked information, and not the full set of credit-card data. The reference sample provided with the product includes an integration with BrainTree for this functionality. The source code for this integration is included in the Sitecore Commerce SDK.

BrainTree offers sandboxing support without requiring financial information, so a solution developer can easily set up and run the integration in a demo environment. The solution developer must set up a sandbox account with BrainTree to be able to demonstrate collecting credit cards.

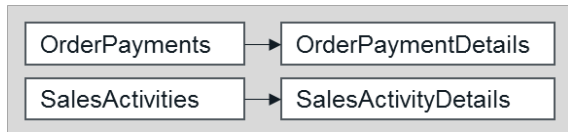
It is not required to use BrainTree in a production Sitecore Commerce implementation. Solution developers can remove the BrainTree integration and replace with their own custom integration, or an integration offered as a solution by a Sitecore partner.

The Reference Solution contains configuration for the BrainTree implementation. This section should be populated with the solution developer's specific values provided by BrainTree after registering. For example, a sample configuration:

```
{
  "$type": "Plugin.Sample.Payments.Braintree.BraintreeClientPolicy,
Plugin.Sample.Payments.Braintree",
  "Environment": "sandbox",
  "MerchantId": "[YourMerchantId]",
  "PublicKey": "[YourPublicKey]",
  "PrivateKey": "[YourPrivateKey]",
  "ConnectTimeout": 120000
}
```

17.2 Payment Views

The following figure displays the hierarchy of payment views:



The following table describes the payment views:

Payment Views	Description
OrderPayments	A list of all the order payments.
OrderPaymentDetails	Displays the order payment details.
SalesActivities	A list of all the sales activities.
SalesActivityDetails	Displays the order payment details.

The following table lists the entity blocks:

Entity Views
GetOrderPaymentDetailsViewBlock
GetOrderPaymentDetailsViewsForSelectBlock
GetOrderPaymentsViewBlock
GetOrderSalesActivitiesEntityViewBlock
PopulateOrderPaymentsViewActionsBlock
Entity Actions
DoActionAddPaymentBlock
DoActionGetCountryRegionsBlock
DoActionGetCountryRegionsFederatedBlock
DoActionRefundPaymentBlock
DoActionSelectPaymentOptionBlock
DoActionVoidPaymentBlock
GetFederatedPaymentOptionBlock
GetPaymentFromViewBlock
SelectFederatedPaymentOptionBlock
SelectPaymentOptionBlock

17.3 Payment Actions, Commands, Pipelines

The following table describes the payment actions, commands, and pipelines:

Action/Command/Pipeline	Description
Api/GetClientToken GetClientTokenCommand GetClientTokenPipeline	Retrieves a client token from the integrated third-party payment provider. This token allows the storefront payment iFrame to communicate directly with the payment provider. Parameters: none
[Put]Api/AddFederatedPayment AddPaymentsCommand AddPaymentsPipeline	Adds a federated payment to a cart. It has the following parameters: <ul style="list-style-type: none"> • <code>cartId</code> – the identifier of the cart. • <code>payment</code> – a federated payment component.
[Delete]Api/RemovePayment RemovePaymentsCommand RemovePaymentsPipeline	Removes a federated payment from a cart. It has the following parameters: <ul style="list-style-type: none"> • <code>cartId</code> – the identifier of the cart. • <code>paymentId</code> – the identifier of the payment to remove.
[Get]Api/GetCartPaymentMethods GetPaymentMethodsCommand	Retrieves a list of available payment methods based on the payment type selected and the items in the Cart

Action/Command/Pipeline	Description
GetCartPaymentMethodsPipeline	<p>Parameters:</p> <ul style="list-style-type: none"> • <code>cartId</code> – the identifier of the cart. • <code>paymentType</code> – a string representing a selected payment type <p>To retrieve a list of the available payment methods, the command:</p> <ul style="list-style-type: none"> • Retrieves the payment methods. • Filters the payment methods based on the payment type. • Returns the available payment methods.

Note

The following are not implemented:

[Get] Api / PaymentMethods
 [Get] Api / PaymentOptions
 [Get] Api / GetCartPaymentOptions

17.4 Payment Policies

The following table describes the payment policies:

Policy	Description
KnownPaymentsActionsPolicy	Provides the ability to change the default payment action names.
KnownPaymentsViewsPolicy	Provides the ability to change the default payment views names.
KnownViewsPolicy	Provides the ability to change the default view names for the Payment service, specifically the <code>PaymentOptions</code> view.

Chapter 18 Fulfillment Service

The Fulfillment service is provided by the **Sitecore.Commerce.Plugin.Fulfillment** plugin.

The Fulfillment plugin offers a basic fulfillment experience. It is expected that solution developers will extend this experience with an additional customized experience. APIs and pipelines are available to allow customization by extending or replacing existing blocks in the pipeline.

Fulfillment capabilities include:

- Resolving a collection of fulfillment options that are available for a cart., for example, physical, electronic, split. Options are filtered based on the type(s) of items in the cart.
- Resolving a collection of fulfillment options for an individual line in a cart.
- Resolving a collection of fulfillment methods that are available for a cart, based on a previously selected fulfillment option, for example, for a physical fulfillment you can select ground, next day, and so on.
- Resolving a collection of fulfillment methods for an individual line in a cart, based on a previously selected fulfillment option.
- Basic calculation of fulfillment fees based on fulfillment policies. A simple calculation model is provided, which focuses on testing calculations. Solution developers will usually integrate with a specific shipping integration service.

18.1 Fulfillment Concepts

The calculation of fulfillment charges is configured by an environment policy called **GlobalPhysicalFulfillmentPolicy**. This policy controls the configuration based on the selected fulfillment method and currency. This policy is set up in the environment configuration JSON files in the `wwwroot/data/Environments` directory. The policy can be modified there. For example, a sample configuration:

```
{
  "$type":
  "Sitecore.Commerce.Plugin.Fulfillment.GlobalPhysicalFulfillmentPolicy,
  Sitecore.Commerce.Plugin.Fulfillment",
  "MaxShippingWeight": 50.0,
  "MeasurementUnits": "Inches",
  "WeightUnits": "Lbs",
  "DefaultCartFulfillmentFees": {
    "$type": "System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Money,
    Sitecore.Commerce.Core]], mscorlib",
    "$values": [
      {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "USD",
        "Amount": 10.0
      },
      {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "CAD",
        "Amount": 12.0
      }
    ]
  },
  "DefaultCartFulfillmentFee": {
    "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
    "CurrencyCode": "USD",
    "Amount": 3.0
  }
}
```

```
    },
    "DefaultItemFulfillmentFees": {
      "$type": "System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Money,
Sitecore.Commerce.Core]], mscorlib",
      "$values": [
        {
          "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
          "CurrencyCode": "USD",
          "Amount": 2.0
        },
        {
          "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
          "CurrencyCode": "CAD",
          "Amount": 3.0
        }
      ]
    },
    "DefaultItemFulfillmentFee": {
      "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
      "CurrencyCode": "USD",
      "Amount": 3.0
    },
    "FulfillmentFees": {
      "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment]], mscorlib",
      "$values": [
        {
          "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
          "Fee": {
            "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
            "CurrencyCode": "USD",
            "Amount": 15.0
          },
          "Name": "Ground",
          "Policies": {
            "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
            "$values": []
          }
        },
        {
          "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
          "Fee": {
            "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
            "CurrencyCode": "USD",
            "Amount": 2.0
          },
          "Name": "Standard",
          "Policies": {
            "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
            "$values": []
          }
        },
        {
          "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
          "Fee": {
            "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
            "CurrencyCode": "USD",
            "Amount": 5.0
          },
          "Name": "Next Day Air",
          "Policies": {
            "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
            "$values": []
          }
        }
      ]
    }
  }
}
```

```
    },
    {
      "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
      "Fee": {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "USD",
        "Amount": 10.0
      },
      "Name": "Standard Overnight",
      "Policies": {
        "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
        "$values": []
      }
    },
    {
      "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
      "Fee": {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "CAD",
        "Amount": 15.0
      },
      "Name": "Ground",
      "Policies": {
        "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
        "$values": []
      }
    },
    {
      "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
      "Fee": {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "CAD",
        "Amount": 2.0
      },
      "Name": "Standard",
      "Policies": {
        "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
        "$values": []
      }
    },
    {
      "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
      "Fee": {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "CAD",
        "Amount": 5.0
      },
      "Name": "Next Day Air",
      "Policies": {
        "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
        "$values": []
      }
    },
    {
      "$type": "Sitecore.Commerce.Plugin.Fulfillment.FulfillmentFee,
Sitecore.Commerce.Plugin.Fulfillment",
      "Fee": {
        "$type": "Sitecore.Commerce.Core.Money, Sitecore.Commerce.Core",
        "CurrencyCode": "CAD",
        "Amount": 10.0
      },
      "Name": "Standard Overnight",
      "Policies": {
```

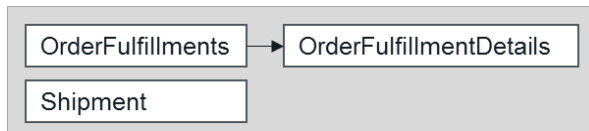
```

        "$type":
"System.Collections.Generic.List`1[[Sitecore.Commerce.Core.Policy, Sitecore.Commerce.Core]],
mscorlib",
        "$values": []
    }
}
]
}
}

```

18.2 Fulfillment Views

The following figure displays the hierarchy of fulfillment views:



The following table describes the fulfillment views terminology:

Payment Views	Description
OrderFulfillments	A list of all the order fulfillments.
OrderFulfillmentDetails	Displays the order fulfillment details.
Shipments	A list of all the shipments.

The following table lists the entity blocks:

Entity Views
GetOrderFulfillmentDetailsViewBlock
GetOrderFulfillmentDetailsViewForSelectBlock
GetOrderFulfillmentsViewBlock
GetOrderShipmentsEntityViewBlock
PopulateOrderFulfillmentViewActionsBlock
Entity Actions
DoActionGetCountryRegionsBlock
DoActionGetFulfillmentMethodsBlock
DoActionSelectFulfillmentOptionBlock
DoActionSetFulfilmentBlock
GetDigitalFulfillmentFromViewBlock
GetFulfillmentFromViewBlock
GetPhysicalFulfillmentFromViewBlock
SelectDigitalFulfillmentOptionBlock
SelectFulfillmentOptionBlock
SelectShipToMeFulfillmentOptionBlock
SelectSplitFulfillmentOptionBlock

18.3 Fulfillment Actions, Commands, Pipelines

The following describes the fulfillment actions, commands, and pipelines:

Action/Command/Pipeline	Description
Api/GetCartFulfillmentOptions GetFulfillmentOptionsCommand GetCartFulfillmentOptionsPipeline	Retrieves a list of fulfillment options, with the ability to filter based on items in the Cart. It has the following parameter: <ul style="list-style-type: none"> cartId – the identifier of the cart The command:

Action/Command/Pipeline	Description
	<ul style="list-style-type: none"> Retrieves the available fulfillment options. Filters based on items in the cart.
Api/GetCartLineFulfillmentOptions GetFulfillmentOptionsCommand GetCartLineFulfillmentOptionsPipeline	Retrieves a list of fulfillment options for a specific cart line. It has the following parameters: <ul style="list-style-type: none"> <code>cartId</code> – the identifier of the cart. <code>cartLineId</code> – the identifier of the cart line. The command: <ul style="list-style-type: none"> Retrieves the available fulfillment options. Filters based on items in the cart line.
Api/GetFulfillmentMethods GetFulfillmentMethodsCommand GetFulfillmentMethodsPipeline	Retrieves a list of all available fulfillment methods. Parameters: none
Api/GetCartFulfillmentMethods GetFulfillmentMethodsCommand GetCartFulfillmentMethodsPipeline	Retrieves a list of all available fulfillment methods for a cart, based on the populated FulfillmentComponent and the items in the cart. It has the following parameters: <ul style="list-style-type: none"> <code>cartId</code> – the identifier of the cart. <code>fulfillment</code> – a PhysicalFulfillmentComponent (ShippingParty populated). The command: <ul style="list-style-type: none"> Loads the cart. Retrieves available fulfillment methods. Filters methods based on items in the cart. Returns methods.
Api/GetCartLineFulfillmentMethods GetFulfillmentMethodsCommand GetCartLineFulfillmentMethodsPipeline	Retrieves a list of all available fulfillment methods for a cart line, based on the populated FulfillmentComponent and the item in the cart line. It has the following parameters: <ul style="list-style-type: none"> <code>cartId</code> – the identifier of the cart <code>fulfillment</code> – a PhysicalFulfillmentComponent (LineId populated with a specific Line Id, ShippingParty populated) The command: <ul style="list-style-type: none"> Retrieves available fulfillment methods. Filters methods based on item in the cart line. Returns methods.
Api/SetCartFulfillment SetCartFulfillmentCommand SetCartFulfillmentPipeline	Sets a cart to a specific FulfillmentComponent. It has the following parameters: <ul style="list-style-type: none"> <code>cartId</code> – the identifier of the cart <code>fulfillment</code> – a FulfillmentComponent The command: <ul style="list-style-type: none"> Validates the FulfillmentComponent. Sets the component into the cart. Calculates the cart. Persists the cart. Adds cart totals to return the models collection.
Api/SetCartLineFulfillment SetCartLinesFulfillmentCommand SetCartLinesFulfillmentPipeline	Sets a cart line to a specific FulfillmentComponent. It has the following parameters: <ul style="list-style-type: none"> <code>cartId</code> – the identifier of the cart. <code>cartLineId</code> – the identifier of the cart line. <code>fulfillment</code> – a FulfillmentComponent. The command: <ul style="list-style-type: none"> Validates the FulfillmentComponent.

Action/Command/Pipeline	Description
	<ul style="list-style-type: none"> Sets the component into the cart line. Calculates the cart line. Persists the cart line. Adds the cart line totals to return the models collection.
Api/Shipment FindEntitiesInListCommand FindEntitiesInListPipeline	Returns all shipments in the shipments list. This could be a very long list so do not use for routine processing. Instead, use the GetList functionality. Parameters: none The command: <ul style="list-style-type: none"> Uses <code>FindEntitiesInList</code> to retrieve all of the items in the Shipments list.
Api/Shipment(id) FindEntityCommand FindEntityPipeline	Returns a specific shipment based on its identifier. It has the following parameter: <ul style="list-style-type: none"> <code>Id</code> – the identifier of the shipment. The command: <ul style="list-style-type: none"> Retrieves a specific shipment entity using the <code>FindEntityPipeline</code>.

Note:

The following is not implemented: `Api/FulfillmentOptions`.

18.4 Fulfillment Models

The following fulfillment model is available:

Model
FulfillmentFee

18.5 Fulfillment Policies

The following describes the fulfillment policies:

Policy	Description
<code>GlobalPhysicalFulfillmentPolicy</code>	Provides policies associated with shipping items, including maximum shipping weight, measurements, and fulfillment fees.
<code>KnownFulfillmentActionsPolicy</code>	Provides the ability to change the default fulfillment actions.
<code>KnownFulfillmentListsPolicy</code>	Provides the ability to change the default fulfillment lists.
<code>KnownFulfillmentViewsPolicy</code>	Provides the ability to change the default fulfillment views.

Note:

`DeliveryInStorePickupPolicy` and `ShippingPolicy` are not used.

Chapter 19 Shops Service

The Shops service is provided by the **Sitecore.Commerce.Plugin.Shops** plugin.

The Shops plugin provides the ability to specify environmental rules and other details for multiple shops or at an individual shop level.

The Shops Service capabilities include:

- Administering multiple shops.
- Administering both online and physical shops.
- Assigning multiple shops to an environment, or assigning each shop to an exclusive environment.

19.1 Shops Concepts

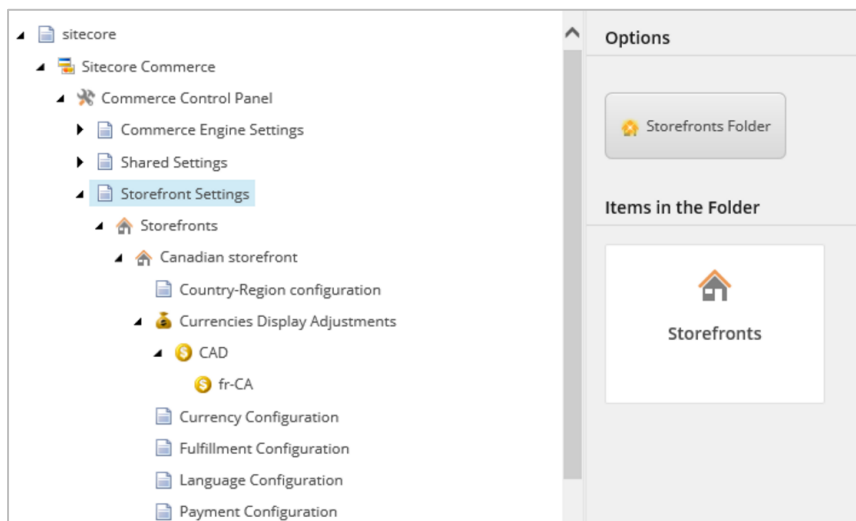
A Sitecore Commerce shop as it applies to the Commerce Engine represents a collection of policies and configurations that are applied to one or more online storefronts or physical brick-and-mortar stores. Configurations include, for example: branding experience, geographic location, and franchise.

A shop can specify a particular environment that provides a collection of policies including location and method of entity storage.

A Sitecore Commerce shop has been developed as a hybrid between:

- Details specified in Sitecore Commerce as part of bootstrap, and
- Details specified in the Sitecore Control Panel.

The following figure shows an example structure of the shops configuration of the Commerce Control Panel:



For more information on accessing and using the Commerce Control Panel for configuring shops, refer to the topics on doc.sitecore.net, specifically the Storefront configuration settings.

19.2 Accessing a Shop from Rules or from Pipeline Blocks

You might want to leverage information provided by a shop entity during the execution of business functionality. In this case, the shop is validated and loaded into the objects collection during the `ValidateContextPipeline`, which is run on every call into Sitecore Commerce. Sitecore Commerce validates that the storefront passed in as a header is a valid and that the requested currency is valid. The shop can then be referenced as a cached object in any pipeline block or by rules in the rules engine.

19.3 Shops Components

Shops components include:

- `OnlineShopComponent` – aspects of an online shop. The `ServiceUrl` is the URL of the shop.
- `ShopFinancialsComponents` – financial aspects that may be used for financial reporting including:
 - `LegalEntity`
 - `DefaultCustomer`
 - `BusinessUnit`
 - `CostCenter`
 - `Department`

The shops components represent an extension point on whether other policies/components can be inserted by various plugins, by listening for the `GetShop` pipeline.

Validation is performed on every call before processing a request in the `OnActionExecuting` event. Any errors cause an `HttpRequest` with the localized message for each of the following:

- `InvalidStorefront` – the storefront requested does not exist.
- `InvalidStorefrontCurrency` – the currency requested is not valid for this storefront.

A `ServiceAPI` is provided for retrieving shop information, as follows:

- `/api/Shops` – retrieves a list of shops.
- `/api/Shops('Storefront')?&expand=Components` – retrieves a single shop instance.

19.4 Shops Models

The following table lists shops models:

Model
<code>ShopFulfillmentMethod</code>
<code>ShopFulfillmentOption</code>
<code>ShopPaymentMethod</code>
<code>ShopPaymentOption</code>

Chapter 20 Guided Tours

This chapter provides examples of guided tours for extending Sitecore Commerce.

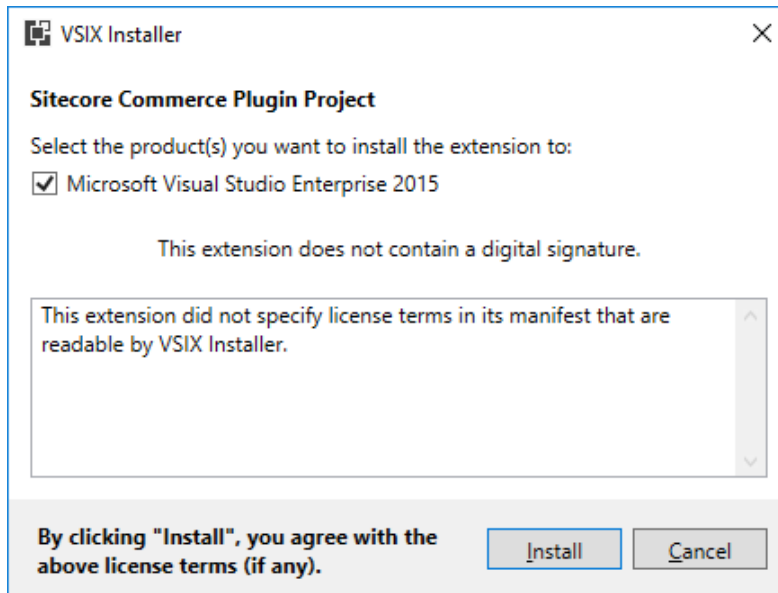
20.1 Get the Customer.Sample.Solution Up and Running

To begin your plugin development, this document assumes that you have followed the [Sitecore Commerce 8.2.1 Deployment Guide](#), and have a working deployment with the standard sample environments and data.

Sitecore Commerce provides a Visual Studio extension to make it easier to create new plugins.

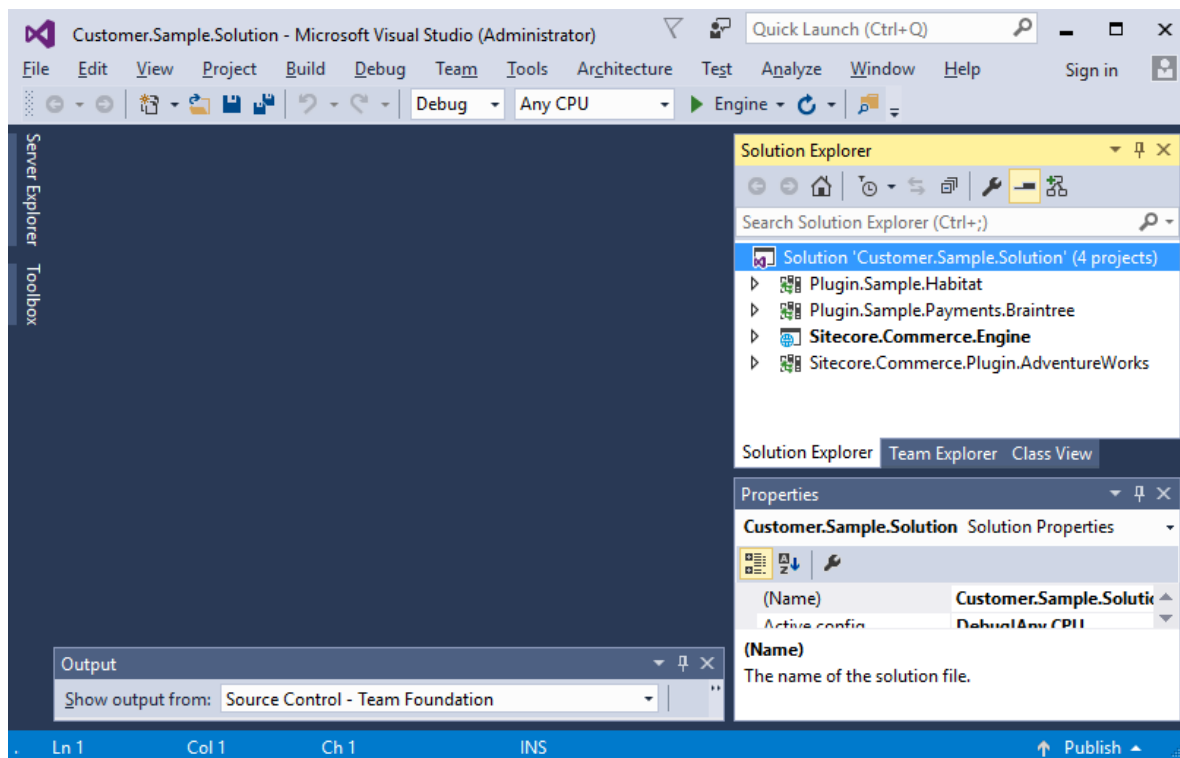
To get the Customer.Sample.Solution up and running:

1. On the machine that has Visual Studio 2015 (Update 3) installed, open the SDK.
2. To install the Visual Studio Extension, double-click `SitecorePluginTemplate.vsix`:



3. In Visual Studio, open `Customer.Sample.Solution.sln`. This is the Visual Studio solution that you will use to develop plugins for the Sitecore Commerce Engine, and to build and deploy a new instance of the Engine.
4. You can rename the solution to conform to the naming standards of your company.
5. Press **F5** to run the Engine in Visual Studio in debug mode. As you develop a plugin, you can interactively debug your plugin using Visual Studio:

Sitecore Commerce 8.2.1



The `Customer.Sample.Solution` contains the Engine project (an ASP.NET Core host), and several sample plugins. As you develop your plugin, you can run your extended solution through a set of sample orders to test the plugin functionality.

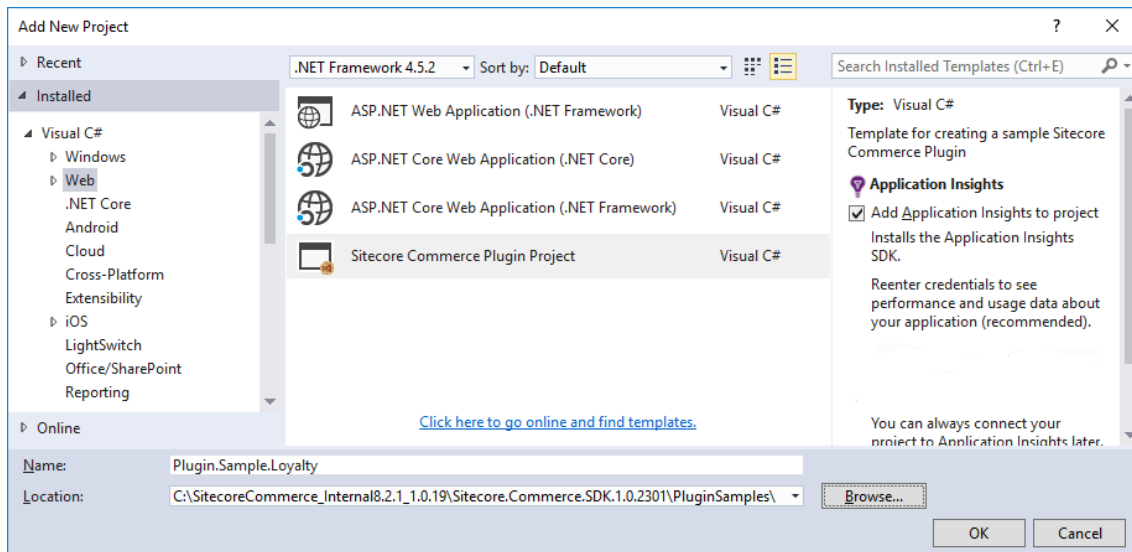
The `Sitecore.Commerce.Sample.Console.sln` is a simple console tool that provides samples, and allows the end-to-end demonstration and functional tests of standard scenarios. It can be loaded in a second Visual Studio instance, and can be enhanced by either removing unneeded scenarios, or adding your own to test specific scenarios. This is useful for quick and iterative development of extensibility and integration. You can build and test back-end scenarios without running Sitecore.

Note

Before you do any extensibility, to make sure that you have a working system, run the `Customer.Sample.Solution` to host the Engine. Also, to ensure that all existing functionality can be used, run the console tool. This is important to make sure that you are starting from a known working state, and do not confuse issues that are related to set up with issues that are caused by your extension.

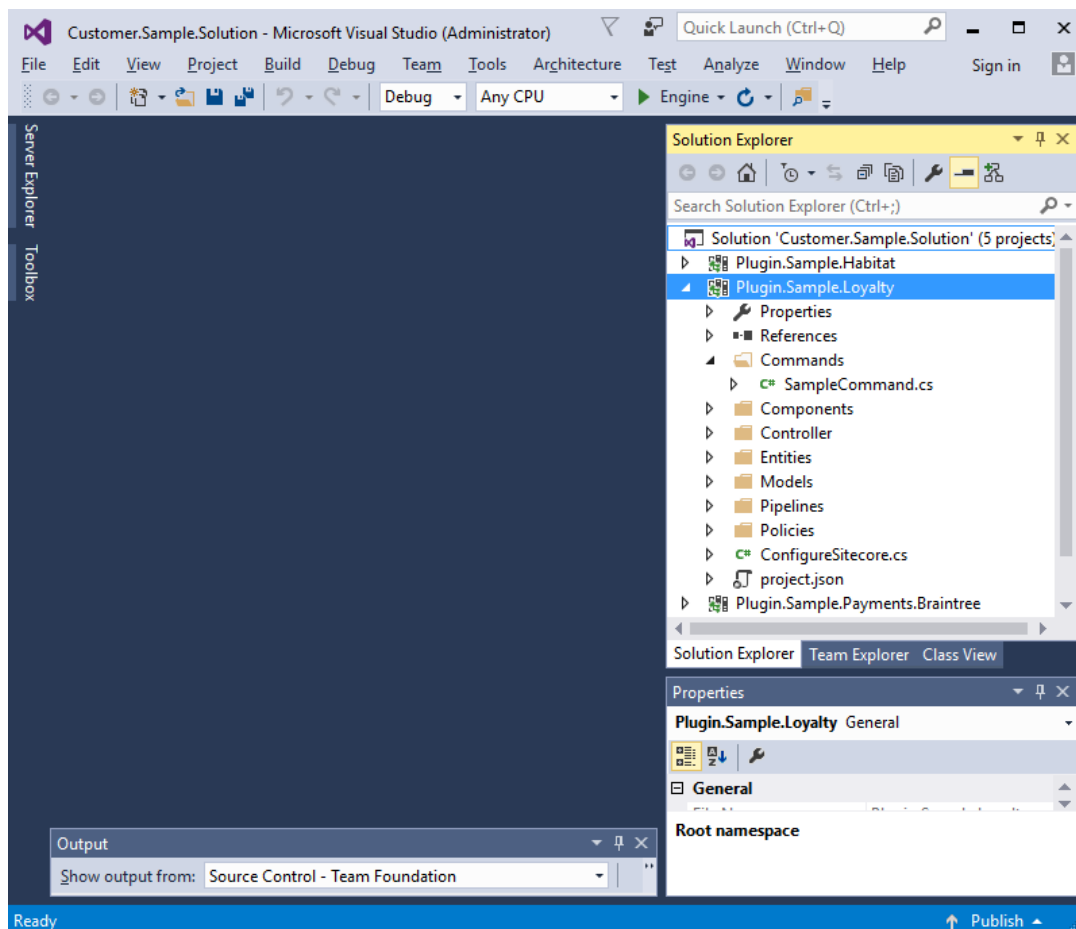
20.2 Creating Your First Plugin

To add your new plugin, the Visual Studio extension has added a new template:



To create your first plugin:

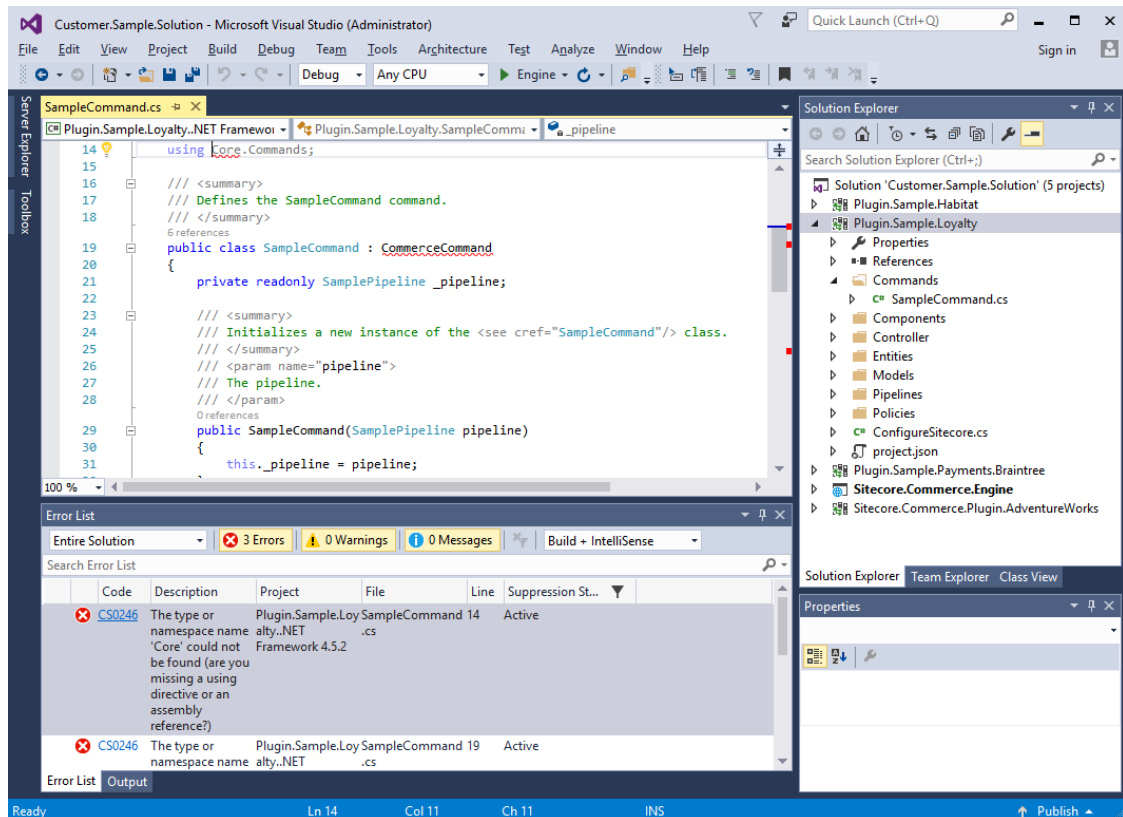
1. Give your plugin a namespace and name.
2. Select the folder to add your plugin to. In the following example, the plugin is added to the *PluginSamples* folder in SDK. Do not try to add your plugin to a folder outside SDK, because the file paths can conflict with proper builds:



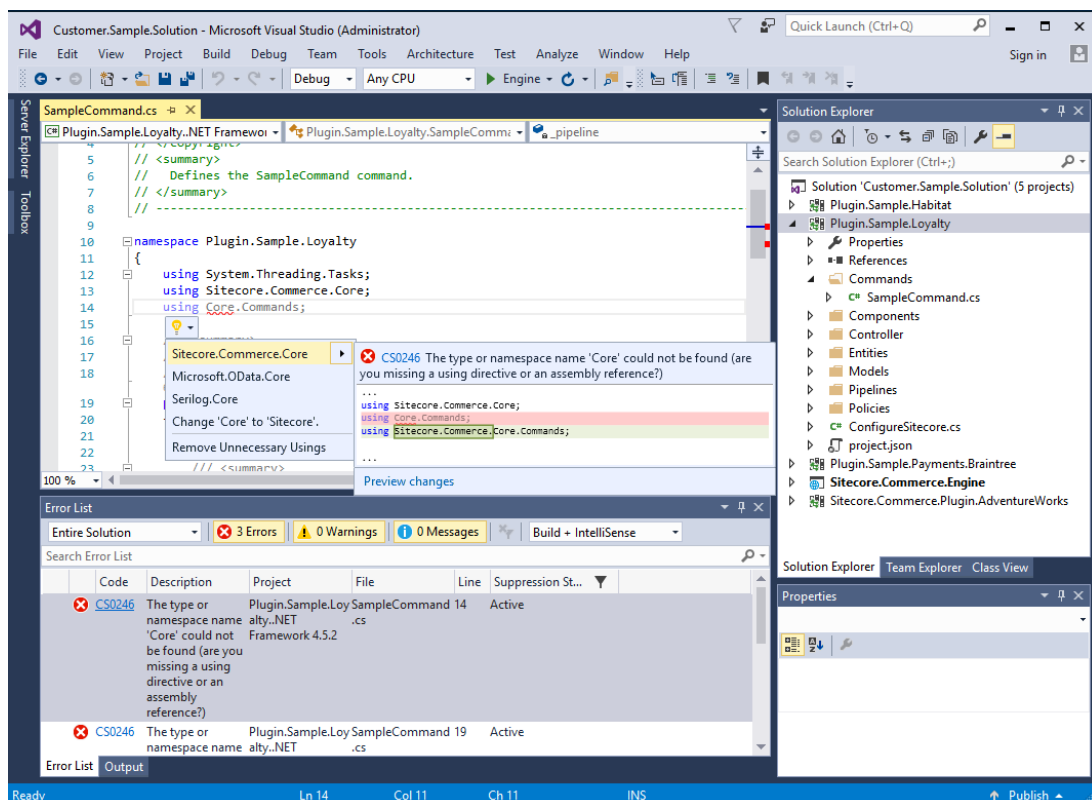
Sitecore Commerce 8.2.1

3. To construct your plugin and add it to your Visual Studio project with a sample Command/Pipeline, click **OK**.

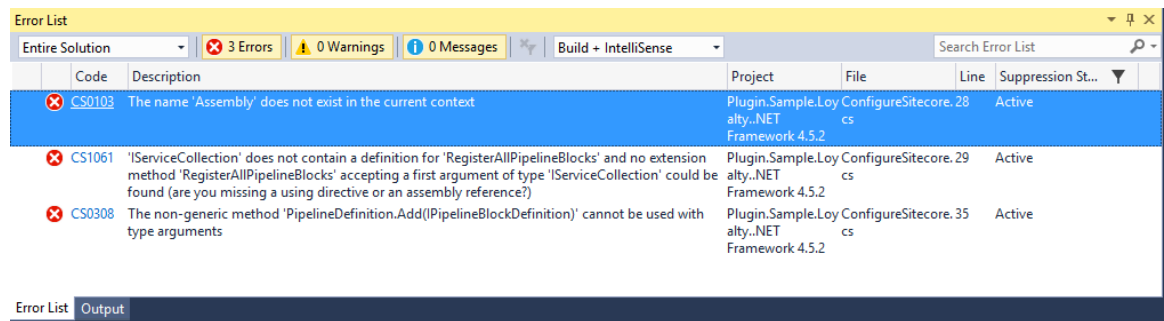
You receive three errors that are simple fix up errors:



Accept the suggestion to fix the path up to `Sitecore.Commerce.Core`:



Once you have fixed the minor namespace error, there are three other errors:



The screenshot shows the Visual Studio Error List window with the following data:

Code	Description	Project	File	Line	Suppression St...
CS0103	The name 'Assembly' does not exist in the current context	Plugin.Sample.Loyalty.NET Framework 4.5.2	ConfigureSitecore.cs	28	Active
CS1061	'IServiceCollection' does not contain a definition for 'RegisterAllPipelineBlocks' and no extension method 'RegisterAllPipelineBlocks' accepting a first argument of type 'IServiceCollection' could be found (are you missing a using directive or an assembly reference?)	Plugin.Sample.Loyalty.NET Framework 4.5.2	ConfigureSitecore.cs	29	Active
CS0308	The non-generic method 'PipelineDefinition.Add(PipelineBlockDefinition)' cannot be used with type arguments	Plugin.Sample.Loyalty.NET Framework 4.5.2	ConfigureSitecore.cs	35	Active

4. Follow the suggested fixes for the first two errors. For the third error, fix it by adding the following using to the `ConfigureSitecore` class:

```
using Sitecore.Framework.Pipelines.DefinitionExtensions;
```

After these minor fixes, your Visual Studio project will build successfully.

The new plugin has an end-to-end sample named `Sample` that shows an extension from the controller level all the way through running a pipeline. In many cases, you will not want to create a new pipeline, but instead want to extend an existing pipeline to add new processing blocks. You can delete these samples, but we recommend that you keep the samples until you are sure you no longer need them. You can copy and paste from the sample to create new items, such as components, because the sample item contains the necessary structure. Then you can change the naming to your preferred patterns.

Once the plugin builds, you must add it to the `Sitecore.Commerce.Engine` project as a reference so that the plugin will be pulled in during a build/publish:

1. In the `Sitecore.Commerce.Engine` project, open the `project.json` file, look for the `dependencies` list, and then add the Loyalty project:

```
"Plugin.Sample.Loyalty": "0.0.1"
```

When the file is saved, it will resolve the dependency and load it when the application is launched.

2. Once you build the project, press **F5** to run the solution in debug mode as you did previously, but now it will pull in your new plugin.

20.3 Mapping Additional Properties from a Commerce Server Catalog

It is often necessary as part of extending the solution, to add additional properties to the Commerce Server catalog, either at the product or at a variation level. These properties are used as part of business functionality.

Scenario: To support a loyalty system, you need to support a new property called `LoyaltyPoints` to all Commerce Server products. You want to map it so that it populates into the Sitecore Engine `SellableItem` when it is loaded.

To add the property to Commerce Server:

1. Open the legacy Windows application Commerce Server Catalog and Inventory Schema Manager.
2. Create a new property definition `LoyaltyPoints` as a number and assign it to all product types.
3. Set the default value to 100, and display it as a base property.

4. To augment a `SellableItem`, you need a block that will be processed as part of the `GetSellableItem` pipeline. To determine where you want to insert the block, look at the current pipeline configuration for the `GetSellableItem` pipeline. You can do this by looking in the `wwwroot/logs` directory of the solution for the latest file that starts with `NodeConfiguration`. This file is automatically generated whenever the solution is run, and outputs the most recent Pipeline configuration. If you search for `GetSellableItem`, you find the following pipeline:

```
-----
Sitecore.Commerce.Plugin.Catalog
IGetSellableItemPipeline (Sitecore.Commerce.Plugin.Catalog.ProductArgument =>
Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
Plugin.Catalog.GetSellableItemInitializeBlock
(Sitecore.Commerce.Plugin.Catalog.ProductArgument =>
Sitecore.Commerce.Plugin.Catalog.ProductArgument)
```

```
-----
Plugin.Catalog.Cs.TranslateProductBlock
(Sitecore.Commerce.Plugin.Catalog.ProductArgument =>
Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
Plugin.Catalog.Cs.TranslateImageryBlock (Sitecore.Commerce.Plugin.Catalog.SellableItem
=> Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
Plugin.Catalog.EnsureSellableItemPoliciesBlock
(Sitecore.Commerce.Plugin.Catalog.SellableItem =>
Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
Plugin.Availability.EnsureSellableItemAvailabilityPoliciesBlock
(Sitecore.Commerce.Plugin.Catalog.SellableItem =>
Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
Plugin.Catalog.ICalculateSellableItemPricesPipeline
(Sitecore.Commerce.Plugin.Catalog.SellableItem =>
Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
Plugin.Availability.IPopulateItemAvailabilityPipeline
(Sitecore.Commerce.Plugin.Catalog.SellableItem =>
Sitecore.Commerce.Plugin.Catalog.SellableItem)
-----
```

5. All of the current translation is performed in the `TranslateProductBlock`. To add additional translations, add a block after the `TranslateProductBlock` to map in additional properties. The output of the `TranslateProductBlock` is a `SellableItem`, and the next block is expecting the `SellableItem`, so you can insert a block that accepts a `SellableItem` as a parameter and returns the same `SellableItem` with the new extended properties.
6. Open the *Pipelines/Blocks* folder, copy `SampleBlock`, and rename it `SellableItemLoyaltyBlock`. Open the block and change all the places that say `SampleBlock` to `SellableItemLoyaltyBlock`.
7. To reference a `SellableItem`, add `Sitecore.Commerce.Plugin.Catalog` to the dependencies list in the `project.json` of the new plugin:

```
"Sitecore.Commerce.Plugin.Catalog": "1.0.2301",
"Sitecore.Commerce.Plugin.Catalog.Cs": "1.0.2301"
```

When `project.json` is saved again, it will resolve the dependency.

8. In the new block, add the using `Sitecore.Commerce.Plugin.Catalog` reference. Change the input/output parameter types to `SellableItem`. The `SellableItem` resolves, and the main `run` method can be changed to look like the following:

```
/// <summary>
/// The execute.
/// </summary>
/// <param name="arg">
/// The SampleArgument argument.
/// </param>
/// <param name="context">
/// The context.
/// </param>
/// <returns>
/// The <see cref="SampleEntity"/>.
/// </returns>
public override Task<SellableItem> Run(SellableItem arg,
CommercePipelineExecutionContext context)
{
    Condition.Requires(arg).NotNull("The argument cannot be null");
    //var result = this.pipeline.Run(arg, context).Result;
    return Task.FromResult(arg);
}
```

9. Now that the new block is able to be built, you must add it to the pipeline in the `ConfigureSitecore.cs` class by adding the following:

```
.ConfigurePipeline<IGetSellableItemPipeline>(<
    configure =>
    {
```

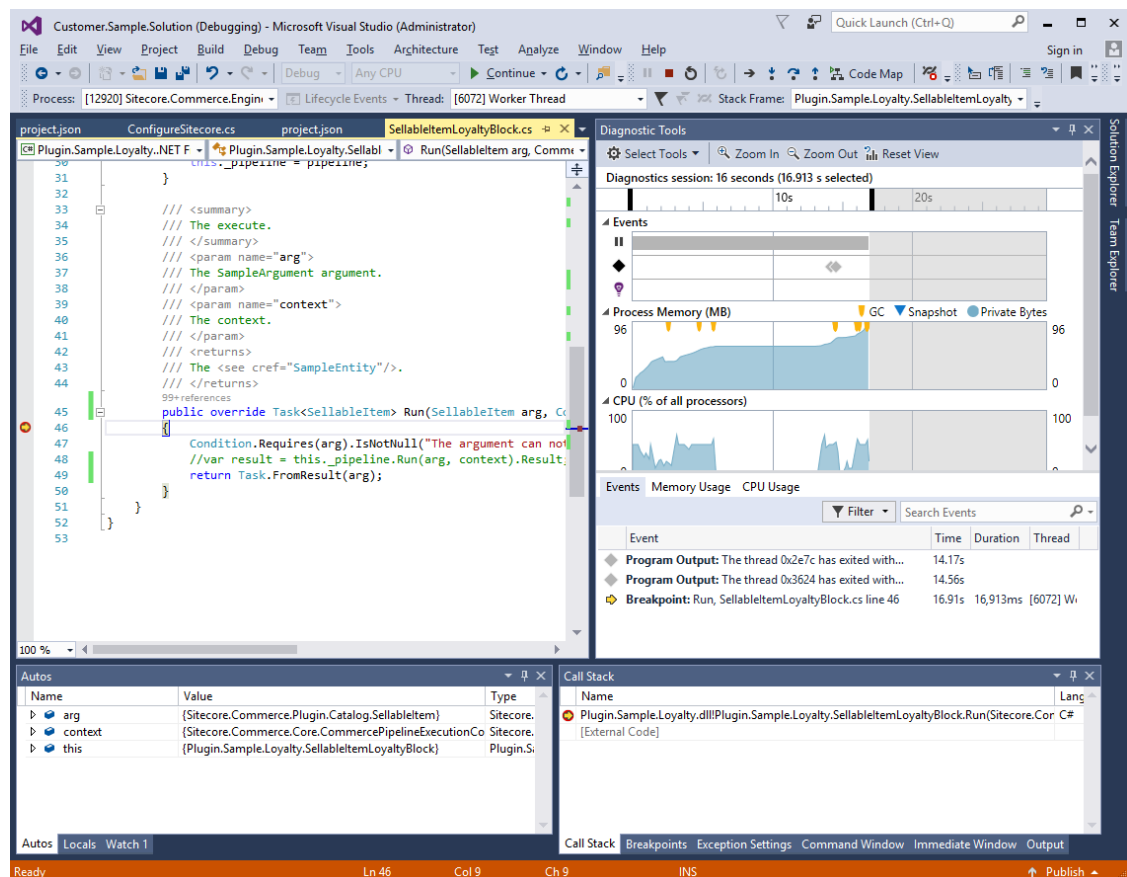
Sitecore Commerce 8.2.1

```
configure.Add<SellableItemLoyaltyBlock>().After<TranslateProductBlock>();  
})
```

This states that the pipeline should run the `SellableItemLoyaltyBlock` after the `TranslateProduct` block.

10. Before adding any logic, make sure that the block you added is going to be hit. To do this, add a break point at the beginning of the `Run` method, and then press **F5** to run the solution with the Engine in debug mode.

The simplest way to test this is with Postman. Load Postman and go to the `CatalogApiSamples/SellableItems` API, and then select "Get SellableItem with a variation". Send the command with no changes. If everything is correct, the break point will be hit, and you know the block will execute:



Adding Logic

Once you know that the block is wired up correctly, you can add the logic.

When you did the original mapping, you retrieved a Commerce Server Product. The object was cached in the `context.CommerceContext.Objects` collection so it would not be retrieved multiple times. As you execute a pipeline, objects are cached so that later blocks can retrieve them from the cache, instead of reloading them from the original source.

To add logic:

1. To gain access to the `Product` block, you must find it in the `Objects` collection. For example:

```
var product = context.CommerceContext.GetObjects<Product>()  
    .FirstOrDefault(p => p.ProductId.Equals(sellableItem.FriendlyId,  
StringComparison.OrdinalIgnoreCase));
```

2. Accept all of the suggestions to automatically add the "usings".

Now you have a local variable of the Commerce Server Product `SellableItem`, which simplifies mapping properties. Before you map any additional properties, you must define a new component to put the extended properties in.

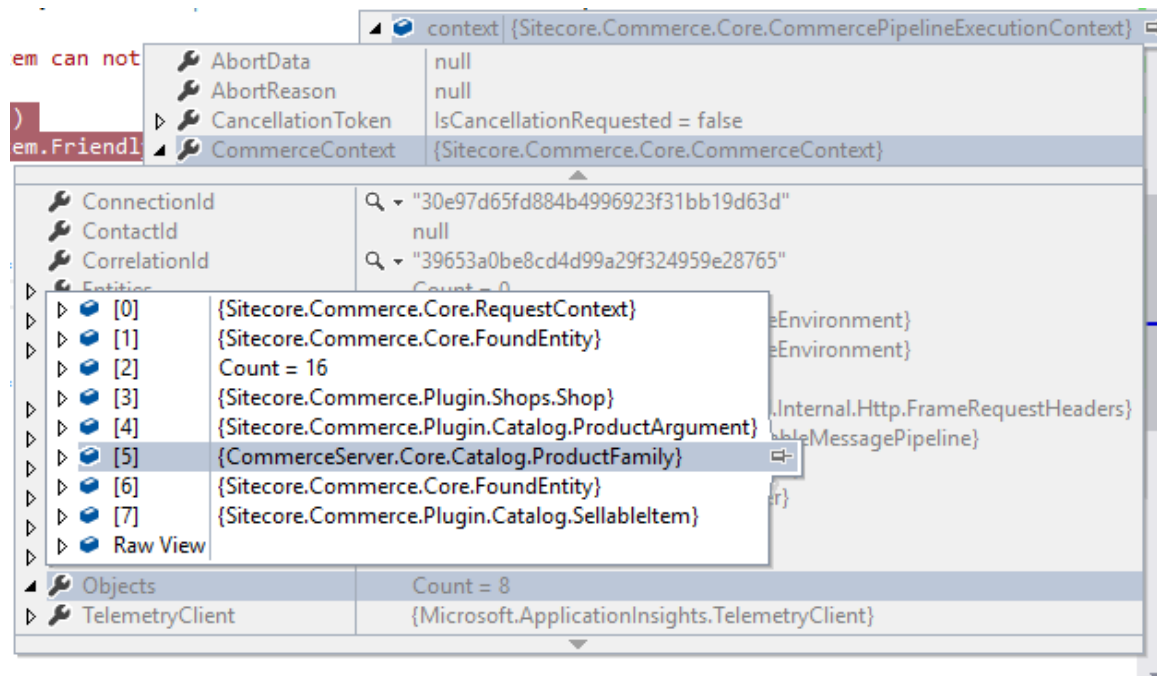
3. Copy and paste the `SampleComponent`, and name it `LoyaltyComponent`. Edit the properties so that there is only one property, named `Points`. You can add more properties later as needed.
4. You can now reference that component directly in the pipeline block, and set the property. For example: `sellableItem.GetComponent<LoyaltyComponent>().Points = (int)product["LoyaltyPoints"];`

The component does not need to be initialized. The special method `GetComponent` will check if the component already exists, and return it or initialize a new one. An instance of that component is guaranteed to be returned, so there is no requirement for null checking. However, you must check for null for the Commerce Server Product property to add a default value if there is no value populated in the `LoyaltyPoints` property.

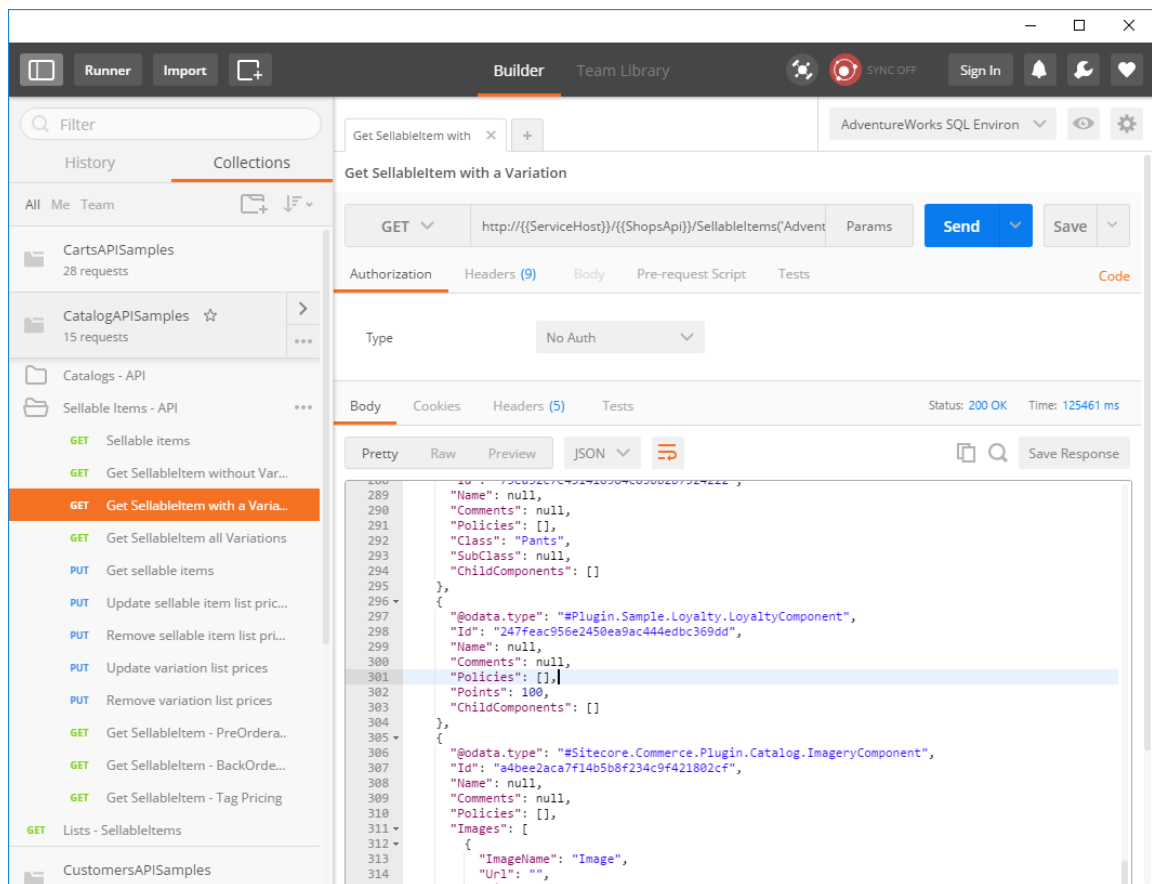
```
var product = context.CommerceContext.GetObjects<Product>()
    .FirstOrDefault(p => p.ProductId.Equals(sellableItem.FriendlyId,
StringComparison.OrdinalIgnoreCase));

if (product["LoyaltyPoints"] == null)
{
    sellableItem.GetComponent<LoyaltyComponent>().Points = 100;
}
else
{
    sellableItem.GetComponent<LoyaltyComponent>().Points =
(int)product["LoyaltyPoints"];
}
```

5. Write some code to map the additional property and populate a new component in the `SellableItem`. Run the solution again, and send the command in Postman to get the sellable item and debug through the break point. It is also helpful to look at what other objects are in the Objects collection. These are all easily accessible to your block:



If the code has run and you get a response in Postman, the new component will be populated:



You have created a new plugin that extends the `SellableItem` Commerce entity with an additional component.

Note

When you want to extend an entity, you must extend it with its own component, rather than try to inherit and extend the component from another entity. This allows a clean separation of concerns between your extensions and any extensions that are made by other plugins.

20.4 Extending a Cart Line

Sitecore Commerce supports easily extending the existing solution using *compositional extensibility* patterns, which enables extending the solution in a way that does not prevent upgrades and enables a separation of concerns between multiple plugins that seek to extend the same Commerce entity.

Scenario: add loyalty points earned to the cart line so it can be carried into the order.

The simplest way to make an item part of the Order is to copy the component from the `SellableItem` into the cart line during the `AddCartLine` pipeline.

To extend a cart line:

1. Open the same `NodeConfiguration` file that you referenced previously, and find the `AddCartLine` pipeline:

```
-----  
Sitecore.Commerce.Plugin.Carts  
IAddCartLinePipeline (Sitecore.Commerce.Plugin.Carts.CartLineArgument =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----  
Plugin.Catalog.ValidateSellableItemBlock  
(Sitecore.Commerce.Plugin.Carts.CartLineArgument =>  
Sitecore.Commerce.Plugin.Carts.CartLineArgument)  
-----  
Plugin.Carts.AddCartLineBlock (Sitecore.Commerce.Plugin.Carts.CartLineArgument =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----  
Plugin.Carts.ICalculateCartLinesPipeline (Sitecore.Commerce.Plugin.Carts.Cart =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----  
Plugin.Carts.ICalculateCartPipeline (Sitecore.Commerce.Plugin.Carts.Cart =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----  
Plugin.GiftCards.AddCartLineGiftCardBlock (Sitecore.Commerce.Plugin.Carts.Cart =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----  
Plugin.Carts.PersistCartBlock (Sitecore.Commerce.Plugin.Carts.Cart =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----  
Plugin.Carts.WriteCartTotalsToContextBlock (Sitecore.Commerce.Plugin.Carts.Cart =>  
Sitecore.Commerce.Plugin.Carts.Cart)  
-----
```

2. Create another component that takes a `Cart` as a parameter and returns a `Cart`, and runs after the `AddCartLineBlock`.
3. To access an Entity from the `Carts` plugin, add a reference to it. For example:

```
/// <summary>  
/// The execute.  
/// </summary>  
/// <param name="arg">  
/// The SampleArgument argument.  
/// </param>  
/// <param name="context">  
/// The context.  
/// </param>  
/// <returns>  
/// The <see cref="SampleEntity"/>.  
/// </returns>  
public override Task<Cart> Run(Cart cart, CommercePipelineExecutionContext  
context)  
{  
    Condition.Requires(cart).NotNull("The argument can not be null");  
    //var result = this. pipeline.Run(arg, context).Result;  
    return Task.FromResult(cart);  
}
```

4. Add the following configuration:

```
.ConfigurePipeline<IAddCartLinePipeline>(  
    configure =>  
    {
```

```
configure.Add<AddCartLineLoyaltyBlock>().After<AddCartLineBlock>();  
    })
```

5. As in the previous scenario, you must run the block with no logic, adding a break point so you can check if the objects that you need are in the objects collection. To test this piece, add a `SellableItem` to the cart. In Postman, go to `CartsAPISamples/Add Cart Line Without Variant` and run it. This adds an item to the cart.
6. When you look at the objects collection, you can see the `SellableItem`, and can get to the line that was added by retrieving the `CartLineArgument` and using the `Line` property. Copy the component from the `SellableItem` to the `Cart Line`. For example:

```
public override Task<Cart> Run(Cart cart, CommercePipelineExecutionContext context)  
{  
    Condition.Requires(cart).NotNull("The argument can not be null");  
  
    var sellableItem =  
context.CommerceContext.GetObjects<SellableItem>().First();  
  
    var arg = context.CommerceContext.GetObjects<CartLineArgument>().First();  
    var cartLine = arg.Line;  
  
    cartLine.SetComponent(sellableItem.GetComponent<LoyaltyComponent>());  
  
    return Task.FromResult(cart);  
}
```

Now, each time an item is added to the cart, the loyalty points for the item are added to the cart. This can be mapped and displayed on the storefront by using the Sitecore Commerce Connect component.

You can run the solution and add the item to the cart to verify using Postman.

Because the item is now a component in a cart line, it is automatically copied over, along with all other components in the cart line, to the order line when an order is placed.

7. For this example, finish the order in Postman.
8. Add a physical fulfillment option, a federated payment option, and complete the order, and then copy the `OrderID` from the completed order results.
9. In Postman, go to `GetOrder`, paste the `OrderID`, and retrieve the order. You can see the `LoyaltyComponent` in the completed order.

20.5 Extending a Commerce View to Show Additional Information in the Business Tools

A related action is to show the summary of points that have been earned on the business users order screen. This screen can be extended by adding a new `EntityView` block.

To extend a commerce view to show additional information:

1. Copy and paste the sample block and name it `GetOrderSummaryViewBlock`.
2. The `GetEntityView` blocks take an `EntityView` as a parameter and return an `EntityView`. Because this extension must see an order, you must add a reference to the Orders plugin:
`Sitecore.Commerce.Plugin.Orders": "1.0.2301`
3. Add the new block:

```
if (request.ViewName != context.GetPolicy<KnownOrderViewsPolicy>().Summary
    && request.ViewName !=
context.GetPolicy<KnownOrderViewsPolicy>().Master)
{
    // Do nothing if this request is for a different view
    return Task.FromResult(entityView);
}

if (request.Entity == null)
{
    // Do nothing if there is no entity loaded
    return Task.FromResult(entityView);
}

// Only do something if the Entity is an order
if (!(request.Entity is Order))
{
    return Task.FromResult(entityView);
}

var order = request.Entity as Order;

EntityView entityViewToProcess;
if (request.ViewName == context.GetPolicy<KnownOrderViewsPolicy>().Master)
{
    entityViewToProcess = entityView.ChildViews.FirstOrDefault(p=>p.Name
== "Summary") as EntityView;
}
else
{
    entityViewToProcess = entityView;
}

int pointsEarned = 0;
foreach(var line in
order.Lines.Where(p=>p.HasComponent<LoyaltyComponent>()))
{
    pointsEarned = pointsEarned +
line.GetComponent<LoyaltyComponent>().Points;
}

entityViewToProcess.Properties.Add(new ViewProperty { Name = "Points
Earned", IsReadOnly = true, RawValue = pointsEarned });

return Task.FromResult(entityView);
```

[End of Document]