**Advanced Pricing Overview**

In the Industries CPQ Pricing course, we covered how Salesforce Industry's pricing solution offers:

* A component-oriented system with reusable items.
* Types of pricing, such as penalties, charges, and adjustments to existing charges.
* Settings that determine the frequency that the charge occurs.
* Pricing components that live independently of products.
* Transitioning from older pricing to newer pricing for less expense and disruption.

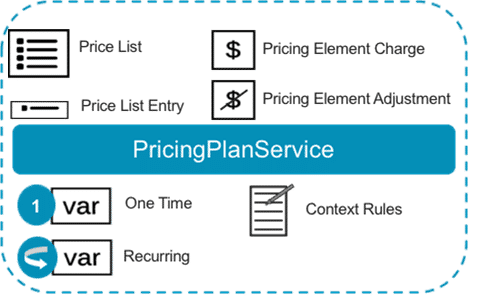
Now, let’s dig a little deeper. Salesforce Industries provides standard and advanced pricing capabilities for you to:

* Offer customers the option of paying with loyalty points.
* Change the price of a product automatically over time.
* Customize the pricing process by adding steps.
* Automatically reprice products, draft items in the Cart, and assets.
* Intercept commands to change an action that would have been performed through the use of an event hook.
* Price energy commodities based on estimated usage.

# Pricing Plan Service

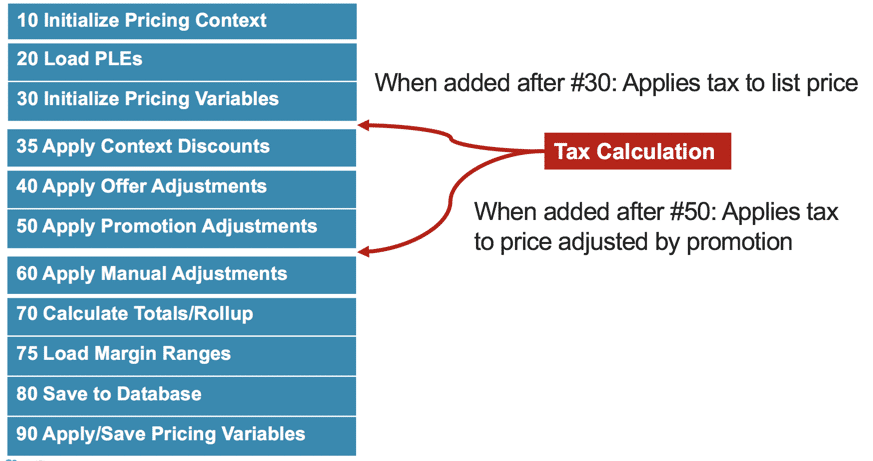
The pricing plan service:

* Works for opportunities, quotes, and orders
* Breaks up the pricing logic into separate steps. This enables flexibility and customization, such as changes to the sequence of some steps and the addition of steps.
* Replaces the hard-coded PricingElementServiceImplementation



The PricingPlanService implementation makes the pricing capabilities possible.

# Pricing Plan Steps



**Inserting a Step or Changing Sequence**

You can insert a step as well as change the sequence of steps. The logical order of steps must make sense, but no step depends on the previous or next step.

**10 Initialize pricing context**

Initializes a pricing plan context map with information needed by subsequent steps. Each step can set or retrieve objects from the context map. Information in the map includes the line items being priced, the Cart header object, the list of associated line adjustment records and promotion item records, and maps that enable efficient processing of hierarchy information.

**20 Load price list entries**

Queries the price list entries associated with the products and price list of the cart and calls the TightestMatchInterface to determine the matching recurring and one-time charges for a product.

**30 Initialize pricing variables**

Charges from tightest match are applied to the recurring and non-recurring pricing variables associated to each line item in the Cart being priced.

**35 Apply context discounts**

Checks if there are existing negotiated or context based discounts associated with an order and applies them. Context = order, account, or contract-based discounts.

**40 - 60 Apply adjustments and overrides**

**40  Apply *offer*adjustments/overrides:**

Applies any offer defined overrides and adjustments loaded in the Initialize Pricing Context step. Overrides are applied first before any adjustments that adjusts the pricing variables associated with them.

**50  Apply *promotion*adjustments/overrides:**

Applies any promotion defined overrides and adjustments loaded in the Initialize Pricing Context step. Overrides are applied first before any adjustments that adjusts the pricing variables associated with them.

**60  Apply *manual*adjustments/overrides:**

Applies any manual overrides or adjustments loaded in the Initialize Pricing Context step.

**70 Calculate totals/rollup**

Calculates line level totals and rollup pricing variables. The calculation starts from the child line items and moves up the hierarchy tree because the parent rollups are dependent on their children pricing variable totals.

**75 Load margin ranges**

Only executed if the Cost and Margin feature is enabled. Calls the MarginRangeLoader interface to invoke a matrix that specifies the upper and lower limits of the margin.

**80 Save**

Saves the line items to the database.

**90 Apply and save pricing variables**

Calculates and saves pricing variables of any parent object (Opportunity/Quote/Order) level.

# Attribute-Based Pricing Overview

Attribute-Based Pricing (ABP) can be implemented using pricing plans, expression sets, and decision matrices. When using ABP, you create a single product entity and a set of attributes, and use a matrix to price each possible combination of attributes.

This reduces the number of products in your catalog and provides an easy way to review and administer your pricing model.

**Products without Attribute-Based Pricing**

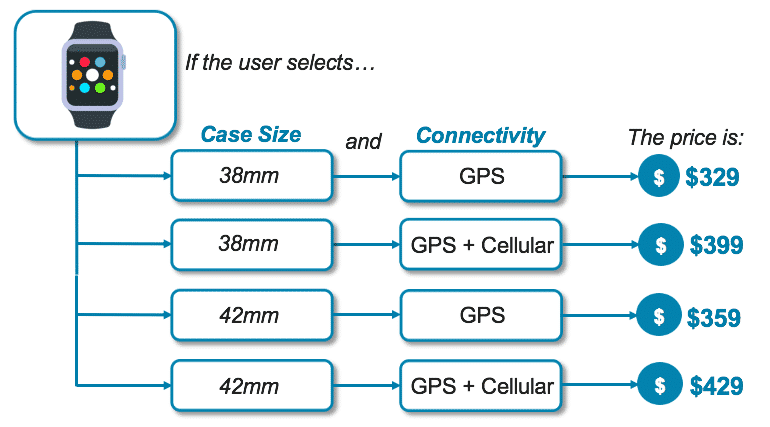


Products often have several characteristics that affect the price. Consider a smartwatch with two product attributes:

* Case Size
* Connectivity

In this example, you would need to create four separate product entities for each configuration in order for each product configuration to be priced accordingly.

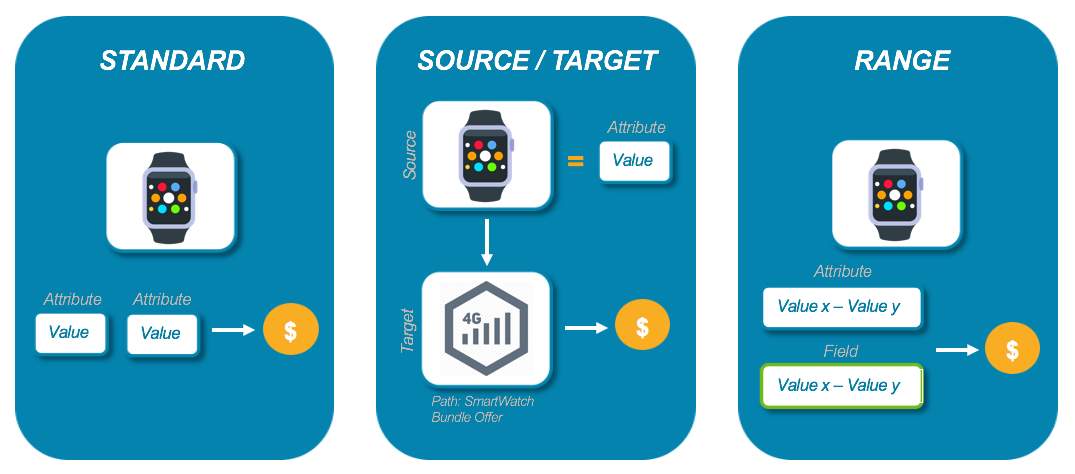
**Products with Attribute-Based Pricing**



In this example, you:

* Create a single product entity (Smartwatch), and
* Set of attributes (Case Size and Connectivity)
* Use a matrix to price each attribute combination

**Types of Attribute-Based Pricing**



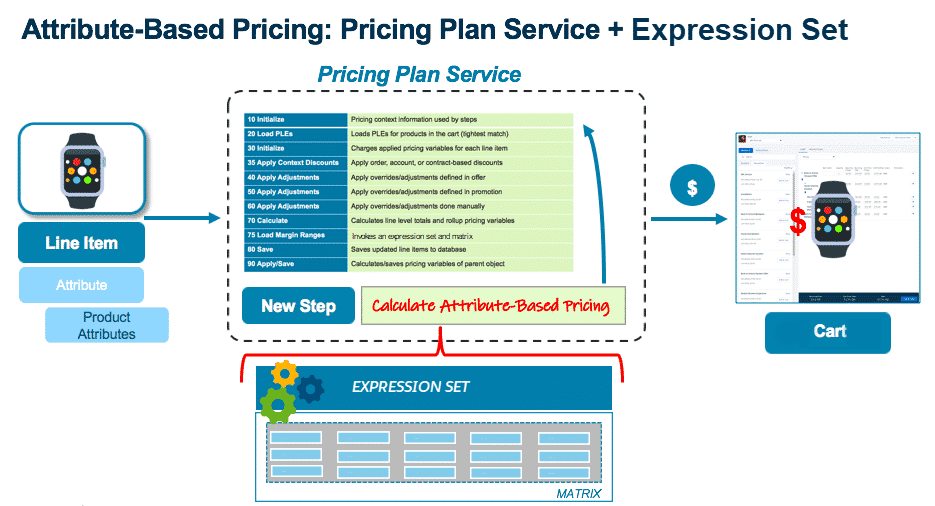
Three types of ABP are supported:

* **Standard**: prices a product using attributes
* **Source/Target**: prices a product based on the attributes of another product
* **Range**: prices a product when the value falls within the defined range for each attribute and field in the matrix

# Attribute-Based Pricing and Pricing Plans

You can invoke the expression sets and matrices from a **pricing plan**. A pricing plan is a series of steps in sequence, and each step invokes a process, function or procedure. You can use pricing plans to perform a wide-variety of calculations such as promotions discounts, taxes, and more --- including attribute-based pricing.

When implementing ABP using pricing plans, you need to create a new step in the default pricing plan. This step will be used to call an expression set and decision matrix that stores the product attributes and pricing information, thereby at runtime, the PricingPlanService implementation passes the Cart items to the calculation service and retrieves the correct price.



# Setting Up Attribute-Based Pricing

**Before You Begin**

To implement ABP, you need to download the following files from the Process Library (in the Salesforce Industries Success Community).

* **AttributePricingDataPack.json** - includes a preconfigured decision matrix for standard attribute-based pricing
* **SourceTargetAttributePricingDataPack.json** - includes a preconfigured decision matrix for source target attribute-based pricing
* **RangeAttributePricingDataPack.json** - includes a preconfigured decision matrix for range attribute-based pricing
* **CustomPricingPlanStepImpl.cls VlocityOpenInterface** - implementation Apex class that exposes the GetMatrixPrice method and invokes the PricingPlanHelper class
* **PricingPlanHelper.cls** - Apex helper class invoked by CustomPricingPlanStepImpl.cls that analyzes the matrix and invokes the expression set.
* **CustomPricingPlanStepImplTest.cls -**Apex test class

**Step 1. Confirm Post-Install Tasks Completed**

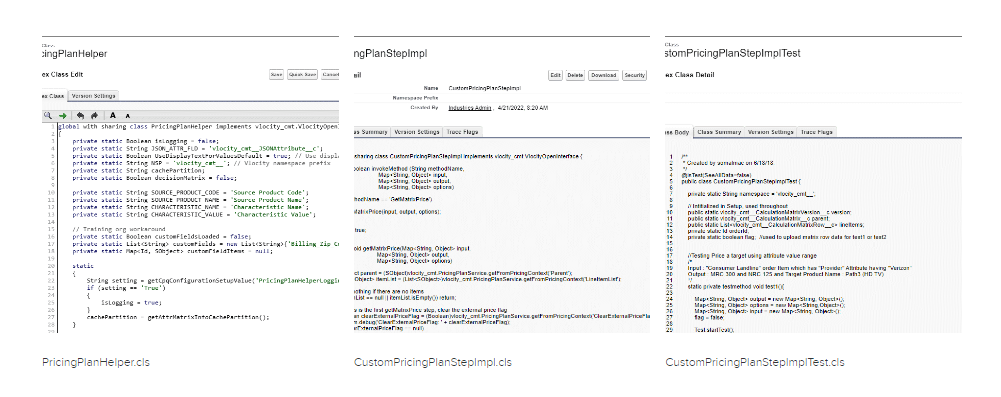
Before setting up attribute-based pricing, confirm that all post-install tasks completed successfully.

1. The Default Pricing Plan is installed.
2. The Parameters field displays in the General Properties pane in the Pricing Plan Steps facet of the Default Pricing Plan.
3. The DefaultPricingPlan custom setting displays in the CPQ Configuration Setup CONFIGURATIONS tab.
4. The PricingPlanService implementation is active and set to default from the PricingInterface on the InterfaceImplementation tab.

**Step 2. Set Up Apex Classes**

Due to code dependencies, you must create the APEX classes in order:

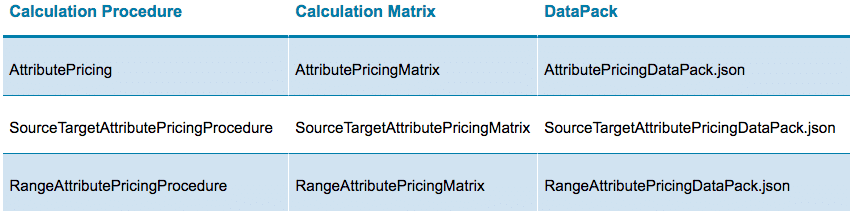
1. PricingPlanHelper.cls
2. CustomPricingPlanStepImpl.cls
3. CustomPricingPlanStepImplTest.cls



**Step 3. Set Up Pricing Matrices**

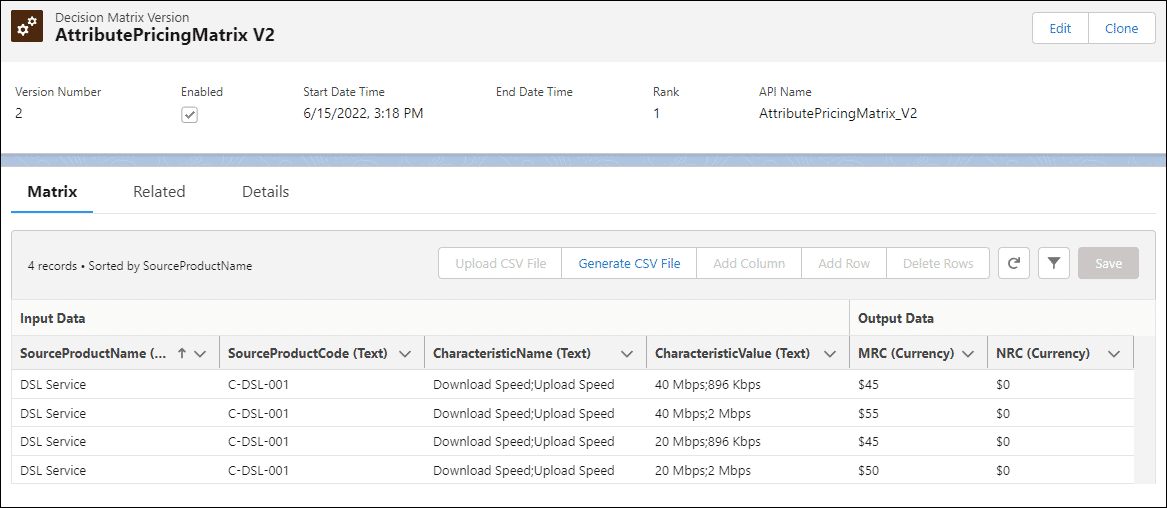
You use decision matrices to store the combinations of product configurations and their corresponding prices. When using a decision matrix of products with one or more attributes, the pricing plan service overrides the product's base price with the price specified for each combination defined in the matrix.

In the file from the Process Library, you will find DataPacks containing preconfigured matrices to support attribute-based pricing. Each DataPack corresponds to a type of attribute-based pricing and has a decision matrix that has been configured to support it. Once imported you can add data to the matrices.



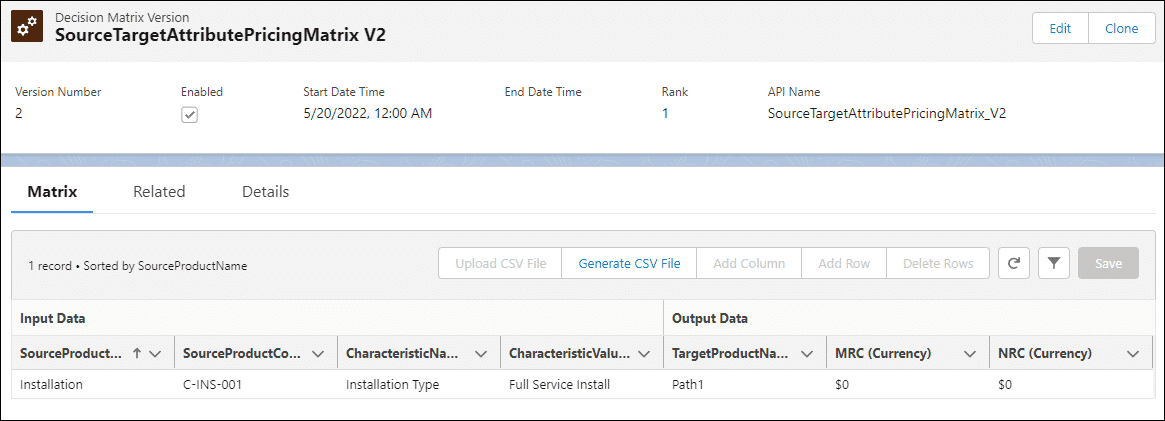
1. Attribute Pricing Matrix

Enables you to implement standard attribute-based pricing where you price a product based on its attributes. The decision matrix has been preconfigured with the column headers that define the source products, product attributes, and the MRC and NRC prices. Enter data manually using the user interface or import a comma-separated values (CSV) file to populate the matrix.



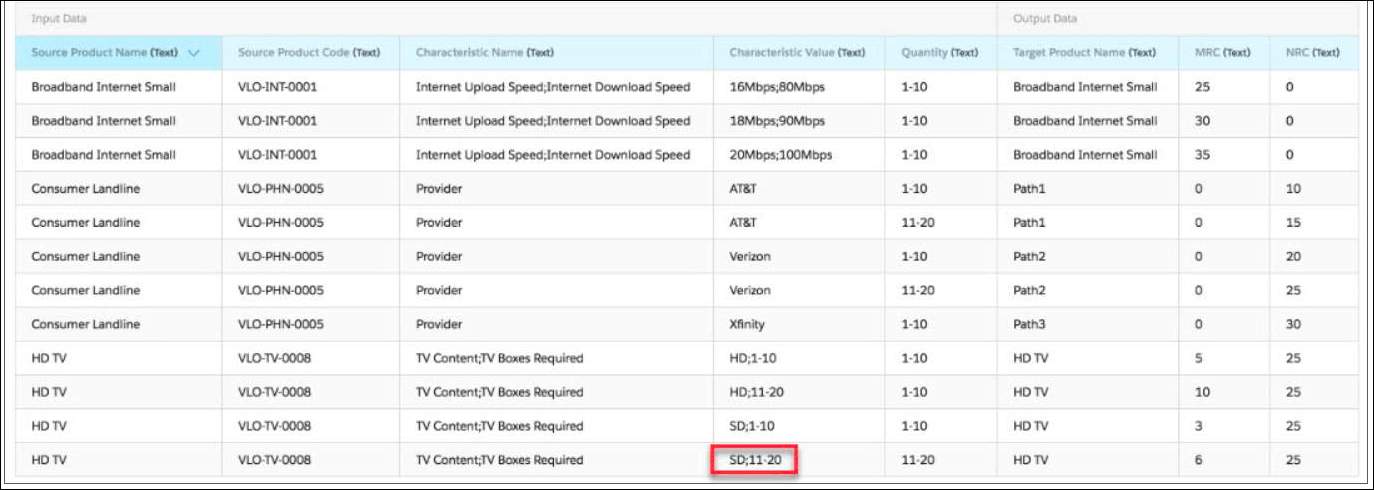
1. Source Target Attribute Pricing Matrix

Using this approach, you can adjust pricing for one product based upon the attributes of another product. You can also use a parameter on the pricing plan step to specify the path to specific products in a bundle. The decision matrix has been preconfigured with the column headers that define the source products, product attributes, MRC and NRC prices, and the Target Product Name. Enter data manually using the user interface or import a comma-separated values (CSV) file to populate the matrix.



1. Range Attribute Pricing Matrix

Using this approach, you can override pricing for a product when, for example, the Quantity field on the line item exceeds a certain threshold or the numeric value of an attribute falls within a certain range. The decision matrix has been preconfigured with the column headers that define the source products, product attributes, MRC and NRC prices, and the Quantity field as an example. Enter data manually using the user interface or import a comma-separated values (CSV) file to populate the matrix.



When entering matrix data, review the following considerations:

**Attribute Pricing Matrix Considerations**

Source Product Code is the key used to match a line item's product to records in the matrix.

* Characteristic Name and Characteristic Value are mapped to Name and Value properties on product attributes.
* Separate multiple attribute names and values by semi-colons.
  + Specify attribute values in the same order as the corresponding attribute names.
* MRC and NRC values do not need to exist as pricing elements on the price list used by the order, quote or opportunity.
  + Values from the matrix override the base pricing element for the line item.
* All rows of the matrix must have values entered in each column.
* Each row of the matrix must be unique.

**Source/Target Pricing Matrix Considerations**

All mentioned in Attribute Pricing Matrix Considerations, plus:

* Target Product Name must be the name of the product to price based upon the attribute values selected for the source product.
  + All products that match the Target Product Name in the Cart receive the pricing override.
* Use the same matrix to include products that do not have a target product alongside products with a target product.
  + For products that do not have a target product, enter a Target Product Name that is the same as the Source Product Name, which applies the pricing override to the source product.

**Range Pricing Matrix Considerations**

All mentioned in Attribute Pricing Matrix Considerations, plus:

* You can specify ranges for numeric fields on Order Product, Quote Product or Opportunity Product sObjects or product attributes of value type number.
* Each range defined for an attribute or field must not overlap with a range on another row of the matrix. (Example: 0-9; 10-19, etc.).
* When entering a range for an attribute specify two numbers separated by a dash. (Example: 10-19).
* Create a separate column for each field that must be evaluated.
* To combine target product and path pricing with range pricing, add a Target Product Name column and adjust the expression set to include the Target Product Name in the output.

**Caching Decision Matrices for Performance**

Caching is not needed for all decision matrices. We recommend implementing your attribute-based pricing solution without a cache at first, and creating the cache if additional speed is required.

Small to medium-sized matrices will likely be quite fast. To optimize performance for large decision matrices or pricing plans that have many attribute-based pricing steps, you can cache your decision matrices in the platform cache.

**Step 4. Review Expression Sets**

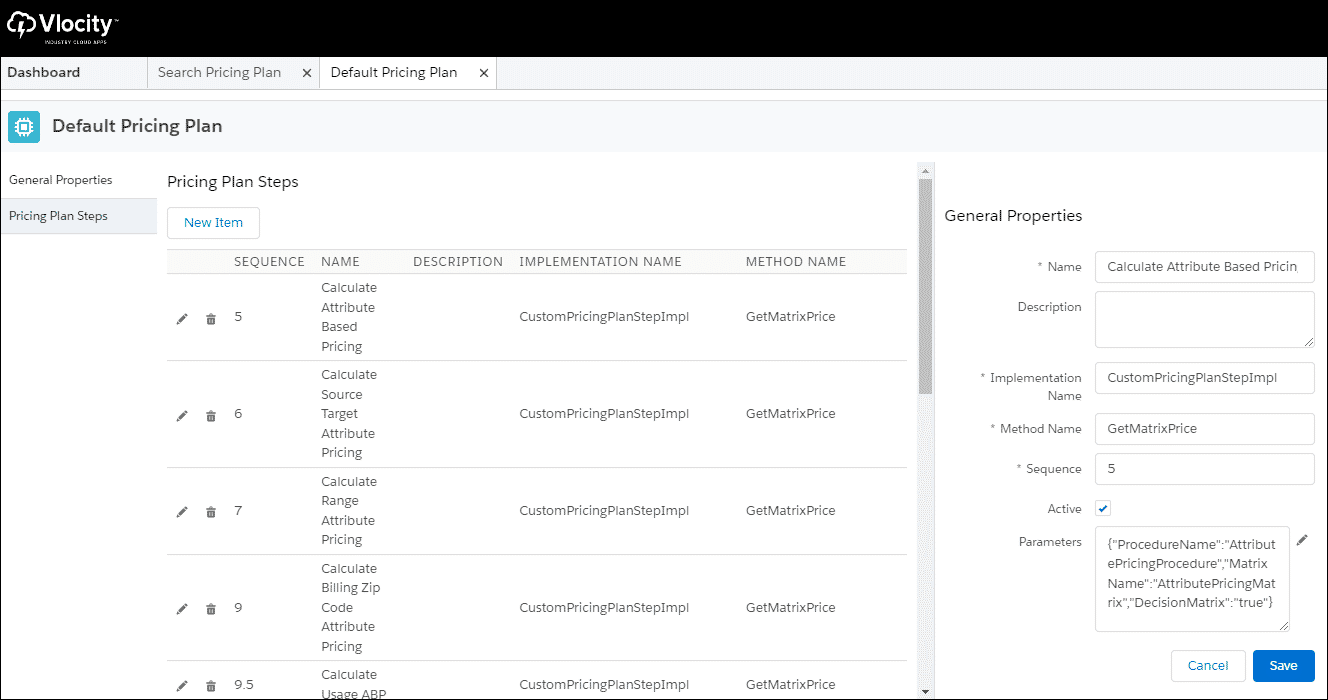
To test the data you entered in your decision matrices, go to the expression set that calls it and simulate the function (using the Classic interface) :

1. In the Lightning App Launcher, navigate to **Expression Sets** and select the expression set that calls the matrix.
2. Click the **Related** tab, and in the Expression Set Versions list, click the latest expression set version.
3. Click **Open in Expression Set Builder**.
4. In the Expression Set Builder, click **Simulate**.
5. Enter sample data for each variable, and then click **Simulate**.
6. Validate results.

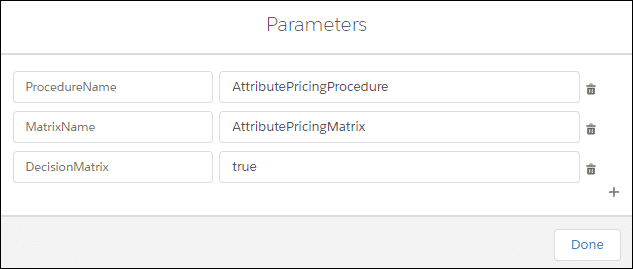
**Step 5. Set Up Custom Pricing Step**

After you have implemented the Apex classes and the expression sets and matrices, create a new step in the Default Pricing Plan, by entering the following:

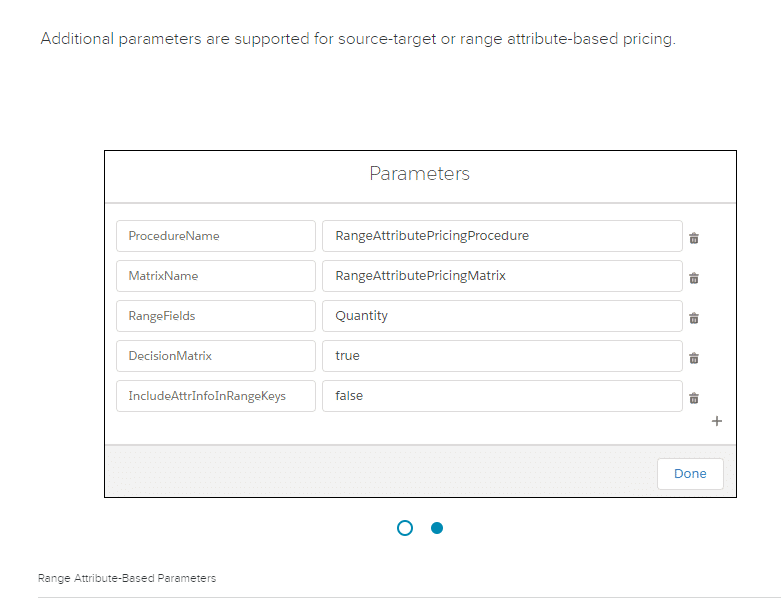
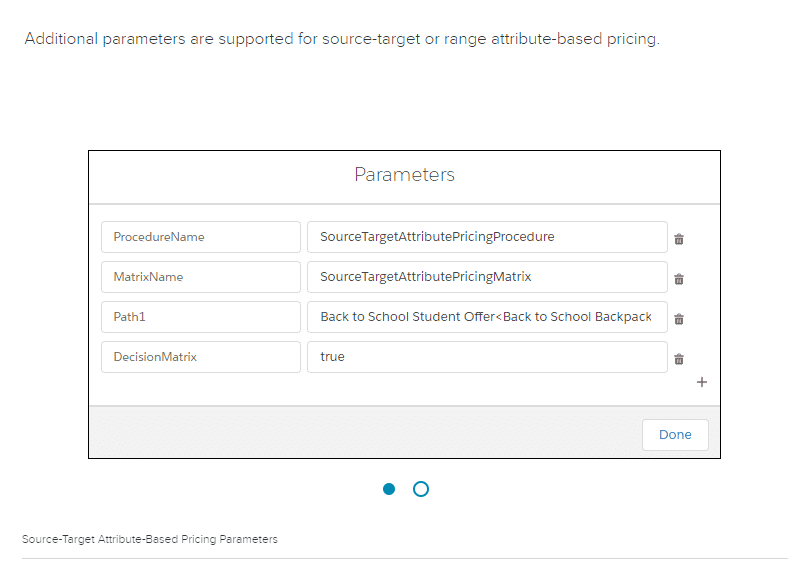
* Name - if you create multiple steps for each type of attribute-based pricing, change the Name to reflect each type.
* Description - custom pricing step.
* Implementation Name - the Apex class included in the Attribute Based Pricing Solution DataPack.
* Method Name - the method in CustomPricingPlanStepImpl.cls.
* Sequence - must be less than 10. All attribute-based pricing steps must occur before the Initialize Pricing Context step.
* Active - check to select.
* Parameters - these parameters are passed to the CustomPricingPlanStepImpl.cls.



**Parameters**



These parameters are used in the expression sets and matrices that are supplied as part of the Attribute-Based Pricing Solution DataPack.



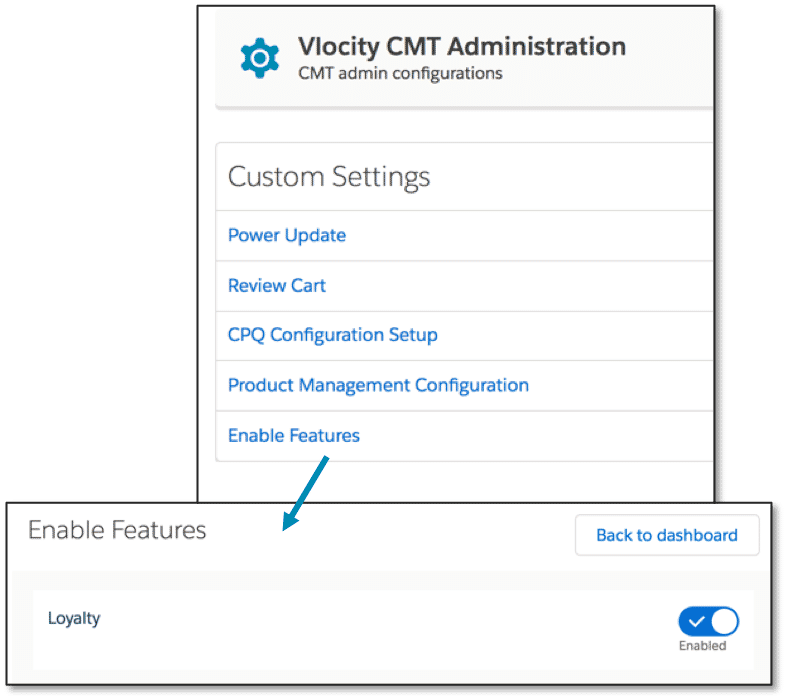
# Loyalty Pricing Overview

With loyalty pricing, your products can be priced in loyalty points as well as currency, and your customers can pay with loyalty points using their loyalty credits. Pricing rollup calculations occur the same way as with currency.

When a product has prices in currency and loyalty points, both are listed as payment choices in the Products list of the Cart.

Salesforce Industries provides the ability to pay for an item in currency or loyalty points. Administering and managing point balances is handled by an external back office system.

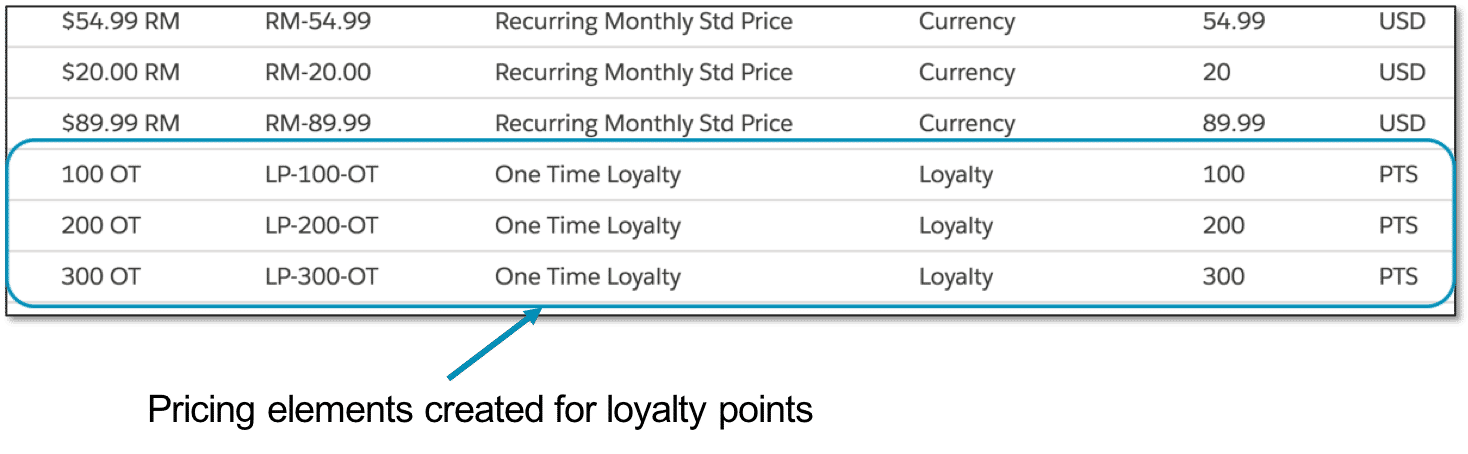
# Set Up Loyalty Pricing



Navigate to **Enable Features** to enable loyalty pricing. Ensure enabling the feature is desired before doing so. Disabling the feature after you have enabled it could cause functionality issues.

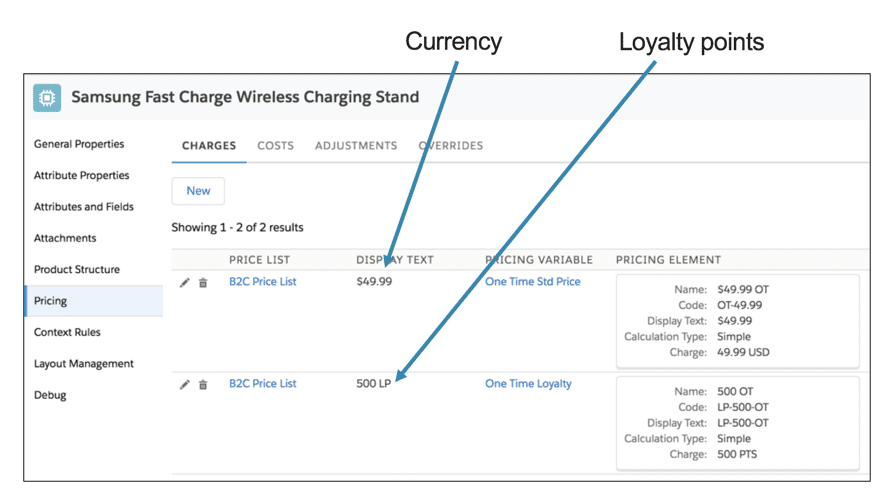
# Loyalty Points

**Pricing Elements for Loyalty Pricing**



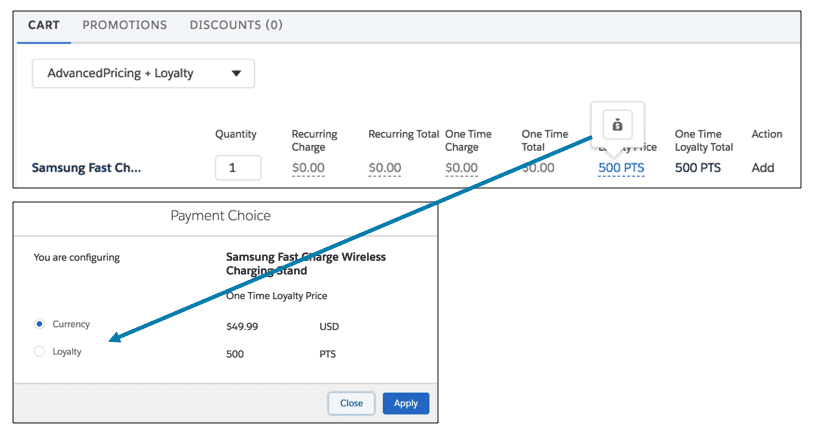
To support loyalty points, you create pricing elements in the same way you create them for currency.

**Assign Loyalty Points to Products**



A product can have a price list entry in currency and a price list entry in loyalty points.

**Pay with Loyalty Points**



In the Cart, you can toggle between the payment options of currency and loyalty points.

# Repricing Overview

Repricing allows you to reprice line items in opportunities, orders, and quotes and assets as needed by your business operations. You can use repricing to support products with prices that fluctuate over time, like natural resources, or products that are priced based on usage or currency exchange rates.

Repricing is done as a background process across one or more records. During repricing, prices are evaluated to ensure they still apply based on the **effectivity dates** of the **price list entry** for the product or applied promotion.

In addition, rules are re-run to ensure that the line items still meet the criteria for any context rules applied to the price list entry. Context rules for products or promotions are not re-run, only context rules for price list entries. Rules for any manual adjustments can also be re-run to ensure they are still valid. If the manual adjustment is no longer valid, the pricing for that line item reverts back to original pricing.

Any price list entries that do not have a current effectivity date yields no change when repricing is run.

**Interface Implementations**

Two implementation types are available in the RepricingInterface:

* **DefaultRepricingImplementation**: will not make any changes to the existing prices.
* **RepricingElementServiceImplementation**: reprices line items or parent objects of the line items (such as order line items or the order header) based on the price list entry effectivity dates attached to the product or promotion at that point in time when the repricing service is run.

There is also the RepricingManAdjEligibilityInterface, which must have the **RepricingManualAdjEligibilityService** implementation active to support repricing using context rules for pricing adjustments.

When repricing, set the interface implementation to **RepricingElementServiceImplementation**.

Repricing utilizes the RepricingElementServiceImplementation whereas attribute-based pricing utilizes the PriclngPlanService implementation. Therefore, repricing does not support attribute-based pricing.

# Repricing Best Practices

* Reprice in batches with input parameters such a batch size or reprice provided line items only. Repricing large updates in real time could cause performance degradation to occur if not run via a batch job.
* When using the repricing service in a batch job or api integration, limit the objectList by 20 in the case of lineItems.
* When invoking the repricing service, make sure the results of the query are sorted by OrderId/QuoteId/OpportunityId/AccountId.

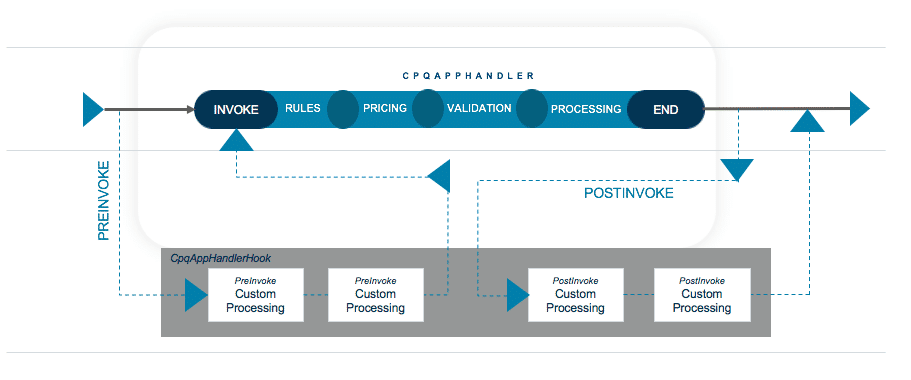
# Event Hooks in Industries CPQ

In software development, hooking is a concept that allows you to modify the original behavior of the application without changing your code. By intercepting the commands, you can change the action that would have been performed originally. It is very useful in the case of adding new functionality to applications, exposing additional fields in pricing calculations or altering how products are priced, to name a few of its applications.

**Using an Event Hook in Industries CPQ**

Industries CPQ allows you to write custom Pre-invoke and Post-invoke hooks for each interface to implement custom logic. Pre-invoke hooks get executed before the actual interface is executed whereas Post-invoke hooks get executed after the actual interface is executed.

In Industries CPQ, CPQ functionality is managed by a global interface called CpqAppHandler. It includes a wide range of methods to perform CPQ processes in the Cart. (These methods are also available as a RESTful API, called the Cart-based APIs.) So, when you need to change the default behavior or processing of a CPQ operation, you create a custom implementation in CpqAppHandler’s hook interface, which is appropriately named CpqAppHandlerHook.

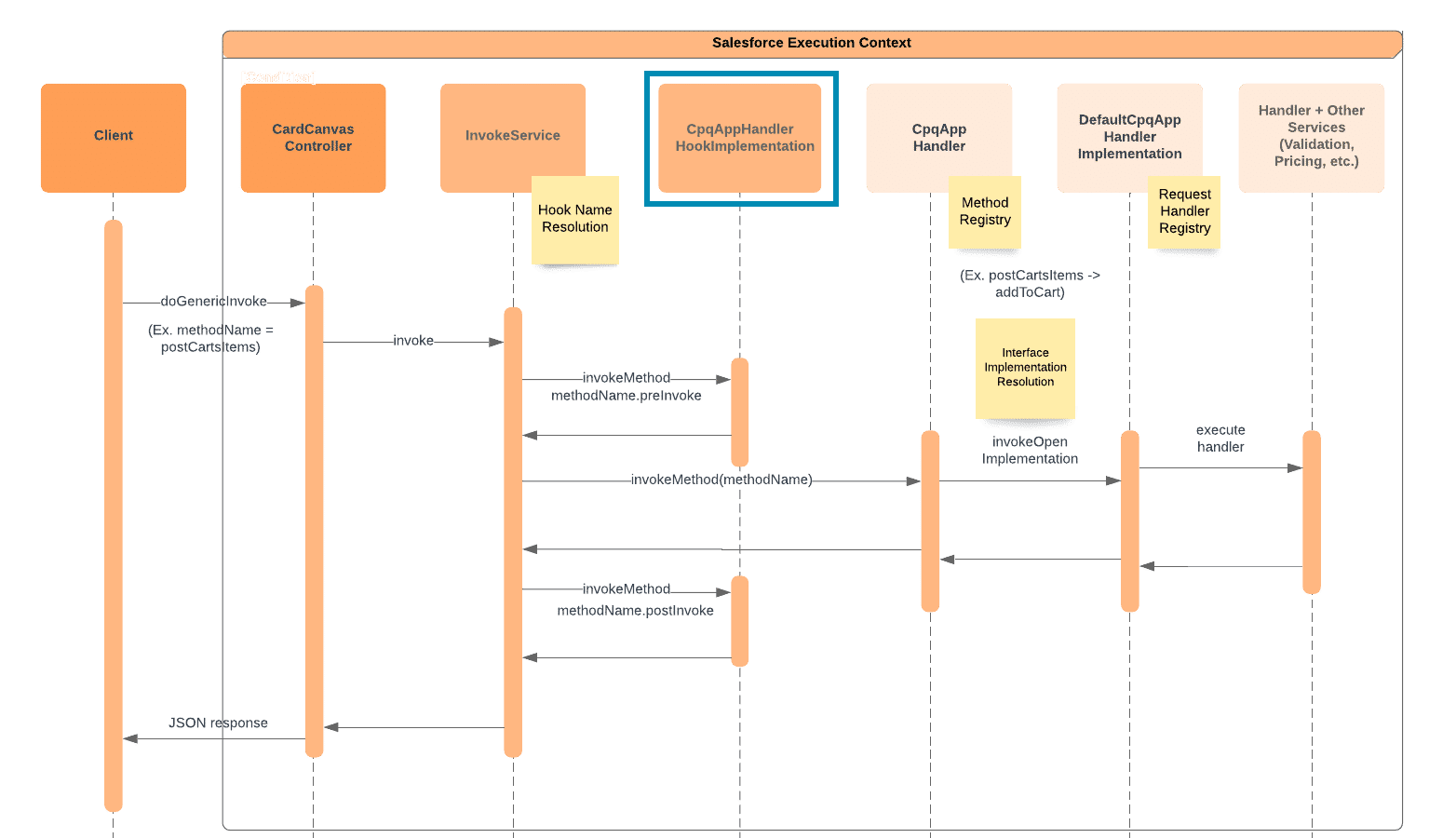


The CpqAppHandlerHook can be used to manage the inputs and outputs of CPQ methods. In addition, you can add business logic to do the following:

| **Manage CPQ methods inputs/outputs** | **Add business logic** |
| --- | --- |
| Add, change or delete input parameters | Conditional processing |
| Add, change, or delete output responses | Preload information that will be used further down the execution chain |
| Debug issues by verifying the input and output are correct | Process custom fields |
| In some cases, provide a workaround when waiting for a patch for an issue | Implement custom functionality |
| - | Do callouts |

**How is the CpqAppHandlerHook Invoked?**

The CpqAppHandlerHook is automatically invoked when the InvokeService is called. The InvokeService simply appends a hook string, e.g., “Hook”, to the interface name and determines if there is an active interface with that name. If there is, the InvokeService will initiate the active implementation in the hook interface. Then, it will add a .preInvoke suffix to the method name and invoke the PreInvoke method in the hook implementation. A corresponding .postInvoke method is invoked on the hook implementation after the hooked class method has completed its processing.



**Caution!**

Injecting custom code is powerful when executed correctly, but you also need to be aware of the ramifications if executed poorly. We recommend the following when implementing a hook:

* Validate that a feature doesn't already exist that addresses your need for a hook.
* Ensure the custom code in the hook doesn't cause the execution to exceed any Apex governor limits.
* Put a try-catch around the hook to catch any exceptions thrown by the custom code.
* When unit testing the hook code, it may be possible to mock the inputs expected by the hook code to minimize test setup and get better code coverage. Use Test.isRunningTest() to extract the mock input from the inputMap or from a static map.

**Examples Using an Event Hook**

Let's take a look at two different examples where an event hook can be used in Industries CPQ.

**Example #1**

**Business Requirement:**

Users must not be allowed to order the same product more than once on the same order. And, check to see if the product exists as an active asset on the account.

**Solution:**

* Implement a CpqAppHandlerHook to check before postCartsItems logic runs to see if the product exists in other active line items or active assets.
* In the Preinvoke hook, if the product exists in active line items or assets, null the items and create a new node that will be checked in the post-hook.
* In the Postinvoke hook, if the new node exists display an error in the UI.
* When the user tries to add a duplicate product, they will get an error in the cart.



**Example #2**

**Business Requirement:**

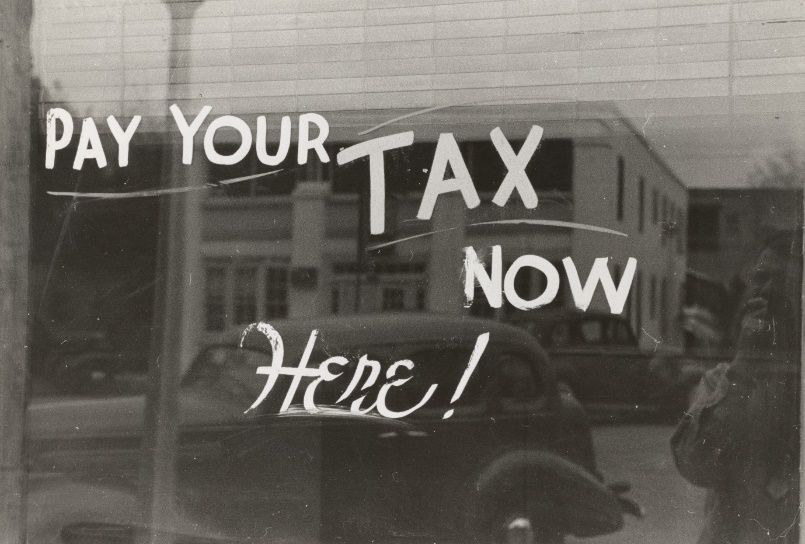
Display the price of an iPhone X plus applicable taxes in the one-time charge column in the Cart.

**Solution:**

Implement a pricing hook to alter how pricing is calculated. For example, Quantity\*Price\*Sales tax Rate.

**Take it further:**

Want to display the sales tax in a separate column? No problem, modify the Cart to display a custom field for the applicable sales tax.



**What is Usage Pricing?**

Usage pricing is a type of pricing used when a customer is charged based on their usage (consumption) of a product or service, such as water, gas, or electricity. Typically, at the end of a period (often, a billing cycle), a service provider collects information about actual service consumption and calculates total charges for that period. Unlike fixed charges, which are predefined, usage-based charges fluctuate from period to period. Then, the customer is charged for their actual usage by the billing system.

This makes sense when you’re talking about services that are installed and already sold to a customer. Most of us have experienced this type of pricing as home or business owners. But things get more complicated when you think about how to price and sell these services before they’re installed—because the usage or consumption hasn’t happened yet. So, during quoting and order capture, you need a way to estimate usage, so you have a defined price that you can share and discuss with your customer.

**In Industries CPQ, usage pricing provides the ability to price a product or service during quoting and order capture based on an estimate of usage.**

It’s important to understand that the customer may not be charged the estimated usage price that displays in the quote or order. Instead, that information can be shared with the customer to help them understand the price they may pay in the future, or it can be used in contracts to lock in a rate for a set period of time. (Note that administering and billing actual usage is handled by an external back-office system.)

Some of the most typical use cases for usage pricing are:

* Long-distance call charges depending on the destination country and call duration.
* Internet data overage charges when a service provider charges for each gigabyte over a certain threshold.
* Cloud backup service proportional to the total volume of backups.
* Cloud computing charges based on the total time of resources being used.
* Utility services, including commodities like electricity, gas, or water.

**What is an Attribute Binding?**

An attribute binding links an attribute to a field on an object so that as you change the attribute’s value in the Cart, it is also changed in the bound field. Why is this useful?

Usually, it's because you want to use the value in some kind of calculation or process. When an attribute value is embedded within the JSONAttribute field, it’s very accessible to the end-users in the Cart, allowing them to change its value easily. But, as it turns out, it’s not very accessible to any other process. You can’t monitor its value and trigger a workflow from it, for example. And importantly for our usage pricing scenario, the pricing engine can’t use attribute values embedded in the JSONAttribute field for its calculations. They need to be stored in a field on the line item.

So, in order to enable usage pricing, we must create an attribute binding to link the attribute to a field, and then the field can be added to the pricing engine’s calculation logic (via a pricing variable binding).

In our usage pricing scenario, we will bind the Estimated Usage Quantity to an existing field that’s provided as part of the application, Usage Quantity (vlocity\_cmt\_\_UsageQuantity\_\_c) on the Order Product (OrderItem) object.

The steps to create an attribute binding are:

1. Create the attribute.
2. Create (or select) the field and object that will be bound to the attribute.
3. Create a Vlocity Object to link the field’s object to the shared catalog’s metadata.
4. Create the attribute binding to link the field to the attribute via the Vlocity Object.

Attribute bindings are a one-way street. you cannot change the bound field's value and have the new value populate the attribute's value in the JSONAttribute field. The attribute always drives the bound field -- and not the other way around.

**What is a Vlocity Object?**

You might have noticed in the steps above that you need to create a Vlocity Object as part of creating an attribute binding, and naturally, you’re probably wondering what that is.

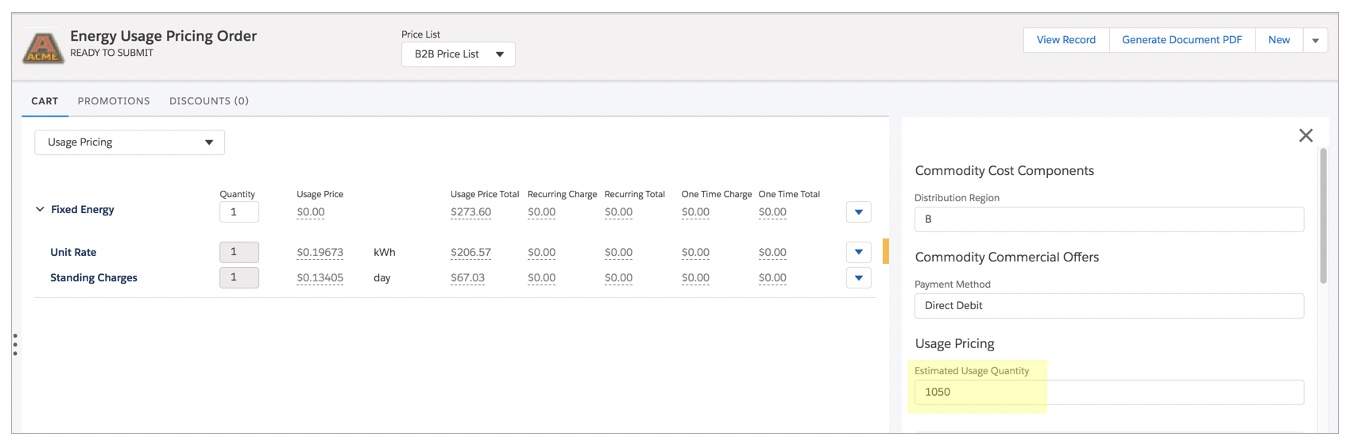
Salesforce Industries has many objects that it has added to Salesforce’s data model, there are some specific objects that the shared catalog considers its responsibility to manage. These objects are called Vlocity Objects, and they can be viewed in the Vlocity Objects and Object Types tab. You’ll recognize most of them because they are precisely the objects you work with using the Product Console—objects like Product2, Attribute, Price List, Rules, and so on. Vlocity Objects adds a layer of metadata around the object itself, and the shared catalog uses the metadata to perform its functions.

So, when you create an attribute binding to a field on an object that is outside the shared catalog’s “field of vision”, so to speak, you must also define the object as a Vlocity Object so that shared catalog knows about it. You only need to do this once for each object. In our scenario, we need to do it for the Order Product (OrderItem) object, and then you could create multiple attribute bindings to different fields on the Order Product object. If we wanted to enable usage pricing during quoting, we’d need to also create a Vlocity Object for the Quote Line Item (QuoteLineItem) object. And so on, for each object that has a field for which you want to create an attribute binding.

**Configure an Attribute and Attribute Binding**

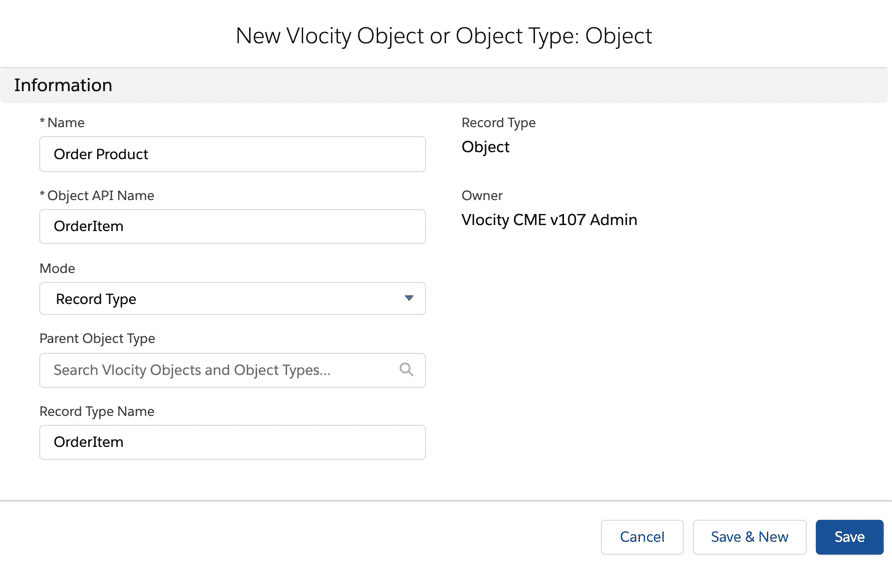
### 1. Create an attribute -- Estimated Usage Quantity

This attribute will display in the Cart and hold the estimated value of energy consumption.



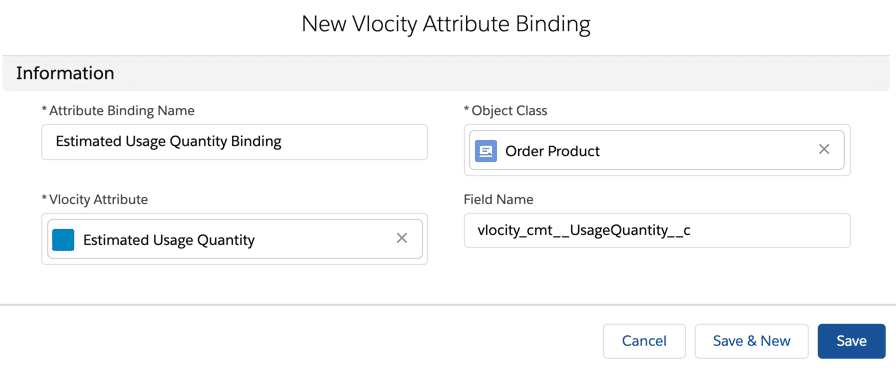
### 2. Create a Vlocity Object for Order Product (OrderItem)

Identify the name of the new object, the object's API name, mode type, and record type name.



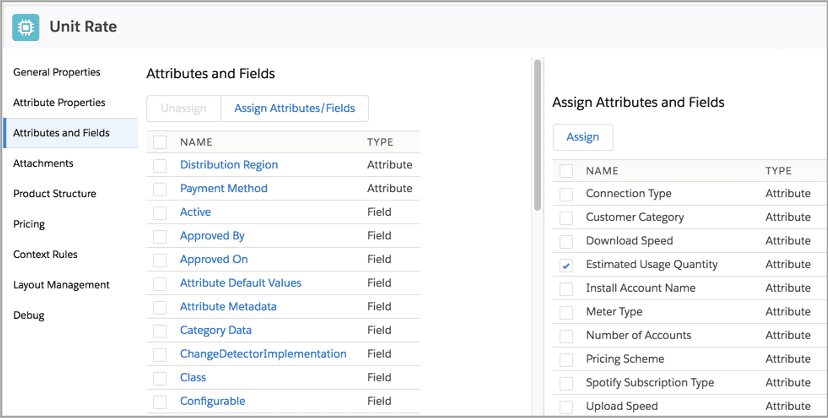
### 3. Create an attribute binding

Bind the Estimated Usage Quantity attribute to the Usage Quantity field on the Order Product (OrderItem) object.



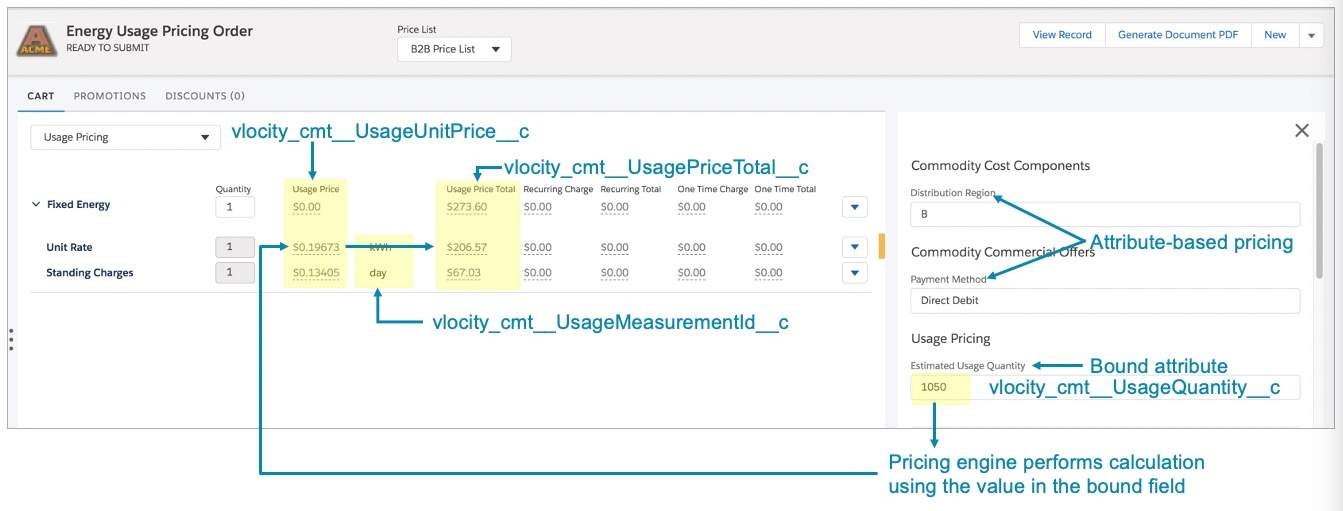
### 4. Assign bound attribute to a product

Assign the bound attribute, Estimated Usage Quantity, to the Fixed Energy child products -- Unit Rate and Standing Charges.



### 5. The Cart

The image displays the attribute, Estimated Usage Quantity, bound to the vlocity\_cmt\_\_UsageQuantity\_\_c field on the OrderItem object where the pricing engine detects it and can perform calculations in the Cart.



# Configure ABP for Usage Pricing

In many Energy & Utilities deployments, usage pricing alone is not enough. The cost of energy commodities fluctuates and will vary based on customers’ geographic location. Therefore, in addition to usage pricing, Eliza needs to create a decision matrix to change the usage price of the Unit Rate and Standing Charges products using attribute-based pricing. And, in order to use attribute-based pricing with usage pricing, you need to extend attribute-based pricing to work with new usage pricing variables.

The steps to configure ABP for usage pricing are:

1. Create an attribute-based pricing matrix.
2. Create an attribute pricing procedure.
3. Create a custom pricing plan step.
4. Configure PricingPlanHelper Apex Class for usage pricing.