How to Use IPlacedPriceProcessor in Optimizely Commerce to Preserve Custom Line Item Prices (Donation Example)

ON 15 OCTOBER, 202515 OCTOBER, 2025 / BY FRANCISCO QUINTANILLA / IN OPTIMIZELY

When working with **Optimizely Commerce Connect**, the built-in pricing pipeline automatically validates and updates each line item's `PlacedPrice` based on catalog pricing.
That's great for normal SKUs — but what if your business model allows **user-entered amounts**, such as **donations**, **gift amounts**, or **pay-what-you-want** products?

By default, Commerce's validation flow will overwrite the user's custom price with the default catalog price whenever the cart is validated or saved.

In this post, you'll learn how to **extend `IPlacedPriceProcessor`** to prevent Optimizely from replacing user-defined prices while keeping the rest of the pricing system intact.

Understanding the Role of `IPlacedPriceProcessor`

`IPlacedPriceProcessor` is the component responsible for retrieving and assigning the correct catalog price to a line item when it's added to the cart or validated.

The default implementation (`PlacedPriceProcessor`) runs during the cart validation pipeline:

```
// Called internally by
DefaultOrderGroupValidator
_placedPriceProcessor.UpdatePlacedPrice(lineItem
, customerContact, marketId, currency,
onValidationError);
```

This ensures every product has a valid catalog price — but it also means that any manually set `lineItem.PlacedPrice` values are replaced.

To support user-entered prices (like donations), we can **replace** this processor to skip re-pricing when we know the price came directly from the user interface.

The Donation Scenario

Imagine a "Give a Gift" block on your site that lets donors:

- Choose an occasion (variant)
- Choose a recipient
- Enter a **custom donation amount**
- Click "Send Gift" to add it to their cart

When the donation hits the Commerce cart, the pipeline runs — and if we don't stop it, our `PlacedPrice` (e.g. $25 entered by the donor) is replaced by the default catalog price (e.g. $10 from price lists).

The Solution: Custom `IPlacedPriceProcessor`

The cleanest and safest way to preserve user-entered prices is to **wrap the default `IPlacedPriceProcessor`** with your own implementation.
This lets you short-circuit the catalog lookup whenever the line item has a special flag (e.g., `IsCustomPrice`).

Implementation

```
//
/Features/Commerce/Pricing/CustomPlacedPriceProc
essor.cs
using System;
using EPiServer.Commerce.Catalog.ContentTypes;
using EPiServer.Commerce.Order;
using Mediachase.Commerce;
using Mediachase.Commerce.Customers;

namespace YourProject.Features.Commerce.Pricing
{
    public class CustomPlacedPriceProcessor :
IPlacedPriceProcessor
    {
        public const string IsCustomPriceKey =
"IsCustomPrice";
        private readonly IPlacedPriceProcessor
_inner;

        public
CustomPlacedPriceProcessor(IPlacedPriceProcessor
inner)
        {
            _inner = inner ?? throw new
ArgumentNullException(nameof(inner));
        }

        private static bool IsCustom(ILineItem
lineItem)
        {
            if (lineItem == null) return false;
            if
(lineItem.Properties.ContainsKey(IsCustomPriceKe
y))
            {
                var val =
lineItem.Properties[IsCustomPriceKey];
                return val is bool b && b &&
lineItem.PlacedPrice > 0m;
            }
            return false;
        }

        public bool UpdatePlacedPrice(
            ILineItem lineItem,
            CustomerContact customerContact,
```

```
            MarketId marketId,
            Currency currency,
            Action<ILineItem, ValidationIssue>
onValidationError)
        {
            // Skip catalog price lookup for
user-defined amounts
            if (IsCustom(lineItem))
            {
                return true; // consider price
valid as-is
            }

            return
_inner.UpdatePlacedPrice(lineItem,
customerContact, marketId, currency,
onValidationError);
        }

        public Money? GetPlacedPrice(
            EntryContentBase entry,
            decimal quantity,
            CustomerContact customerContact,
            MarketId marketId,
            Currency currency)
        {
            // delegate for all non-custom cases
            return _inner.GetPlacedPrice(entry,
quantity, customerContact, marketId, currency);
        }
    }
}
```

Registering your own implementation

```
context.Services.Intercept<IPlacedPriceProcessor
>((locator, defaultImplementation) =>
    new
CustomPlacedPriceProcessor(defaultImplementation
));
```

Marking Custom-Priced Line Items

When creating or updating the cart (e.g., in your `CartService` or controller), mark any line that should skip catalog pricing:

```
if (request.PlacedPrice.HasValue &&
request.PlacedPrice.Value > 0)
{
    lineItem.PlacedPrice =
request.PlacedPrice.Value;

lineItem.Properties[CustomPlacedPriceProcessor.I
sCustomPriceKey] = true;
}
```

This flag tells your processor: "Don't overwrite this value — it's already correct."

What Happens Now

When the cart is validated:

1. `DefaultOrderGroupValidator` calls `_placedPriceProcessor.UpdatePlacedPrice(...)`.
2. Your implementation sees the `IsCustomPrice` flag.
3. It returns `true` immediately — the inner processor never runs.
4. The donor's entered amount remains untouched.
5. The rest of the pricing, discounts, and tax calculations continue as usual.

Why This Is the Right Extension Point

Many developers first try to override `ILineItemCalculator` or modify the cart after validation, but that's reactive and fragile. Changing `IPlacedPriceProcessor` works at the **source** — the exact point where Optimizely sets the price — and integrates cleanly with the existing service pipeline.