The Kentico E-commerce Solution allows you to customize the shopping cart calculation engine. The system runs this calculation when working with the shopping cart during checkout and when creating or editing orders. The calculation determines how the system applies discounts, handles shipping costs, adds taxes, etc.

For example, you can:

- Completely replace the shopping cart calculation with a custom solution
- Integrate custom steps into the calculation process

To customize the shopping cart calculation:

1. Open your Kentico project in Visual Studio.
2. Create new classes that implement the calculation interfaces (described below).

   Class location

   For production sites, we recommend creating a new assembly (Class library project) in your Kentico solution and including the classes there. Then, add the appropriate references to both the assembly and the main Kentico web project. For more information, see Best practices for customization.

3. Implement all methods required by the given interfaces.
4. Register your implementations of the interfaces using the **RegisterImplementation** assembly attribute (or use a *Factory* class to serve instances dynamically in certain cases).

When you reload the website, the system uses the custom implementations when calculating the price totals and other data of shopping carts and orders.

In code, you can run the calculation for a shopping cart object (*ShoppingCartInfo*) by calling the object's **Evaluate()** method.

Calculation interfaces and data types

All mentioned classes can be found in the **CMS.Ecommerce** namespace.

Main calculation logic

The core shopping cart calculation is performed by the registered implementations of the following interfaces:

- **IShoppingCartCalculator**
- **IShoppingCartCalculationFactory** – serves an instance of a class implementing *IShoppingCartCalculator* for each site. Must be implemented to use custom *IShoppingCartCalculator* implementations.

Implementations of *IShoppingCartCalculator* must contain the **Calculate** method. The method provides a **CalculatorData** parameter with the following properties:

- **Request** – a **CalculationRequest** object that contains the data of the processed shopping cart, such as a collection of cart items (products), the customer object, the cart currency, etc.
- **Result** – a **CalculationResult** object that stores the results of the calculation, such as shopping cart totals, discount summaries, tax summaries, etc.

Note: To ensure that your custom calculator implementations work without errors, you need to perform *null* checks before accessing **CalculationRequest** properties. The system runs the calculation process during early stages of the shopping cart life cycle where many properties are not set yet. For example, the *CalculationRequest.PaymentOption* property has a *null* value when calculating shopping carts where the customer has not selected a payment option yet.

To fully replace the shopping cart calculation logic, the **Calculate** method of your *IShoppingCartCalculator* implementation needs to process the data provided in **CalculatorData.Request** and set all available properties of **CalculatorData.Result**.

The default shopping cart calculation in Kentico uses a composite *IShoppingCartCalculator* step, consisting of multiple other *IShoppingCartCalculator* implementations that are executed in a specific order. See the Default calculation pipeline section for details.

Integration with shopping carts

The E-commerce API uses the following interfaces to connect the main calculation logic to shopping cart objects:

- **IShoppingCartAdapterService** – prepares the data for the *IShoppingCartCalculator* calculations and applies the results to the shopping cart. Implementations must contain the following methods:
  - **GetCalculationRequest**, **GetCalculationResult** – methods that initialize the *CalculationRequest* and *CalculationResult* objects for the calculation using the data of a specified shopping cart (*ShoppingCartInfo*).
  - **ApplyCalculationResult** – applies the results (*CalculationResult*) to the specified shopping cart (*ShoppingCartInfo*) after the calculation.

    You can create and register your own **IShoppingCartAdapterService** implementation if you need to add custom properties into the calculation data for use within *IShoppingCartCalculator* classes:

    1. Create custom classes that inherit from **CalculationRequest** or **CalculationResult**.
    2. Add properties to the classes according to your requirements.
    3. Define the **GetCalculationRequest** and **GetCalculationResult** methods within your *IShoppingCartAdapterService* implementation and return instances of your derived request or result classes.
    4. If you need to use custom result properties to apply values to the shopping cart, convert the *CalculationResult* parameter to your derived type within the **ApplyCalculationResult** method.

- **IShoppingCartEvaluator** – contains an *Evaluate* method that encapsulates the entire shopping cart evaluation and calculation logic. The default implementation prepares the calculation data using *IShoppingCartAdapterService*, runs the calculation using *IShoppingCartCalculationFactory*, and ensures automatic adding of free products to shopping carts for Buy X Get Y discounts.

Default calculation pipeline

The default shopping cart calculation in Kentico uses a composite implementation of the *IShoppingCartCalculator* interface – the **ShoppingCartCalculatorCollection** class. The *ShoppingCartCalculatorCollection* accepts an *IEnumerable* collection of other *IShoppingCartCalculator* instances (steps), calls the **Calculate** method of each step, and passes the **CalculatorData** between the steps.

The default calculation pipeline consists of the following steps (each step is an *IShoppingCartCalculator* implementation):

	Calculation step	Function
1	UnitPriceCalculator	Calculates the unit price for all products in the shopping cart (including unit-level discounts – catalog discounts and volume discounts). Sets the following properties of each *CalculationResultItem* in the *CalculationResult.Items* collection: • *ItemUnitPrice* • *UnitDiscount* • *UnitDiscountSummary*

2	CartItemDiscountCalculator	Evaluates and calculates Buy X Get Y discounts and Product coupons for the shopping cart. • Sets the *ItemDiscount* and *ItemDiscountSummary* properties for each *CalculationResultItem* in the *CalculationResult.Items* collection. • Adds any coupon codes related to the applied discounts into the *CalculationResult.AppliedCouponCodes* collection.
3	TotalValuesCalculator	Calculates and sets the following total values: • *CalculationResult.Subtotal* • *CalculationResult.Total* • *CalculationResult.GrandTotal* • *CalculationResultItem.LineSubtotal* for each item in the *CalculationResult.Items* collection At this point, the *Subtotal*, *Total* and *GrandTotal* values are all the same – a sum of the *LineSubTotal* values for all items in *CalculationResult.Items*.
4	OrderDiscountsCalculator	Evaluates and calculates order discounts for the shopping cart. • Sets the *CalculationResult.OrderDiscount* value. • Adds each applied order discount to the *CalculationResult.OrderDiscountSummary* collection. • If any of the applied order discounts use a coupon code, adds the coupon codes to the *CalculationResult.AppliedCouponCodes* collection.
5	TotalValuesCalculator	Recalculates and sets the following total values: • *CalculationResult.Subtotal* • *CalculationResult.Total* • *CalculationResult.GrandTotal* • *CalculationResultItem.LineSubtotal* for each item in the *CalculationResult.Items* collection The *CalculationResult.OrderDiscount* value set by the previous step is now subtracted from the *Total* and *GrandTotal* values.
6	ShippingCalculator	Calculates the shipping price for the shopping cart. • Sets the *CalculationResult.Shipping* value. • If any free shipping offers with a coupon code are applied, adds the coupon codes to the *CalculationResult.AppliedCouponCodes* collection.
7	TaxCalculator	Calculates taxes for all *CalculationRequest.Items* and the *CalculationResult.Shipping* value. • Sets the overall *CalculationResult.Tax* value. • Fills the *CalculationResult.TaxSummary* collection based on the applied tax types.
8	TotalValuesCalculator	Recalculates and sets the following total values: • *CalculationResult.Subtotal* • *CalculationResult.Total* • *CalculationResult.GrandTotal* • *CalculationResultItem.LineSubtotal* for each item in the *CalculationResult.Items* collection The *CalculationResult.Shipping* and *CalculationResult.Tax* values set by the previous steps are now added to the *Total* and *GrandTotal* values.

9	OtherPaymentsCalculator	Calculates the values of gift cards applied to the shopping cart. • Sets the *CalculationResult.OtherPayments* value. • Adds each applied gift card to the *CalculationResult.OtherPaymentsApplication* summary collection. • Adds the coupon code of each applied gift card to the *CalculationResult.AppliedCouponCodes* collection.
10	TotalValuesCalculator	Recalculates and sets the following total values: • *CalculationResult.Subtotal* • *CalculationResult.Total* • *CalculationResult.GrandTotal* • *CalculationResultItem.LineSubtotal* for each item in the *CalculationResult.Items* collection The *CalculationResult.OtherPayments* value set by the previous step is now subtracted from the final *GrandTotal* value.

Modifying the default calculation

You can use the **ShoppingCartCalculatorCollection** class to customize the shopping cart calculation while preserving the default calculation (or parts of it):

1. Prepare an *IEnumerable* collection of **IShoppingCartCalculator** instances. Define your own calculation pipeline, consisting of the default calculation steps and your own custom *IShoppingCartCalculator* steps.
2. Create and register a custom **IShoppingCartCalculationFactory** implementation.
3. Implement the **GetCalculator** method in the *IShoppingCartCalculationFactory* class and return an instance of the default **ShoppingCartCalculatorCollection** class, with the prepared *IShoppingCartCalculator* collection as the constructor parameter.

> **Note**: If you have custom steps that modify the **Subtotal**, **Total** or **GrandTotal** values of the **CalculationResult**, you either need to run the steps after the last usage of the default **TotalValuesCalculator** step or you need to implement a custom calculator for the total values that reflects your customizations. Otherwise the *TotalValuesCalculator* step performs the default calculation of the shopping cart totals, which overwrites any values assigned by custom steps.

Example – Adding an extra charge calculation step

The following example demonstrates how to add a custom step to the shopping cart calculation, while preserving all of the default calculation logic. The step adds an extra charge to the total price of the shopping cart or order based on the selected payment option.

Prepare a separate project for custom classes in your Kentico solution:

1. Open your Kentico solution in Visual Studio.
2. Create a new *Class Library* project in the Kentico solution (or reuse an existing custom project).
3. Add references to the required Kentico libraries (DLLs) for the new project:
    a. Right-click the project and select **Add -> Reference**.
    b. Select the **Browse** tab of the **Reference manager** dialog, click **Browse** and navigate to the ***Lib*** folder of your Kentico web project.
    c. Add references to the following libraries (and any others that you may need in your custom code):
        - **CMS.Base.dll**
        - **CMS.Core.dll**
        - **CMS.DataEngine.dll**
        - **CMS.Ecommerce.dll**
        - **CMS.Globalization.dll**
        - **CMS.Helpers.dll**

- **CMS.Membership.dll**

4. Reference the custom project from the Kentico web project *(CMSApp* or *CMS)*.
5. Edit the custom project's **AssemblyInfo.cs** file (in the *Properties* folder).
6. Add the **AssemblyDiscoverable** assembly attribute:

```
using CMS;

[assembly:AssemblyDiscoverable]
```

Continue by creating custom implementations of the **IShoppingCartCalculationFactory** and **IShoppingCartCalculator** interfaces:

1. Add a new class under the custom project, implementing the *IShoppingCartCalculator* interface.
2. Implement the **Calculate** method to define the custom calculation logic that adds the extra charge to the shopping cart totals.

```
using CMS.Ecommerce;
using CMS.Core;

public class ExtraChargeShoppingCartCalculator : IShoppingCartCalculator
{
    /// <summary>
    /// Runs shopping cart calculation based on specified calculation data.
    /// </summary>
    public void Calculate(CalculatorData calculationData)
    {
        CalculationRequest request = calculationData.Request;
        CalculationResult result = calculationData.Result;

        decimal extraCharge = 0;

                // Adds an extra charge of 5 (in the site's main currency) to the
shopping cart totals
                // if the selected payment option's code name is 'custompayment'
        if (calculationData.Request.PaymentOption?.PaymentOptionName.
ToLowerInvariant() == "custompayment")
        {
            extraCharge = 5m;

            // Uses the default Kentico currency API to convert the price
            // from the site's main currency into the requested currency (if
necessary)
            ICurrencyConverterFactory currencyConverterFactory = Service.
Resolve<ICurrencyConverterFactory>();
            ICurrencyConverter currencyConverter = currencyConverterFactory.
GetCurrencyConverter(calculationData.Request.Site);

            string siteMainCurrencyCode = Service.
Resolve<ISiteMainCurrencySource>().GetSiteMainCurrencyCode(calculationData.
Request.Site);
            string currencyCode = calculationData.Request.Currency.CurrencyCode;

            extraCharge = currencyConverter.Convert(extraCharge,
siteMainCurrencyCode, currencyCode);
        }

        result.Total += extraCharge;
        result.GrandTotal += extraCharge;
    }
}
```

3. Add a new class under the custom project, implementing the *IShoppingCartCalculationFactory* interface.
4. Prepare an *IEnumerable* collection of *IShoppingCartCalculator* instances matching the default calculation steps, and add an instance of the custom *ExtraChargeShoppingCartCalculator* step to the end.
5. Implement the **GetCalculator** method in the *IShoppingCartCalculationFactory* class.
    - Return an instance of the default **ShoppingCartCalculatorCollection** class, with the prepared *IShoppingCartCalculator* collection as the constructor parameter.

```
using System.Collections.Generic;

using CMS;
using CMS.Ecommerce;
using CMS.DataEngine;

// Registers the custom implementation of IShoppingCartCalculationFactory
[assembly: RegisterImplementation(typeof(IShoppingCartCalculationFactory), typeof
(CustomShoppingCartCalculationFactory))]

public class CustomShoppingCartCalculationFactory :
IShoppingCartCalculationFactory
{
    // Provides an instance of the customized shopping cart calculator
    public IShoppingCartCalculator GetCalculator(SiteInfoIdentifier
siteIdentifier)
    {
        // This sample does not parameterize the calculator based on the current
site
        // Use the method's 'siteIdentifier' parameter if you need different
calculation logic for each site
        return new ShoppingCartCalculatorCollection(CustomCalculationSteps);
    }

    // Defines a custom pipeline of calculation steps
    private static List<IShoppingCartCalculator> CustomCalculationSteps =
        new List<IShoppingCartCalculator>
        {
            // The default calculation pipeline
            new UnitPriceCalculator(),
            new CartItemDiscountCalculator(),
            new TotalValuesCalculator(),
                        new OrderDiscountsCalculator(),
            new TotalValuesCalculator(),
                        new ShippingCalculator(),
            new TaxCalculator(),
            new TotalValuesCalculator(),
            new OtherPaymentsCalculator(),
            new TotalValuesCalculator(),

            // Adds the custom "extra charge" step to the end of the calculation
            new ExtraChargeShoppingCartCalculator()
        };
}
```

6. Save all changes and **Build** the custom project.

The registered **CustomShoppingCartCalculationFactory** class provides an instance of **ShoppingCartCalculatorCollection** with an extended list of calculation steps. The calculation performs all of the default steps without any changes, and the custom **ExtraChargeShoppingCartCalculator** step at the end of the calculation pipeline adds an extra charge to orders that use the *"Cu stom payment"* payment option.

Note: This basic example does not display any information about the reason for the price increase to customers or store administrators (for example in the shopping cart details during checkout, when editing orders, or in invoices). The total values of shopping carts or orders may appear inconsistent unless you add content explaining the price change (for example, you can mention the extra charge in the display name of the related payment option).