In the **Kentico.Ecommerce** integration package, you can find supporting models and services to build a shopping cart in your on-line store. Visitors can put products into shopping carts and after their browsing through your store, they can continue with the checkout process to create an order and become a customer.

Creating a shopping cart on MVC websites

When a visitor opens your website, you need to create a shopping cart instance and assign the instance to the customer. To create a shopping cart, use the **Kentico.Ecommerce** integration package.

1. Open your MVC project in Visual Studio.
2. Create a controller for the shopping cart.
   - In all documentation examples regarding the checkout process, the **CheckoutController** is shared by all checkout process steps.

     ```
     public class CheckoutController : Controller
     {
     }
     ```

3. Initialize the **ShoppingService, PricingService, KenticoPaymentMethodRepository**, **KenticoCustomerAddressRepository**, and **KenticoShippingOptionRepository** classes that connects to the shopping cart in Kentico.

   We recommend using a dependency injection container to initialize service instances. When configuring the lifetime scope for the services and repositories, create a separate instance for each request.

   ```
           private readonly IShoppingService shoppingService;
           private readonly IPricingService pricingService;
           private readonly IPaymentMethodRepository paymentRepository;
           private readonly ICustomerAddressRepository addressRepository;
           private readonly IShippingOptionRepository shippingOptionRepository;

           /// <summary>
           /// Constructor.
           /// You can use a dependency injection container to initialize the
   services and repositories.
           /// </summary>
           public CheckoutController()
           {
               shoppingService = new ShoppingService();
               pricingService = new PricingService();
               paymentRepository = new KenticoPaymentMethodRepository();
               addressRepository = new KenticoCustomerAddressRepository();
               shippingOptionRepository = new KenticoShippingOptionRepository();
           }
   ```

After initializing the services and repositories, your MVC application can use the shopping cart and other steps of the checkout process from Kentico. The shopping cart processes actions that customers require when using your on-line store. Namely, customers can expect the following actions that you need to implement:

- Display the shopping cart content
- Add a product to the shopping cart
- Update an item (a product) already in the shopping cart
- Remove an item (a product) from the shopping cart
- Continue with the checkout process to make an order

The source code examples below often use classes from the **Kentico.Ecommerce** namespace from the **Kentico.Ecommerce** integration package.

Tip: To view the full code of a functional example directly in Visual Studio, download the Kentico MVC solution from GitHub and inspect the **LearningKit** or **DancingGoat** projects.

Displaying the shopping cart

Add an action method that retrieves and displays the current visitor's shopping cart to the checkout controller:

```
        /// <summary>
        /// Displays the current site's shopping cart.
        /// </summary>
        public ActionResult ShoppingCart()
        {
            // Gets the current user's shopping cart
            ShoppingCart currentCart = shoppingService.GetCurrentShoppingCart();

            // Initializes the shopping cart model
            ShoppingCartViewModel model = new ShoppingCartViewModel
            {
                // Assigns the current shopping cart to the model
                Cart = currentCart,
                RemainingAmountForFreeShipping = pricingService.
CalculateRemainingAmountForFreeShipping(currentCart)
            };

            // Displays the shopping cart
            return View(model);
        }
```

If you do not want to use coupon codes, you do not need to create a model for your shopping cart. You can just display the shopping cart and adjust the view accordingly:

```
return View(cart);
```

In the shopping cart's view, you can then use the shopping cart's properties to display the content of the shopping cart. For example:

```
@if (Model.Cart.IsEmpty)
{
    <span>Your shopping cart is empty.</span>
}
else
{
    <ul>
        @* Loops through all shopping cart items *@
        @foreach (Kentico.Ecommerce.ShoppingCartItem cartItem in Model.Items)
        {
            @* Displays the shopping cart items' properties *@
            <li>
                @cartItem.Units&times; @cartItem.Name ... @currency.FormatPrice
(cartItem.Subtotal)

                @using (Html.BeginForm("RemoveItem", "Checkout", FormMethod.Post))
                {
                    @Html.Hidden("ItemId", cartItem.ID)
                    <input type="submit" value="Remove" />
                }
            </li>
        }
    </ul>
}
```

Add a button to enable the visitors to continue with the checkout process.

```
    @* The continue button *@
    @Html.ActionLink("Continue to the customer details step", "DeliveryDetails")
```

The example code targets at the customer details step.

Evaluating and saving the shopping cart

When working with objects of the **Kentico.Ecommerce.ShoppingCart** class, you need to manually handle recalculation and saving of the shopping cart. Otherwise your shopping carts may display or store incorrect values.

Perform these operations by calling the following methods for the shopping cart object:

- **Evaluate()** – fully recalculates the shopping cart's values and totals based on its current content and properties (discounts, taxes, shipping, etc.). You need to call the *Evaluate* method after you modify the shopping cart content or set any properties that could affect the calculation results.
- **Save()** – saves the given shopping cart's state into the database. You need to call the *Save* method after making any shopping cart modifications that you wish to persist in the database. Not required for changes made to shopping cart items that are stored separately (i.e. products in the shopping cart or applied coupon codes).

Adding a product to the shopping cart

To the checkout controller, add a POST method that adds a product to the shopping cart. For example, the system can trigger the method after a visitor clicks the *Add to cart* button.

```
        /// <summary>
        /// Adds products to the current site's shopping cart.
        /// </summary>
        /// <param name="itemSkuId">ID of the added product (its SKU object).</param>
        /// <param name="itemUnits">Number of added units.</param>
        [HttpPost]
        public ActionResult AddItem(int itemSkuId, int itemUnits)
        {
            // Gets the current user's shopping cart
            ShoppingCart cart = shoppingService.GetCurrentShoppingCart();

            // Adds a specified number of units of a specified product
            cart.AddItem(itemSkuId, itemUnits);

            // Evaluates the shopping cart
            cart.Evaluate();

            // Displays the shopping cart
            return RedirectToAction("ShoppingCart");
        }
```

Updating a shopping cart item

To the checkout controller, add a POST method that changes the number of units of a shopping cart item. For example, the system can trigger the method after a visitor clicks the *Update* button in the shopping cart.

```
        /// <summary>
        /// Updates number of units of shopping cart items in the current site's
shopping cart.
        /// </summary>
        /// <param name="itemID">ID of the updated shopping cart item.</param>
        /// <param name="itemUnits">Result number of units of the shopping cart item.<
/param>
        [HttpPost]
        public ActionResult UpdateItem(int itemID, int itemUnits)
        {
            // Gets the current user's shopping cart
            ShoppingCart cart = shoppingService.GetCurrentShoppingCart();

            // Updates a specified product with a specified number of units
            cart.UpdateQuantity(itemID, itemUnits);

            // Evaluates the shopping cart
            cart.Evaluate();

            // Displays the shopping cart
            return RedirectToAction("ShoppingCart");
        }
```

Removing a shopping cart item

To the checkout controller, add a POST method that removes a shopping cart item from the shopping cart. For example, the system can trigger the method after a visitor clicks the *Remove* button in the shopping cart.

```
        /// <summary>
        /// Removes a shopping cart item from the current site's shopping cart.
        /// </summary>
        /// <param name="itemID">ID of the removed shopping cart item.</param>
        [HttpPost]
        public ActionResult RemoveItem(int itemID)
        {
            // Gets the current user's shopping cart
            ShoppingCart cart = shoppingService.GetCurrentShoppingCart();

            // Removes a specified product from the shopping cart
            cart.RemoveItem(itemID);

            // Evaluates the shopping cart
            cart.Evaluate();

            // Displays the shopping cart
            return RedirectToAction("ShoppingCart");
        }
```

Checking out

To the checkout controller, add a method that enables the visitor to continue with the checkout process to the next step to make an order. You need to validate the shopping cart first.

The next step is usually a step where the visitor/customer types their details, see Creating the customer details step in MVC checkout processes.

```
        /// <summary>
        /// Validates the shopping cart and proceeds to the next checkout step with
customer details.
        /// </summary>
        [HttpPost]
        [ActionName("ShoppingCart")]
        public ActionResult ShoppingCartCheckout()
        {
            // Gets the current user's shopping cart
            ShoppingCart cart = shoppingService.GetCurrentShoppingCart();

            // Validates the shopping cart
            ShoppingCartCheckResult checkResult = cart.ValidateContent();

            // If the validation is successful, redirects to the next step of the
checkout process
            if (!checkResult.CheckFailed)
            {
                // Evaluates the shopping cart
                cart.Evaluate();

                return RedirectToAction("DeliveryDetails");
            }

            // If the validation fails, redirects back to shopping cart
            return RedirectToAction("ShoppingCart");
        }
```