Lesson 6

The exercises for this lesson is based on Adam Friman: *Pro ASP.NET MVC 5*, pp. 214-234 and adopted for the MbmStore project. In these exercises, we'll build a shopping cart for the MbmStore application. The project you'll work on during the exercises is the hand-in you did for mandatory assignment 1.

Exercise 1, making the catalogue view a strongly typed view (Mandatory)

Whenever you write code, you always want to optimize it to be efficient and well organized. To convert the catalogue view to a strongly typed view, we have to change the controller to send data as a strongly typed model to the view. We want to display products in the view and therefore we want the controller to send a list og products to the view+:

```
public class CatalogueController : Controller
{
    // GET: Catalogue
    public ActionResult Index()
    {
        Repository repository = new Repository();
        return View(repository.Products);
    }
}
```

Next, we must change the view itself to a strongly typed view. Right now, we have a `List` of products, but later on when the products come from the database the collection of products might not be a `List` but a collection of some other type. For that reason, it is better to use a more generic type like `iEnumerable`, because many different collection types implement that interface which supports iteration over a collection.

```
@using MbmStore.Models;
@model IEnumerable<Product>
@{
    ViewBag.Title = "Index";
}
```

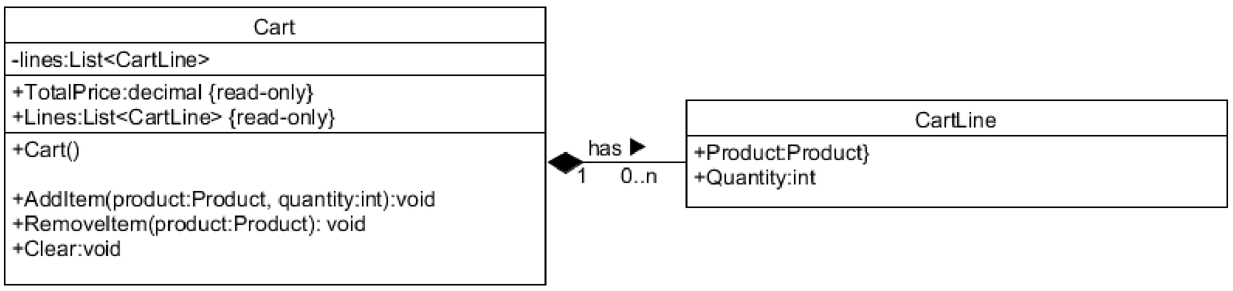
For each of the product categories, you call the `Model`, insert that specific category into a list, and iterate over it:

```
@foreach (Book book in Model.OfType<Book>().ToList())
{
    @RenderBook(book);
    <br />
}
```

Build the project and display the `http://localhost:[portnumber]/Catalogue` page to ensure that everything is fine and the page still looks the same.

Exercise 2, the shopping cart model (Mandatory)

To implement a shopping cart, we need a model to keep track of products in the cart.


Cart
-lines:List<CartLine>
+TotalPrice:decimal {read-only}
+Lines:List<CartLine> {read-only}
+Cart()
+AddItem(product:Product, quantity:int):void
+RemoveItem(product:Product): void
+Clear:void
has
1
0..n
CartLine
+Product:Product}
+Quantity:int

This model is for supporting the UI and it is not part of the data domain model that already has the Invoice and `OrderItem` classes. For that reason, we'll create In **ViewModels** folder for these classes:

```
public class Cart
  {

     private List<CartLine> lines = new List<CartLine>();

     public decimal TotalPrice
     {
        // Linq syntax
        get { return lines.Sum(e => e.Product.Price * e.Quantity); }
     }

     public List<CartLine> Lines { get { return lines; } }

     public Cart() { }

     public void AddItem(Product product, int quantity)
     {

        CartLine item = lines.Where(p => p.Product.ProductId == product.ProductId).FirstOrDefault();

        if (item == null)
        {
           lines.Add(new CartLine { Product = product, Quantity = quantity });
        }
        else
        {
           item.Quantity += quantity;
        }
     }

     public void RemoveItem(Product product)
     {
        lines.RemoveAll(i => i.Product.ProductId == product.ProductId);
     }

     public void Clear()
     {
        lines.Clear();
     }
  }
```

```
public class CartLine
{
   public Product Product { get; set; }
   public int Quantity { get; set; }
}
```

The `Cart` class uses the `CartLine` class, defined in the same file, to represent a product selected by the customer and the quantity the user wants to buy. We defined methods to add an item to the cart, remove a previously added item from the cart, calculate the total cost of the items in the cart, and reset the cart by removing all of the items. We also provided a property that gives access to the contents of the cart using a `List<CartLine>`. This is all straightforward stuff, easily implemented in C# with the help of a little LINQ.

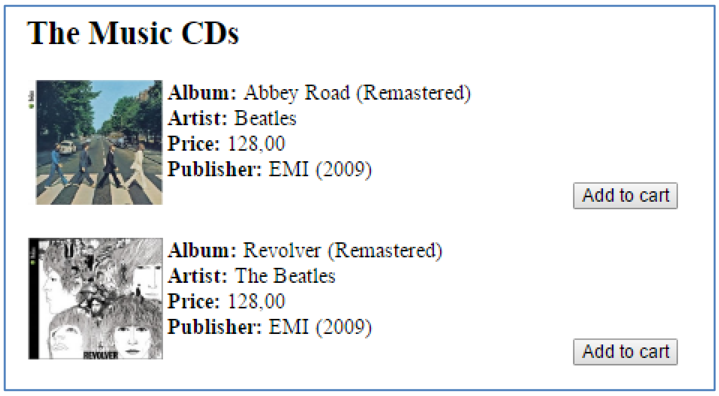
Exercise 3, adding the Add to Cart buttons (Mandatory)

We need to edit the Views/Catalogue/Index.cshtml view to add the buttons to the product listings.

```
@using (Html.BeginForm("AddToCart", "Cart"))
{
        @Html.Hidden("ProductId", movie.ProductId)
        @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
        <input type="submit" class="btn btn-success" value="Add to cart" />
}
```

Note: By default, the `BeginForm` helper method creates a form that uses the HTTP `POST` method. You can change this so that forms use the `GET` method, but you should think carefully about doing so. The HTTP specification requires that `GET` requests must be *idempotent*, meaning that they must not cause changes, and adding a product to a cart is definitely a change.

Add a button to each product like this:



The Music CDs

Album: Abbey Road (Remastered)
Artist: Beatles
Price: 128,00
Publisher: EMI (2009)

Album: Revolver (Remastered)
Artist: The Beatles
Price: 128,00
Publisher: EMI (2009)

Creating multiple html forms in a page

Using the Html.BeginForm helper in each product listing means that every Add to cart button is rendered in its own separate HTML form element. This may be surprising if you have been developing with ASP.NET Web Forms, which imposes a limit of one form per page if you want to use the view state feature or complex controls (which tend to rely on view state). Since ASP.NET MVC does not use view state, there is no limit the number of forms you can create.

Equally, there is no requirement to create a form for each button. However, since each form will post back to the same controller method, but with a different set of parameter values, it is a nice and simple way to deal with the button presses.

Exercise 4, implementing the cart controller (Mandatory)

We need a controller to handle the Add to cart button presses. Create a new controller called `CartController` and edit the content so that it matches:

```
public class CartController : Controller
 {
        private Repository repository;

        // constructor
        // instantiale a new repository object
        public CartController()
        {
            repository = new Repository();
        }


        public RedirectToRouteResult AddToCart(int productId, string returnUrl)
        {
            Product product = repository.Products.FirstOrDefault(p => p.ProductId ==
productId);

            if (product != null)
            {
                GetCart().AddItem(product, 1);
            }

            return RedirectToAction("Index", new { controller=returnUrl.Substring(1)
});
        }

public RedirectToRouteResult RemoveFromCart(int productId, string returnUrl)
{
    Product product = repository.Products
    .FirstOrDefault(p => p.ProductId == productId);
    if (product != null)
    {
        GetCart().RemoveItem(product);
    }
    return RedirectToAction("Index", new { controller="Cart" });
}

        private Cart GetCart()
        {
            Cart cart = (Cart)Session["Cart"];

            if (cart == null)
            {
                cart = new Cart();
                Session["Cart"] = cart;
            }
            return cart;
        }
 }
```

There are a few points to note about this controller. The first is that I use the ASP.NET session state feature to store and retrieve `Cart` objects. This is the purpose of the `GetCart` method. ASP.NET has a nice session feature that uses cookies or URL rewriting to associate multiple requests from a user together to form a single browsing session. A related feature is session state, which associates data with a session. This is an ideal fit for the `Cart` class. We want each user to have their own cart, and we want the cart to be persistent between requests. Data associated with a session is deleted when a session expires (typically because a user has not made a request for a while), which means that we do not need to manage the storage or life cycle of the `Cart` objects. To add an object to the session state, we set the value for a key on the `Session` object, like this:

```
...

Session["Cart"] = cart;

...
```

To retrieve an object again, we simply read the same key, like this:

```
...

Cart cart = (Cart)Session["Cart"];

...
```

Tip: Session state objects are stored in the memory of the ASP.NET server by default, but you can configure a range of different storage approaches, including using a SQL database. See ASP.NET Session State Overview (msdn) for details.

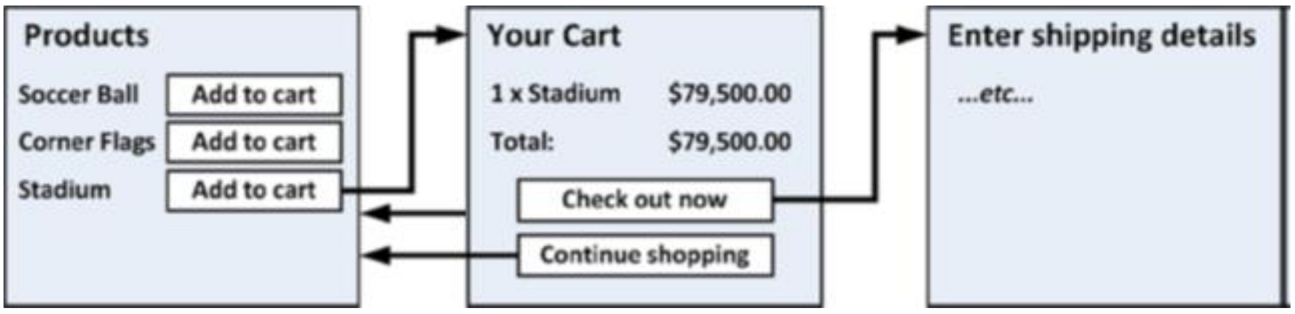
For the `AddToCart` and `RemoveFromCart` methods, we have used parameter names that match the input elements in the HTML forms inserted into the view. This allows the MVC Framework to associate incoming form `POST` variables with those parameters, meaning we do not need to process the form myself.

Remember add `productId` to all products in the repository class

Exercise 5, display the content of the cart (Mandatory)

The final point to note about the `Cart` controller is that both the `AddToCart` and `RemoveFromCart` methods call the `RedirectToAction` method. This has the effect of sending an HTTP redirect instruction to the client browser, asking the browser to request a new URL. In this case, we have asked the browser to request a URL that will call the `Index` action method of the `Cart` controller.

We are going to implement the `Index` method and use it to display the contents of the Cart. If you take a look at this diagram, you will see that this is the workflow when the user clicks the Add to cart button. The users are redirected to the Products page, and later we'll add a View Cart button to the top of the Products to enable users to see the content of the cart.



We need to pass two pieces of information to the view that will display the contents of the cart: the `Cart` object and the URL to display if the user clicks the Continue shopping button. We created a new class file called CartIndexViewModel.cs in the Models folde. The contents of this file are shown here:

```
namespace MbmStore.ViewModels
{
    public class CartIndexViewModel
    {
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; }
    }
}
```

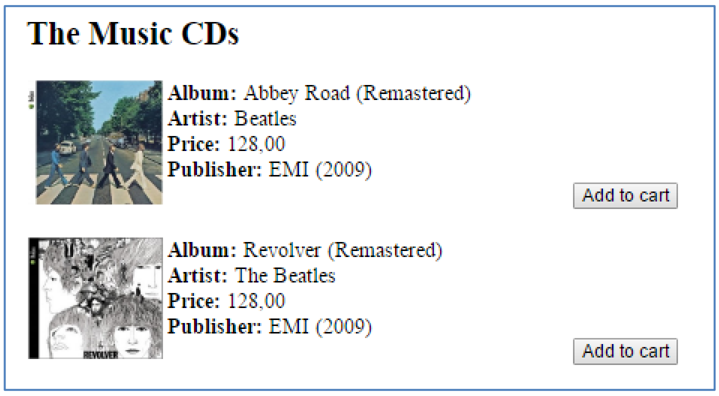
Now that we have the view model, we can implement the `Index` action method in the Cart controller class, as shown here:

```
namespace MbmStore.Controllers
{
    public class CartController : Controller
    {
        private Repository repository;

        // constructor
        // instantiale a new repository object
        public CartController()
        {
            repository = new Repository();
        }


        public ViewResult Index(string returnUrl)
        {
            return View(new CartIndexViewModel
            {
                Cart = GetCart(),
                ReturnUrl = returnUrl
            });
        }
        …
    }
}
```

The last step to display the contents of the cart is to create the new view. Right-click on the Index action method and select **Add View** from the pop-up menu. Set the name to `Index` and click the OK button to create the **Index.cshtml** view file. Edit the view to match the contents shown here:

```
@model MbmStore.ViewModels.CartIndexViewModel
@{
    ViewBag.Title = "MusicBookMovieStore: Your Cart";
}
<h2>Your cart</h2>
<table class="table">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.CartLines)
        {
            <tr>
                <td class="text-center">@line.Quantity</td>
                <td class="text-left">@line.Product.Title</td>
                <td class="text-right">@line.Product.Price.ToString("n")</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Price).ToString("n"))
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</td>
            <td class="text-right">
                @Model.Cart.TotalPrice.ToString("n")
            </td>
        </tr>
    </tfoot>
</table>
<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>
```

The view enumerates the lines in the cart and adds rows for each of them to an HTML table, along with the total cost per line and the total cost for the cart. The CSS classes assigned to the elements correspond to Bootstrap styles for tables and text alignment that we'll use that in an upcoming lesson. We now have the basic functions of the shopping cart in place. First, products are listed along with a button to add them to the cart:


The Music CDs

Album: Abbey Road (Remastered)
Artist: Beatles
Price: 128,00
Publisher: EMI (2009)

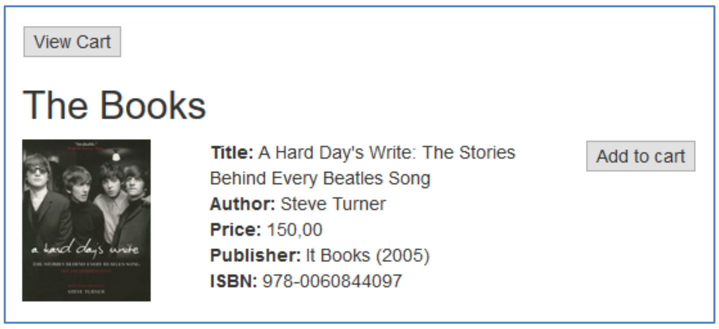
Album: Revolver (Remastered)
Artist: The Beatles
Price: 128,00
Publisher: EMI (2009)

And second, when the user clicks the Add to cart button, the appropriate product is added to their cart and the user returns to the product page they came from.

Open the Views/Catalogue/index.cshtml file and add a button at the top of the page that requests the `Index` method of the `Cart` controller:

```
@using MbmStore.Models;
@model MbmStore.ViewModels.ProductsListViewModel
@{
    ViewBag.Title = "Index";
}

@using (Html.BeginForm("Index", "Cart"))
{
    @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
    <input type="submit" class="btn btn-success" value="View Cart" />
}

@if(Model.Products.OfType<Book>().Count() > 0) {
    <h2>The Books</h2>
    foreach (Book book in Model.Products.OfType<Book>().ToList())
    {
        @Html.Partial("ProductSummaryBook", book);
        <br />
    }
}
```

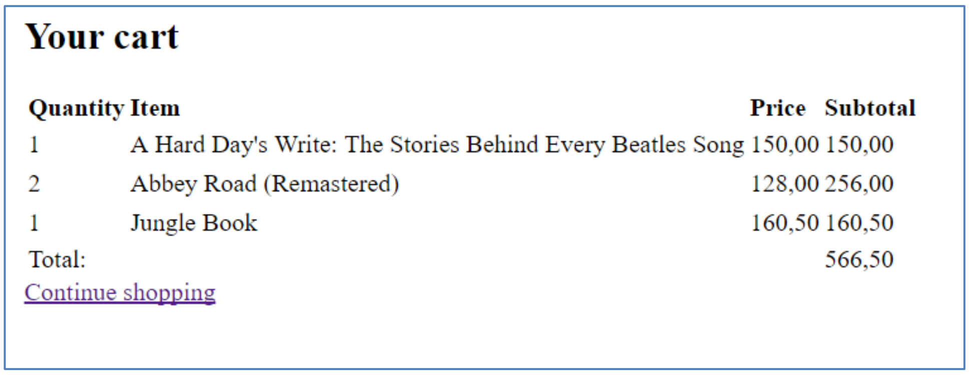
Users can now click the **View Cart** button in the catalogue page:



The Books

Title: A Hard Day's Write: The Stories Behind Every Beatles Song
Author: Steve Turner
Price: 150,00
Publisher: It Books (2005)
ISBN: 978-0060844097

and the cart is displayed:



Your cart

Quantity	Item	Price	Subtotal
1	A Hard Day's Write: The Stories Behind Every Beatles Song	150,00	150,00
2	Abbey Road (Remastered)	128,00	256,00
1	Jungle Book	160,50	160,50
Total:			566,50

Continue shopping

Clicking the Continue shopping button let users return to the product page they came from

Exercise 6, Using Custom Model Binding (Mandatory)

The MVC Framework uses a system called *model binding* to create C# objects from HTTP requests in order to pass them as parameter values to action methods. This is how the MVC framework processes forms, for example: it looks at the parameters of the action method that has been targeted and uses a *model binder* to get the form values sent by the browser and convert them to the type of the parameter with the same name before passing them to the action method.

Model binders can create C# types from any information that is available in the request. This is one of the central features of the MVC Framework. I am going to create a custom model binder to improve the `CartController` class.

The session state feature in the `Cart` controller to store and manage the Cart objects is fine, but the way it is implemented could be better. It does not fit the rest of the application model, which is based around action method parameters, and it also have disadvantages when it comes to unit testing the application. To solve this problem, we are going to create a custom model binder that obtains the `Cart` object contained in the session data. The MVC Framework will then be able to create `Cart` objects and pass them as parameters to the action methods in the `CartController` class. The model binding feature is powerful and flexible. (See Adam Friman: *Pro ASP.NET MVC 5*, chapter 24 for more information).

Creating a Custom Model Binder

We'll create a custom model binder by implementing the `System.Web.Mvc.IModelBinder` interface. To create this implementation, we'll add a new folder in the called **Infrastructure/Binders** and created a `CartModelBinder.cs` class file inside it. The contents of the new file should be like this:

```
namespace MbmStore.Infrastructure.Binders
{
    public class CartModelBinder : IModelBinder
    {
        private const string sessionKey = "Cart";

        public object BindModel(ControllerContext controllerContext, ModelBindingContext 
bindingContext)
        {
            // get the Cart from the session
            Cart cart = null;
            if (controllerContext.HttpContext.Session != null)
            {
                cart = (Cart)controllerContext.HttpContext.Session[sessionKey];
            }
            // create the Cart if there wasn't one in the session data
            if (cart == null)
            {
                cart = new Cart();
                if (controllerContext.HttpContext.Session != null)
                {
                    controllerContext.HttpContext.Session[sessionKey] = cart;
                }
            }

            // return the cart
            return cart;
        }
    }
}
```

The `IModelBinder` interface defines one method: `BindModel`. The two parameters are provided to make creating the domain model object possible. The `ControllerContext` provides access to all the information that the controller class has, which includes details of the request from the client. The `ModelBindingContext` gives you information about the model object you are being asked to build and some tools for making the binding process easier.

For our purposes, the `ControllerContext` class is the one we are interested in. It has an `HttpContext` property, which in turn has a `Session` property that lets us get and set session data. We can obtain the `Cart` object associated with the user's session by reading a value from the session data, and create a `Cart` if there is not one there already.

We need to tell the MVC Framework that it can use the `CartModelBinder` class to create instances of `Cart`. We do this in the `Application_Start` method of **Global.asax**, as shown here below:

```
namespace MbmStore
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            // Add custom model binder
            ModelBinders.Binders.Add(typeof(Cart), new CartModelBinder());
        }
    }
}
```

We can now update the `Cart` controller to remove the `GetCart` method and rely on the model binder to provide the controller with `Cart` objects. The changes are emphasized here:

```
using MbmStore.Infrastructure;
using MbmStore.Models;
using MbmStore.ViewModels;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class CartController : Controller
    {
        private Repository repository;

        // constructor
        // instantiale a new repository object
        public CartController()
        {
            repository = new Repository();
        }


        public ViewResult Index(Cart cart, string returnUrl)
        {
            return View(new CartIndexViewModel
            {
                Cart = cart,
                ReturnUrl = returnUrl
            });
        }


        public RedirectToRouteResult AddToCart(Cart cart, int productId, string returnUrl)
        {
            Product product = repository.Products.FirstOrDefault(p => p.ProductId ==
productId);

            if (product != null)
            {
                cart.AddItem(product, 1);
            }

            return RedirectToAction("Index", new { controller=returnUrl.Substring(1) });
        }


        public RedirectToRouteResult RemoveFromCart(Cart cart, int productId, string
returnUrl)
        {
            Product product = repository.Products
            .FirstOrDefault(p => p.ProductId == productId);
            if (product != null)
            {
                cart.RemoveItem(product);
            }
            return RedirectToAction("Index", new { controller = "Cart" });
        }
    }
}
```

Framework receives a request that requires, say, the `AddToCart` method to be invoked, it begins by looking at the parameters for the action method. It looks at the list of binders available and tries to find one that can create instances of each parameter type. The custom binder is asked to create a `Cart` object, and it does so by working with the session state feature. Between the custom binder and the default binder, the MVC Framework is able to create the set of parameters required to call the action method, allowing us to refactor the controller so that it has no knowledge of how `Cart` objects are created when requests are received.

There are several benefits to using a custom model binder like this. The first is that we have separated the logic used to create a `Cart` from that of the controller, which allows us to change the way we store `Cart` objects without needing to change the controller. The second benefit is that any controller class that works with `Cart` objects can simply declare them as action method parameters and take advantage of the custom model binder. The third benefit is that we can now unit test the Cart controller without needing to mock a lot of ASP.NET plumbing.

Exercise 7, Removing Items from the Cart (Mandatory)

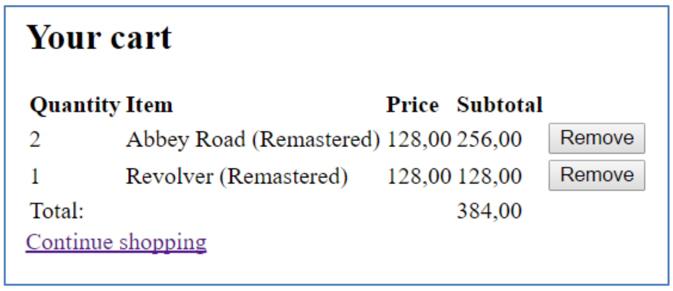
Now that we have implemented the custom model binder, it is time to complete the cart functionality by allowing the customer to remove an item from the cart:

```
@model MbmStore.ViewModels.CartIndexViewModel
@{
    ViewBag.Title = "MusicBookMovieStore: Your Cart";
}
<h2>Your cart</h2>
<table class="table">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines)
        {
            <tr>
                <td class="text-center">@line.Quantity</td>
                <td class="text-left">@line.Product.Title</td>
                <td class="text-right">@line.Product.Price.ToString("n")</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Price).ToString("n"))
                </td>
                <td>
                    @using (Html.BeginForm("RemoveFromCart", "Cart"))
                    {
                        @Html.Hidden("ProductId", line.Product.ProductId)
                        @Html.HiddenFor(x => x.ReturnUrl)
                        <input class="btn btn-sm btn-warning"
                               type="submit" value="Remove" />
                    }
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</td>
            <td class="text-right">
                @Model.Cart.TotalPrice.ToString("n")
            </td>
        </tr>
    </tfoot>
</table>
<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>
```

We added a new column to each row of the table that contains a form with an input element. For the design that we'll implement later, we styled the input element as a button with Bootstrap and added a style element and an id to the table element to ensure that the button and the content of the other columns are properly aligned.

Note We use the strongly typed `Html.HiddenFor` helper method to create a hidden field for the `ReturnUrl` model property, but I had to use the string-based `Html.Hidden` helper to do the same for the `ProductId` field. If we had written `Html.HiddenFor(x => line.Product.ProductID)`, the helper would render a hidden field with the name `line.Product.ProductID`. The name of the field would not match the names of the parameters for the `CartController.RemoveFromCart` action method, which would prevent the default model binders from working, so the MVC Framework would not be able to call the method.

You can see the Remove buttons at work by running the application and adding items to the shopping cart. Remember that the cart already contains the functionality to remove it, which you can test by clicking one of the new buttons:



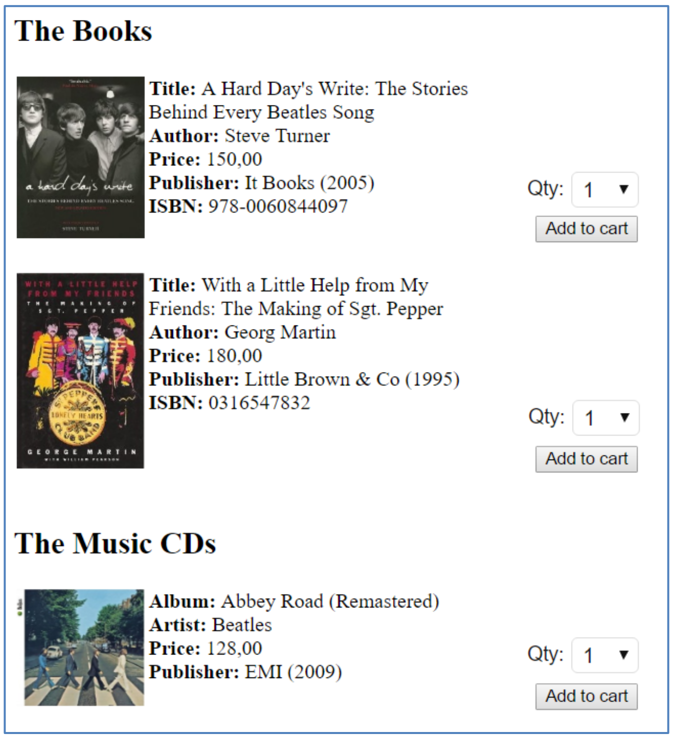
Your cart

Quantity	Item	Price	Subtotal	
2	Abbey Road (Remastered)	128,00	256,00	Remove
1	Revolver (Remastered)	128,00	128,00	Remove
Total:			384,00	

Continue shopping

Exercise 8, Enhance the Cart (optional)

Add a Quantity input dropdown for adding quantity before clicking the **Add to cart** button:



The Books

Title: A Hard Day's Write: The Stories Behind Every Beatles Song
Author: Steve Turner
Price: 150,00
Publisher: It Books (2005)
ISBN: 978-0060844097

Title: With a Little Help from My Friends: The Making of Sgt. Pepper
Author: Georg Martin
Price: 180,00
Publisher: Little Brown & Co (1995)
ISBN: 0316547832

The Music CDs

Album: Abbey Road (Remastered)
Artist: Beatles
Price: 128,00
Publisher: EMI (2009)