When building an MVC e-commerce site, you may want to display product catalog and product detail pages. Product detail pages are pages that display information about a specific product.

This page describes how to display products from the **Products** application in an MVC application connected to Kentico. Regardless of products' position in the product tree, you can display them on your MVC site based on your needs. When creating products, use a product page type labeled as content only.

The **Kentico.Ecommerce** integration package currently supports only products consisting of a page and an SKU object (the default product setting). The stand-alone SKU product mode is not supported in MVC.

To build product detail pages, you need to take care of the following steps:

- Setting URLs of product detail pages
- Displaying product information
- (Optional) Getting URLs of products

Setting URLs of product detail pages

To provide SEO-friendly identifiers in the URL, you can follow the same principle as noted for articles in Providing friendly URLs on MVC sites. Based on that, your product detail pages will be available at *<your domain>/Product/<node ID>/<node alias>*. The node ID and alias are provided from the product's page in Kentico. If you include the node ID, the site displays the correct product to the visitor even when your store manager changes the alias (i.e. name of the product).

Add a new route to your **RouteConfig** class in the **App_Start** folder:

```
            routes.MapRoute(
                name: "Product",
                url: "Product/{id}/{productAlias}",
                defaults: new { controller = "Product", action = "Detail" },
                constraints: new { id = new IntRouteConstraint() }
            );
```

The example assumes that the system displays product detail pages with the *Product* controller and its *Detail* action (as described below in Displaying product information).

Displaying product information

Display all relevant product information to visitors to enable them choose from the offered products properly. Use the **Kentico. Ecommerce** integration package to help you with displaying of product details.

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** project. You can also run the Learning Kit website after connecting the project to a Kentico database.

1. Open your MVC project in Visual Studio.
2. Add a view model for products.

```
        public readonly ProductPrice PriceDetail;
        public readonly string Name;
        public readonly string Description;
        public readonly string ShortDescription;
        public readonly int SKUID;
        public readonly string ImagePath;
        public readonly int ProductPageID;
        public readonly string ProductPageAlias;
        public readonly bool IsInStock;

        /// <summary>
        /// Creates a new product model.
        /// </summary>
        /// <param name="productPage">Product's page.</param>
        /// <param name="priceDetail">Price of the product.</param>
        public ProductViewModel(SKUTreeNode productPage, ProductPrice priceDetail)
        {
            // Fills the page information
            Name = productPage.DocumentName;
            Description = productPage.DocumentSKUDescription;
            ShortDescription = productPage.DocumentSKUShortDescription;
            ProductPageID = productPage.NodeID;
            ProductPageAlias = productPage.NodeAlias;

            // Fills the SKU information
            SKUInfo sku = productPage.SKU;
            SKUID = sku.SKUID;
            ImagePath = sku.SKUImagePath;
            IsInStock = sku.SKUTrackInventory == TrackInventoryTypeEnum.Disabled
||
                        sku.SKUAvailableItems > 0;

            PriceDetail = priceDetail;
        }
```

3. Add a new controller with an action that handles displaying the product detail page. The example also adds a method for retrieving product's pages.

Initialize services and repositories

```
            shoppingService = new ShoppingService();
            pricingService = new PricingService();
            variantRepository = new KenticoVariantRepository();
```

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.

Methods for a basic detail page (only products, not product variants)

```
        /// <summary>
        /// Displays a product detail page of a product specified by ID of the
product's page.
        /// </summary>
        /// <param name="id">Node ID of the product's page.</param>
        /// <param name="productAlias">Node alias of the product's page.</param>
        public ActionResult BasicDetail(int id, string productAlias)
        {
            // Gets the product from Kentico
            SKUTreeNode product = GetProduct(id);

            // If the product is not found or if it is not allowed for sale, 
redirects to error 404
            if ((product == null) || (product.SKU == null) || !product.SKU.
SKUEnabled)
            {
                return HttpNotFound();
            }

            // Redirects if the specified page alias does not match
            if (!string.IsNullOrEmpty(productAlias) && !product.NodeAlias.Equals
(productAlias, StringComparison.InvariantCultureIgnoreCase))
            {
                return RedirectToActionPermanent("Detail", new { id = product.
NodeID, productAlias = product.NodeAlias });
            }

            // Initializes the view model of the product with a calculated price
            ShoppingCart cart = shoppingService.GetCurrentShoppingCart();
            ProductViewModel viewModel = new ProductViewModel(product, 
pricingService.CalculatePrice(product.SKU, cart));

            // Displays the product detail page
            return View(viewModel);
        }

        /// <summary>
        /// Retrieves the product specified by ID of the product's page.
        /// </summary>
        /// <param name="nodeID">Node ID of the product's page.</param>
        private SKUTreeNode GetProduct(int nodeID)
        {
            // Gets the page with the node ID
            TreeNode node = DocumentHelper.GetDocuments()
                            .LatestVersion(false)
                            .Published(true)
                            .OnSite(siteName)
                            .Culture("en-US")
                            .CombineWithDefaultCulture()
                            .WhereEquals("NodeID", nodeID)
                            .FirstOrDefault();

            // If the found page is not a product, returns null
            if (node == null || !node.IsProduct())
            {
                return null;
            }

            // Loads specific fields of the product's product page type from the 
database
            node.MakeComplete(true);

            // Returns the found page as a product page
            return node as SKUTreeNode;
        }
```

4. Add a view that sets the appearance of the product detail page.

```
@model LearningKit.Models.Products.ProductViewModel

<h2>@Model.Name</h2>

<img src="@Url.Kentico().ImageUrl(Model.ImagePath, SizeConstraint.MaxWidthOrHeight
(500))" alt="@Model.Name">

@* Product details *@
<ul>
    <li>
        @Html.Raw(@Model.Description)
    </li>
    <li>
        In stock:
        @if(!Model.IsInStock)
        {
            <span id="stockMessage">Yes</span>
        }
        else
        {
            <span id="stockMessage">No</span>
        }
    </li>
    <li>
        Total price: <span id="totalPrice">@Model.PriceDetail.Currency.FormatPrice
(Model.PriceDetail.Price)</span>
    </li>
</ul>
```

✓ With the **FormatPrice** method from the **Currency** model in the **Kentico.Ecommerce** integration package, you can display a formatted price based on the format string configured for the currency.

```
            decimal price = 5.50M;
            string formattedPrice = shoppingCart.Currency.FormatPrice
(price);
```

Include a button for adding the product to the shopping cart. The button in the example calls the **AddItem** method from the **CheckoutController** implemented in Using a shopping cart on MVC sites.

```
    @* Add to cart button *@
    using (Html.BeginForm("AddItem", "Checkout", FormMethod.Post))
    {
        <input type="hidden" name="itemSkuId" value="@Model.SKUID" />
        <label>Qty</label>
        <input type="text" name="itemUnits" value="1" />
        <input type="submit" name="AddItem" value="Add to cart" />
    }
```

Visitors can now browse through product detail pages and add the products to the shopping cart.

✓ You can access the product page types' fields through generated model classes. See more in Generating classes for Kentico objects.

Getting URLs of products

When placing a link to a product detail page on a page where you have just an SKU object available, you lack the product's page information to build a correct URL (and to redirect to the appropriate controller's action method). Typically in the shopping cart, you have just SKU properties available without knowing its page details and you do not have enough information to link back to product detail pages. To find out the product's URL, i.e. to redirect the product detail page to a correct controller's action method:

1. Open a controller that processes the checkout process in your MVC project in Visual Studio.
2. Add a method that finds out the product's page information from an SKU object.

```
        /// <summary>
        /// Redirects to a product detail page based on the ID of a product's SKU
object.
        /// </summary>
        /// <param name="skuID">ID of the product's SKU object.</param>
        public ActionResult ItemDetail(int skuID)
        {
            // Gets the SKU object
            SKUInfo sku = SKUInfoProvider.GetSKUInfo(skuID);

            // If the SKU does not exist or it is a product option, returns error
404
            if (sku == null || sku.IsProductOption)
            {
                return HttpNotFound();
            }

            // If the SKU is a product variant, uses its parent product's ID
            if (sku.IsProductVariant)
            {
                skuID = sku.SKUParentSKUID;
            }

            // Gets the product's page
            TreeNode node = DocumentHelper.GetDocuments()
                .LatestVersion(false)
                .Published(true)
                .OnSite(siteName)
                .Culture("en-us")
                .CombineWithDefaultCulture()
                .WhereEquals("NodeSKUID", skuID)
                .FirstOrDefault();

            // If no page for the product exists, returns error 404
            if (node == null)
            {
                return HttpNotFound();
            }

            // Redirects to product detail page action method with the product
information
            return RedirectToAction("Detail", "Product", new
            {
                id = node.NodeID,
                productAlias = node.NodeAlias
            });
        }
```

If you do not use product variants on your MVC site, you can move the variant check to the null check or not deal with it at all.

3. Add a link to the **ItemDetail** action method to places where you have only the SKU object available.

```
    @Html.ActionLink(cartItem.Name, "ItemDetail", new { skuId = cartItem.SKUID })
```

OR

```
    <a href="@Url.Action("ItemDetail", new { skuId = cartItem.SKUID })">link text<
/a>
```

The link now correctly builds the product's URL and you can link product detail pages even when you do not have the product's page information available.