When you create a product listing in your MVC application to display a list of offered products, you may want to filter products based on its properties. For example, if you sell notebooks, you may filter them based on their manufacturers, screen resolutions, mobile internet modules and many other attributes.

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.

You can filter different attributes of products. Different processes follows when filtering:

- Based on page properties
- Based on SKU properties of a primitive type or string (typically, Boolean, byte, integer or string variables)
- Based on SKU properties from another database table (typically, public statuses or manufacturers)

**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.

## Filtering based on page properties

Page properties are available in its *TreeNode* object and its coupled data.

To filter products based on page properties:

1. Open your MVC application in Visual Studio.
2. Modify a controller that processes the product listing with a filter:
    a. Initialize the **ShoppingService** and **PricingService** from the **Kentico.Ecommerce** integration package if they are not already.

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

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

    b. Add a POST controller's action that processes the filter information. For example, the following code loads products based on a boolean check box (the *LPTWithFeature* represents a page type's field):

    ```
    // Creates a view model that consists of information
    // whether the 'LPTWithFeature' is selected and a list of
products
    ProductFilterViewModel filteredModel = new
ProductFilterViewModel
    {
        LPTWithFeature = model.LPTWithFeature,
        FilteredProducts = LoadProducts(GetWithFeatureWhereCondition
(model))
    };

    return View(filteredModel);
    ```

    c. Create a where condition that limits the set of products. For example, for a check box (the *GetWithFeatureWhereCondition* method from the previous example):

```
            // Initializes a new where condition
            WhereCondition withFeatureWhere = new WhereCondition();

            // If the feature is selected, sets the where condition
            if (model.LPTWithFeature)
            {
                withFeatureWhere.WhereTrue("LPTWithFeature");
            }
```

   d. Create a method that loads products based on the where condition.
3. Create a view and models based on your filter.

The filter now loads products that fulfill the condition specified by the visitor.

## Filtering based on SKU properties of a primitive type or string

SKU properties are available in the *COM_SKU* database table. In this subsection, you can learn about filtering based on SKU data that carry the full meaning in the database table. For properties that are only foreign keys from a different table, see Filtering based on SKU properties from another database table.

To filter products based on the specified SKU properties:

1. Open your MVC application in Visual Studio.
2. Modify a controller that processes the product listing with a filter:
   a. Initialize the **ShoppingService** and **PricingService** from the **Kentico.Ecommerce** integration package if they are not already.

> We recommend using a dependency injection container to initialize service instances. When configuring the lifetime scope for *ShoppingService* and *PricingService*, create a shared instance (singleton) for all requests.

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

   b. Add a POST controller's action that processes the filter information. For example, the following code loads products based on a price range from–to (the *SKUPrice* column from the product's SKU database table):

```
            // Creates a view model that consists of the entered price range
            // and a list of products
            ProductFilterViewModel filteredModel = new
ProductFilterViewModel
            {
                PriceFrom = model.PriceFrom,
                PriceTo = model.PriceTo,
                FilteredProducts = LoadProducts(GetPriceWhereCondition
(model))
            };

            return View(filteredModel);
```

   c. Create a where condition that limits the set of products. For example, for two text boxes for a from–to range (the *GetPriceWhereCondition* method from the previous example):

```
            // Initializes a new where condition
            WhereCondition priceWhere = new WhereCondition();

            // Sets the price where condition based on the model's values
and limited by the price from-to range
            if (Constrain(model.PriceFrom, model.PriceTo))
            {
                priceWhere.WhereGreaterOrEquals("SKUPrice", model.PriceFrom)
                    .And().WhereLessOrEquals("SKUPrice", model.PriceTo);
            }
```

d. Create a method that loads products based on the where condition.

3. Create a view and models based on your filter.

The filter now loads products that fulfill the condition specified by the visitor.

## Filtering based on SKU properties from another database table

SKU properties are available in the *COM_SKU* database table. In this subsection, you can learn about filtering based on linked SKU data from another database table. For properties that carry the full meaning in the *COM_SKU* table, see Filtering based on SKU properties of a primitive type or string.

To filter products based on the specified linked properties:

1. Open your MVC application in Visual Studio.
2. Modify a controller that processes the product listing with a filter:
   a. Initialize the **ShoppingService** and **PricingService** from the **Kentico.Ecommerce** integration package if they are not already.

> We recommend using a dependency injection container to initialize service instances. When configuring the lifetime scope for *ShoppingService* and *PricingService*, create a shared instance (singleton) for all requests.

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

   b. In the GET controller's action that displays the listing with the filter, you need to get all objects to display the filter. For example, the following code gets all manufacturers' names and loads all products of the product page type:

```
            // Creates a view model that consists of all foreign objects
(manufacturers) related to the products
            // and a list of products that will be filtered
            ProductFilterViewModel model = new ProductFilterViewModel
            {
                Manufacturers = GetManufacturers(),
                FilteredProducts = LoadProducts()
            };

            return View(model);
```

   c. Add a POST controller's action that processes the filter information. For example, the following code loads products based on selected manufacturers' display names):

```
            // Creates a view model that consists of all foreign objects
(manufacturers) related to the products
            // and a list of products that will be filtered with their
selected state
            ProductFilterViewModel filteredModel = new
ProductFilterViewModel
            {
                Manufacturers = model.Manufacturers,
                FilteredProducts = LoadProducts
(GetManufacturersWhereCondition(model))
            };

            return View(filteredModel);
```

d. Create a method that loads the foreign objects. For example, load the objects with an object query.
e. Create a where condition that limits the set of products. For example, for a check box (the *GetManufacturersWhereCondition* method from the previous example):

```
            // Initializes a new where condition
            WhereCondition manufacturersWhere = new WhereCondition();

            // Gets a list of strings representing display names of
selected manufacturers
            List<string> selectedManufacturersIds =
GetSelectedManufacturersIds(model);

            // If any manufacturer is selected, sets the where condition
            if (selectedManufacturersIds.Any())
            {
                manufacturersWhere.WhereIn("SKUManufacturerID",
selectedManufacturersIds);
            }
```

f. Create a method that loads products based on the where condition.

3. Create a view and models based on your filter.

The filter now loads products that fulfill the condition specified by the visitor.