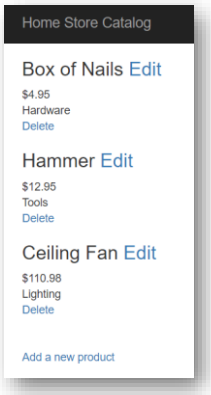
Adding CRUD Operations to HomeStore

Create, Update, Delete



Download HomeStoreWithDatabase.zip from Canvas, unzip, and open.

Let's add some editing capabilities.

First, add a link to the List view for each product.

This will cause an integer value to be sent into the Edit method.

```
@foreach (var p in Model)
{
    <div>
        <h3>@p.Name @Html.ActionLink("Edit", "Edit", new { p.ProductID })</h3>
        @p.Price.ToString("c")
        <br />
        @p.Category.CategoryName
    </div>
}
```

Add an Edit Method to the controller.

Will return the first record with that particular ProductID. If there is no record with the ProductID, it won't cause an error when the "OrDefault" is present.

```
public ActionResult Edit(int productId)
{
    Product product = repository.Products.FirstOrDefault(p => p.ProductID == productId);
    ViewBag.CategoryID = new SelectList(repository.Categories, "CategoryID", "CategoryName", product.CategoryID);
    return View(product);
}
```

This will create a list that can be used on the Category dropdown box on the Edit form. The list is saved in the ViewBag.

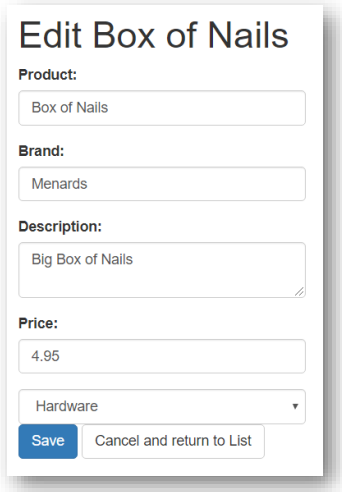
Create Edit view.

```
@model HomeStore.Models.Product
@{
    ViewBag.Title = "Edit";
}
<div>
    <h1> Edit @Model.Name</h1>

    @using (Html.BeginForm())
    {
        <div>

          @Html.HiddenFor(m => m.ProductID)
            <div class="form-group">
                <label>Product:</label>
                @Html.TextBoxFor(x => x.Name, new { @class = "form-control" })
            </div>
            <div class="form-group">
                <label>Brand:</label>
                @Html.TextBoxFor(x => x.Brand, new { @class = "form-control" })
            </div>
            <div class="form-group">
                <label>Description:</label>
                @Html.TextAreaFor(x => x.Description, new { @class = "form-control" })
            </div>
            <div class="form-group">
                <label>Price:</label>
                @Html.TextBoxFor(x => x.Price, new { @class = "form-control" })
            </div>
            <!–See the comments on the next slide -->
            @Html.DropDownList("CategoryID", null, htmlAttributes: new { @class = "form-control" })
        </div>
        <div>
        <br /><br />
            <input type="submit" value="Save" class="btn btn-primary" />
            @Html.ActionLink("Cancel and return to List", "List", null, new { @class = "btn btn-default" })
        </div>
    }
</div>
```

We need this line so that the ProductID, even though not shown on the form, will still be passed on when the form is submitted.



Remember that we named the selectlist that we saved in the ViewBag CategoryID which didn't make a lot of sense since this is a list, not an integer representing the category..

```
ViewBag.CategoryID = new SelectList(repository.Categories, "CategoryID", "CategoryName", product.CategoryID);
```

```
<!--This below is a strange overload for DropDownList.  If we put null for selectlist (the second parameter),
 the method will look for a selectlist in the ViewBag property with the name of the first parameter.
 However, this method also needs to know what property of the product object should be bound to this
 drop-down box. So the first parameter also tells the method what property to bind to the dropdown.
 This requires the name of the ViewBag property to be the name of the property to bind.
 So we need to name the ViewBag property CategoryID. This seems strange since in one case, CategoryID represents
 an integer and in the other case, an entire selectlist. But because of the way the overload works, this is
 what we have to do. -->
    @Html.DropDownList("CategoryID", null, htmlAttributes: new { @class = "form-control" })
```

A selectlist would normally go here. But null tells the method to look for a property of the ViewBag with the name matching the string in the first parameter which is CategoryID.

However, this method also needs to know what property should be bound to the drop-down box. The method *looks again at the first parameter* to find that out.

So we need to have named our ViewBag property with CategoryID since that's what we want to bind to the drop-down box.

Since we are having a TextArea for Description, let's allow multiple lines.

Add the using statement to the top:

```
using System.ComponentModel.DataAnnotations;
```

```
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
    [DataType(DataType.MultilineText)]
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

Add SaveProduct method to IProductRepository and EFProductRepository.

```
public interface IProductRepository
{
    IEnumerable<Product> Products { get; }
    void SaveProduct(Product product);
}
```

In EFProductRepository

```
public void SaveProduct(Product product)
{
        Product dbEntry = context.Products.Find(product.ProductID);
        if (dbEntry != null)
        {
            dbEntry.Name = product.Name;
            dbEntry.Description = product.Description;
            dbEntry.Price = product.Price;
            dbEntry.Category = product.Category;
        }

    context.SaveChanges();
}
```

Add an overload of the Edit method to handle POST requests.

```
[HttpPost]
public ActionResult Edit(Product product)
{
    if (ModelState.IsValid)
    {
        repository.SaveProduct(product);
        TempData["message"] = string.Format("{0} has been saved", product.Name);
        return RedirectToAction("List");  ← Invokes the List method.
    }
    else
    {
        // there is something wrong with the data values
        return View(product);
    }
}
```

TempData is like the ViewBag. But we cannot use ViewBag in this situation because the user is being redirected. ViewBag passes data between the controller and view, and it cannot hold data for longer than the current HTTP request.

Add a message to the layout above @RenderBody().

```
@if (TempData["message"] != null)
{
    <div class="alert alert-success">@TempData["message"]</div>
}

@RenderBody()
```

Let's try it out!

Creating New Products

Create a link at the bottom of the List view.

```
<div>
    <br /><br />
    @Html.ActionLink("Add a new product", "Create")
</div>
```

Create a Create Action Method.

```
public ActionResult Create()
{
    ViewBag.CategoryID = new SelectList(repository.Categories, "CategoryID", "CategoryName");
    return View("Edit", new Product());
}
```

The Create method does not render its default view. Instead, it specifies that the Edit view should be used. The Edit View will be sent a new empty Product object.

We won't need a Create View since the Create method displays the Edit view with an empty product.

We want the form to be posted back to the Edit action so that we can save the newly created product data.

First change Edit View.

```
@using (Html.BeginForm())
@using (Html.BeginForm("Edit", "Catalog"))
```

Now the form will always be posted to the Edit action, regardless of which action rendered it.

If we don't change this, the Create method will render the Edit view but then the form will try to post back to the Create action.

Now that this form will be used for both editing and adding, we'll add this if statement in place of `Edit @Model.Name`.

```
@if (@Model.Name == null)
{
    <h1>Add Product</h1>
}
else
{
    <h1> Edit @Model.Name</h1>
}
```

I'm saying to run it now in the video!

We also need to add something to SaveProduct so that we can use that method for adding as well as editing an object.

In EFProductRepository

```
public void SaveProduct(Product product)
{
    if (product.ProductID == 0)
    {
        context.Products.Add(product);
    }
    else
    {
        Product dbEntry = context.Products.Find(product.ProductID);
        if (dbEntry != null)
        {
            dbEntry.Name = product.Name;
            dbEntry.Description = product.Description;
            dbEntry.Price = product.Price;
            dbEntry.Category = product.Category;
        }
    }
    context.SaveChanges();
}
```

Let's try it out!

Deleting Products

Add a "Delete" link for each product on the List view.

```
@foreach (var p in Model)
{
    <div>
        <h3>@p.Name @Html.ActionLink("Edit", "Edit", new { p.ProductID })</h3>
        @p.Price.ToString("c")
        <br />
        @p.Category.CategoryName
        <br />
        @Html.ActionLink("Delete", "Delete", new { p.ProductID })
    </div>
}
```

Add DeleteProduct method to IProductRepository and EFProductRepository.

```
public interface IProductRepository
{
    IEnumerable<Product> Products { get; }
    void SaveProduct(Product product);
    Product DeleteProduct(int productID);
}
```

In EFProductRepository:

```
public Product DeleteProduct(int productID)
{
    Product dbEntry = context.Products.Find(productID);
    if (dbEntry != null)
       {
              context.Products.Remove(dbEntry);
              context.SaveChanges();
       }
    return dbEntry;
}
```

Why are we returning the product that was just deleted?
We'll see why on the next slide.

Create the Delete Action Method

```
public ActionResult Delete(int productId)
{
    Product deletedProduct = repository.DeleteProduct(productId);
    if (deletedProduct != null)
    {
        TempData["message"] = string.Format("{0} was deleted", deletedProduct.Name);
    }
    return RedirectToAction("List");
}
```

Try it out!