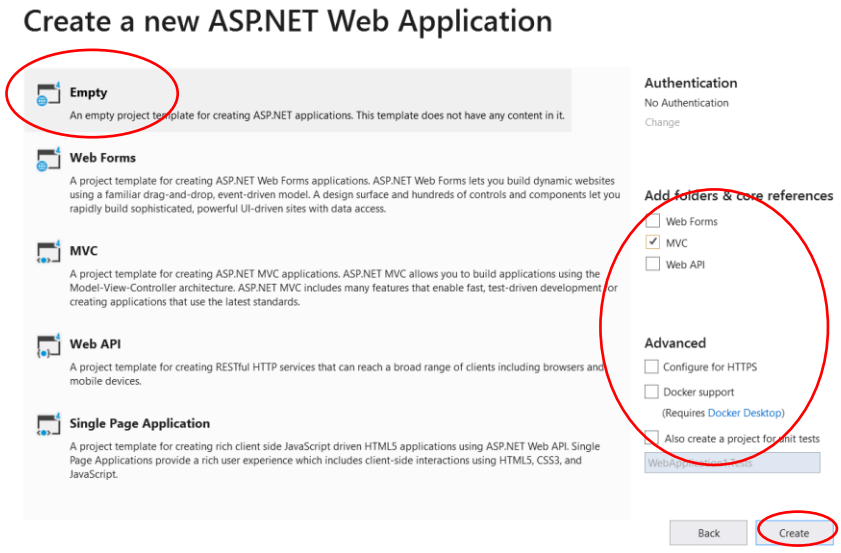
Setting Up Our Home Store Application With a Mock Database

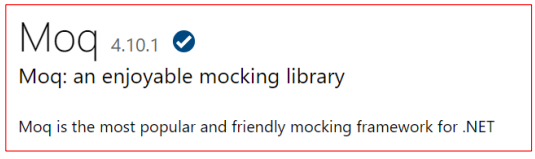
First, let's set up a new project called HomeStore.



Tools > NuGet Package Manager

Install **Ninject.MVC5** and use **version 3.2.1** instead of the latest stable version.

Install **Moq** so that we can use a mock database at first.



Finally, install **EntityFramework**.

Setting Up the DI Container

Create an Infrastructure folder in the create NinjectDependencyResolver.cs.

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using HomeStore.Models;
using Ninject;
namespace HomeStore.Infrastructure
{
  public class NinjectDependencyResolver : IDependencyResolver
  {
    private IKernel kernel;
    public NinjectDependencyResolver(IKernel kernelParam)
    {
      kernel = kernelParam;
      AddBindings();
    }
    public object GetService(Type serviceType)
    {
      return kernel.TryGet(serviceType);
    }
    public IEnumerable<object> GetServices(Type serviceType)
    {
      return kernel.GetAll(serviceType);
    }
    private void AddBindings()
    {
      // put bindings here
    }
  }
}
```

Now add this line to NinjectWebCommon.cs in the App_Start folder:

```
System.Web.Mvc.DependencyResolver.SetResolver(new
Infrastructure.NinjectDependencyResolver(kernel));
```

Add a class under Models called Product.cs.

```
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
    public string Brand { get; set; }
    public string Description { get; set; }
    public string ImagePath { get; set; }
    public decimal Price { get; set; }
    public int CategoryID { get; set; }
    public virtual Category Category { get; set; }
}
```

Add a class under Models called Category.cs.

```
public class Category
{
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

Creating an Abstract Repository

We need some way of getting Product entities from a database.

> We want to keep a **degree of separation** between the data model entities and the storage and retrieval logic which can be done using the ***repository pattern***.

We won't worry about how to connect to the database yet, but we will create an interface that will hold a list of Product objects and Category objects.

Create a new interface under Models called **IProductRepository.cs**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HomeStore.Models
{
    public interface IProductRepository
    {
        IEnumerable<Product> Products { get; }
        IEnumerable<Category> Categories { get; }

    }
}
```

Be sure that the interface is public!

A class that depends on the IProductRepository interface can obtain Product objects and Category objects ***without needing to know anything*** about where they are coming from or how the implementatior class will deliver them.

This is the essence of the **repository pattern**.

Making a Mock Repository

We are going to create a **mock implementation** of the IProductRepository interface that will stand in until we get a database set up.

Put the following code into AddBindings in NinjectDependencyResolver.

```
private void AddBindings()
{
    Mock<IProductRepository> mock = new Mock<IProductRepository>();

    mock.Setup(m => m.Products).Returns(new List<Product> {
        new Product { Name = "Ceiling Fan", Price = 129.99M},
        new Product { Name = "Hammer", Price = 14.99M},
        new Product { Name = "Box of Nails", Price = 5.99M}
        });

    kernel.Bind<IProductRepository>().ToConstant(mock.Object);
}
```

You'll need to put a using statement in for Moq at the top:

```
using Moq;
```

Add a controller called CatalogController. Give it a constructor that will accept a repository object from Ninject when the application is run.

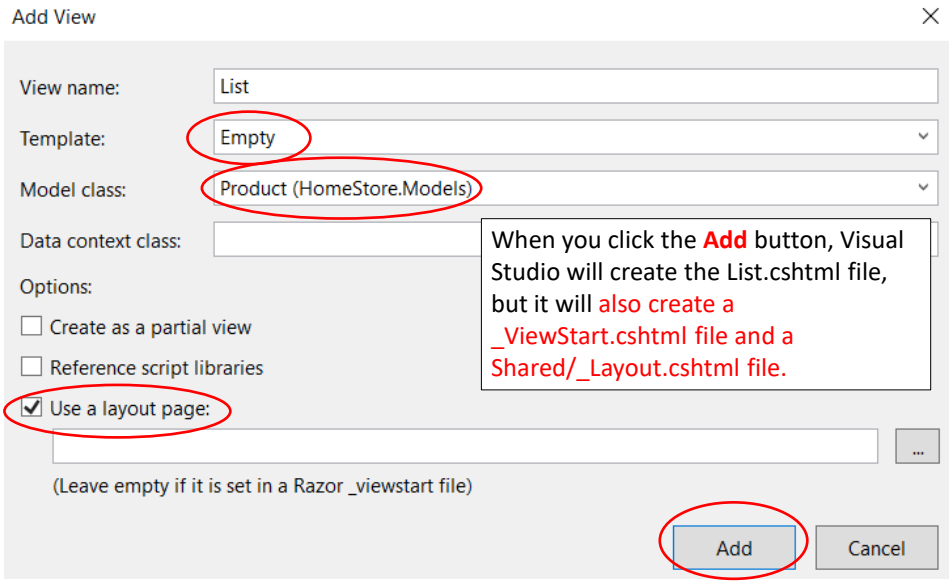
```
using HomeStore.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace HomeStore.Controllers
{
    public class CatalogController : Controller
    {
        // GET: Catalog
        IProductRepository repository;

        public CatalogController(IProductRepository productRepository)
        {
            repository = productRepository;
        }
        public ActionResult List()
        {
            return View(repository.Products);
        }
    }
}
```

Give the controller an action method called List that will pass the list of products from the repository on to a view called List.

Adding the View, ViewStart file, and the Layout



Keep what is in _Layout.cshtml but change the first three parameters in the ActionLink method.

```
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
        @Html.ActionLink("Home Store Catalog", "List", "Catalog", new { area = "" },
</div>
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
```

Put this in List.cshtml.

```
@model IEnumerable<HomeStore.Models.Product>

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model)
{
<div>
    <h3>@p.Name</h3>
    @p.Price.ToString("c")
    <br />
</div>
}
```

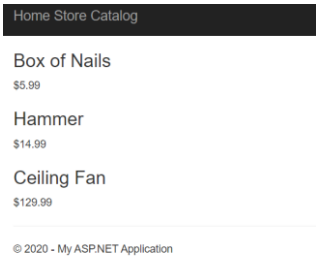
Note that List will be a strongly-typed view that is bound to a list of Product objects.

← This will put the Price into currency format.

Let's set the *default route* in RouteConfig.cs to use the Catalog controller and the List action method.

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Catalog", action = "List", id = UrlPar
);
```

Run the application.



Now it's time to create and access a real database.