

# Lab: ASP.NET MVC Introduction

This document defines several walkthroughs for creating ASP.NET MVC-based apps, from setting up the framework to implementing the fully functional applications.

## Non-Data-Driven Apps

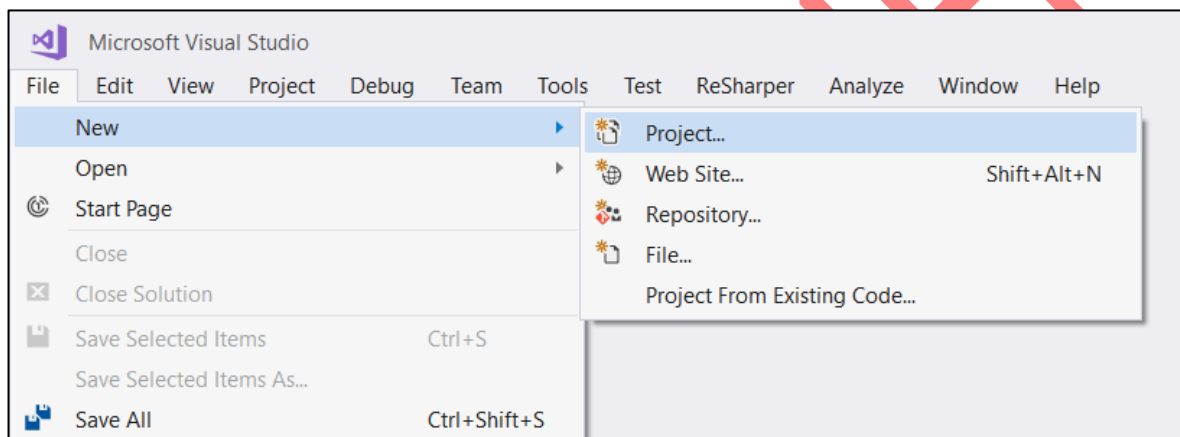
These are apps, which do not need a database to work.

### Numbers from 1-50

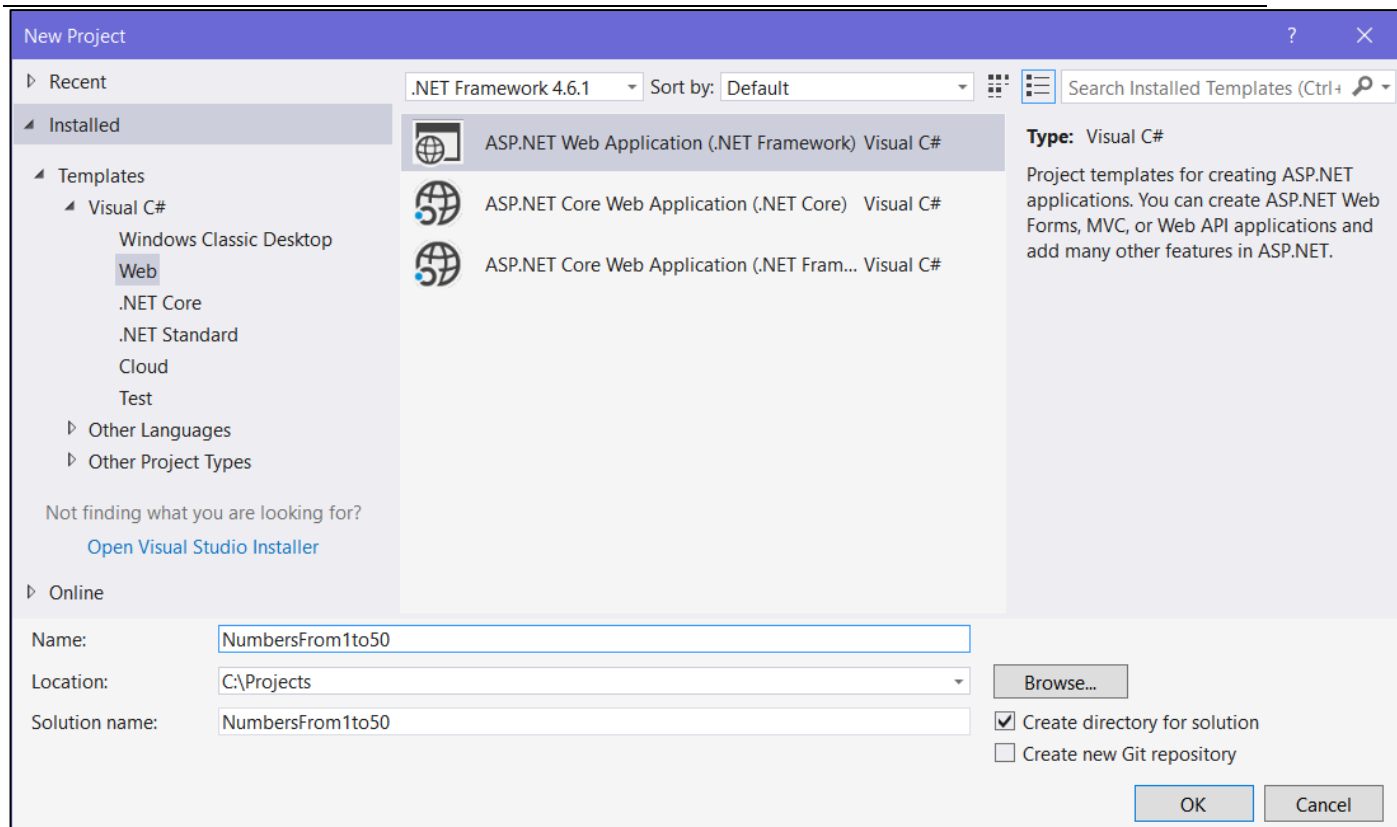
Create an MVC application, which **prints** the numbers from **1** to **50** inside a view.

#### Create New Project

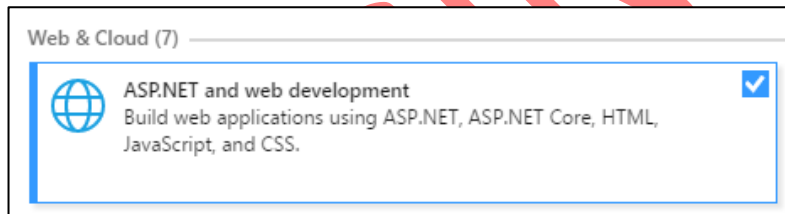
Let's create a new project. Open Visual Studio and click on **[File] → [New] → [Project]**:



Next, select **[Templates] → [Visual C#] → [Web]**:



If you don't have "Web" in the selection, click **[Open Visual Studio Installer]** and install the **"ASP.NET and web development"** component:



Next, choose the **MVC** template. After that, make sure to change **authentication type** to **"No authentication"**:

New ASP.NET Web Application - NumbersFrom1to50

**ASP.NET 4.6.1 Templates**

Empty Web Forms **MVC** Web API Single Page Application

Azure API App

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

[Learn more](#)

Change Authentication

Authentication: **No Authentication**

Add folders and core references for:

☐ Web Forms ☒ MVC ☐ Web API

☐ Add unit tests

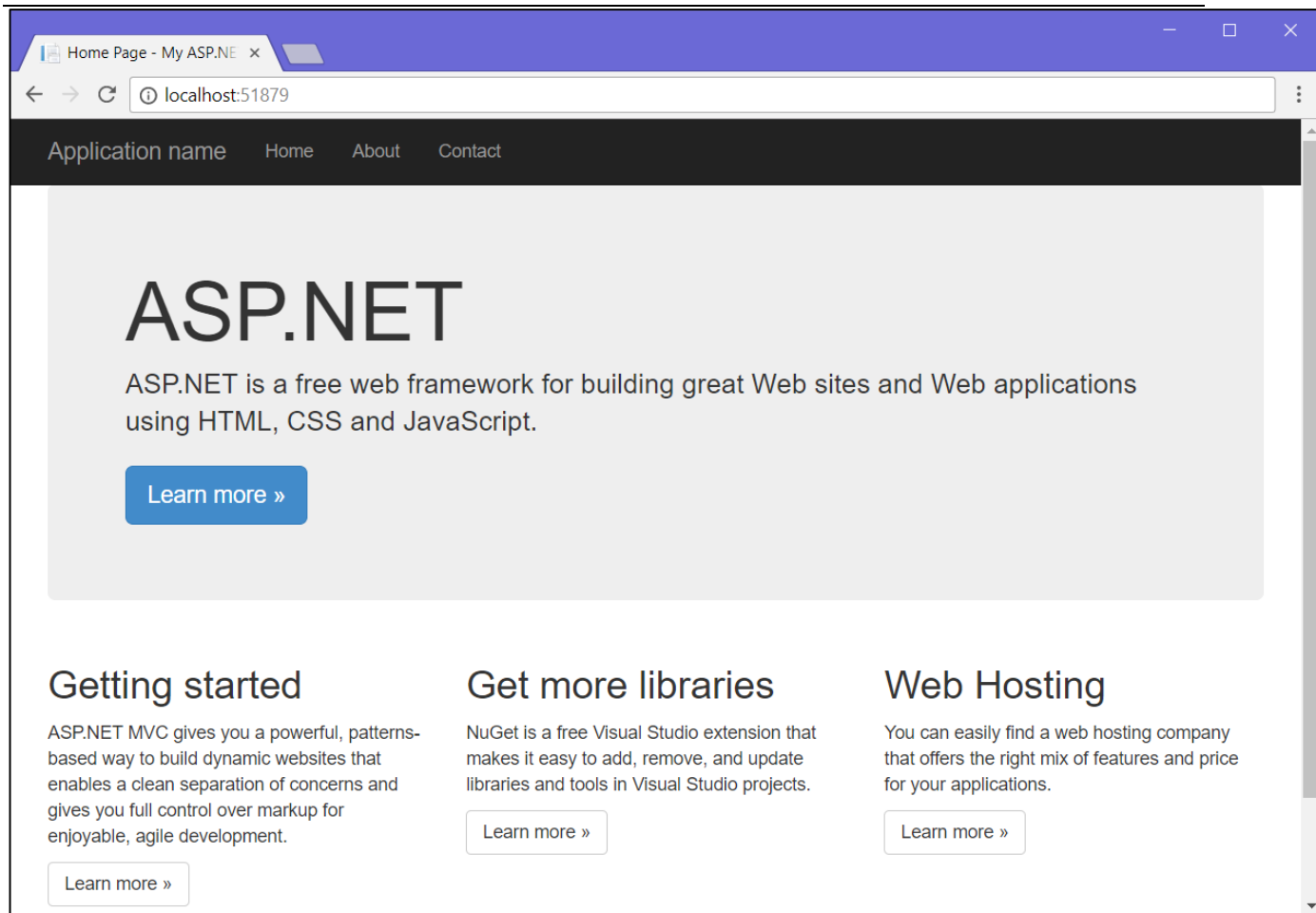
Test project name: NumbersFrom1to50.Tests

OK Cancel

At this point, we should have the project created and in front of us.

Run the Project

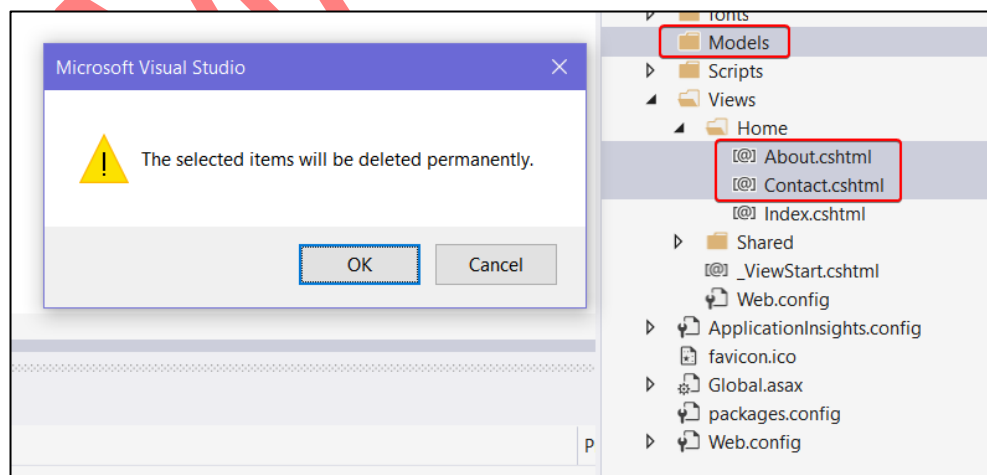
Let's run it with **[Ctrl+F5]** and see what we have:



As we can see, we don't really need that big panel about ASP.NET and those little sections at the bottom, so let's remove them!

#### Remove Unnecessary Views

Now that Visual Studio created the project, we need to remove some views we're sure **won't be needed**, like **Contact.cshtml** and **About.cshtml** views. Go into **solution explorer** and delete them, as well as the **Models** folder:



## Update Layout View

Now that we've deleted the unnecessary views, it's time to go into the **"Views/Shared/\_Layout.cshtml"** file and **edit** it a bit. For starters, let's **update the header link**, which usually says **"Application Name"** to something nicer:

```
@Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar"

```



```
@Html.ActionLink("Numbers from 1-50", "Index", "Home", new { area = "" }, new { @class = "navb

```

Next, let's remove those unneeded **menu items** in the **header**:

```
21 <div class="navbar-collapse collapse">
22 <ul class="nav navbar-nav">
23 <li>@Html.ActionLink("Home", "Index", "Home")</li>
24 <li>@Html.ActionLink("About", "About", "Home")</li>
25 <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
26 </ul>
27 </div>
```

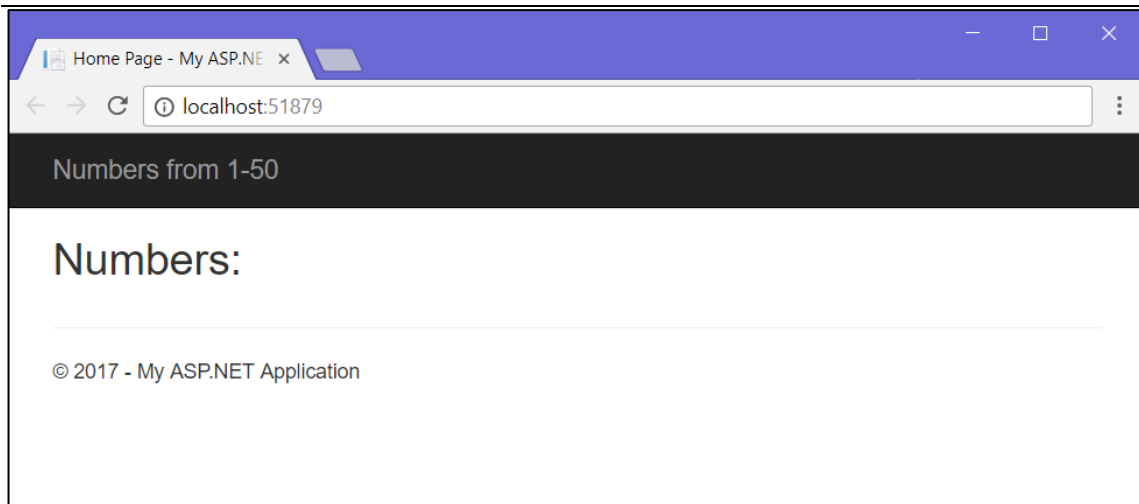
## Update Index View

Let's go in **"Views/Index.cshtml"** and remove everything unneeded, until all we're left with is this:

```
1 @{}
2 ViewBag.Title = "Home Page";
3 }
4
5 <div class="row">
6 <div class="col-md-4">
7 <h2>Numbers:</h2>
8
9 </div>
10 </div>
```

## Run the Project

Let's see what we've got so far:



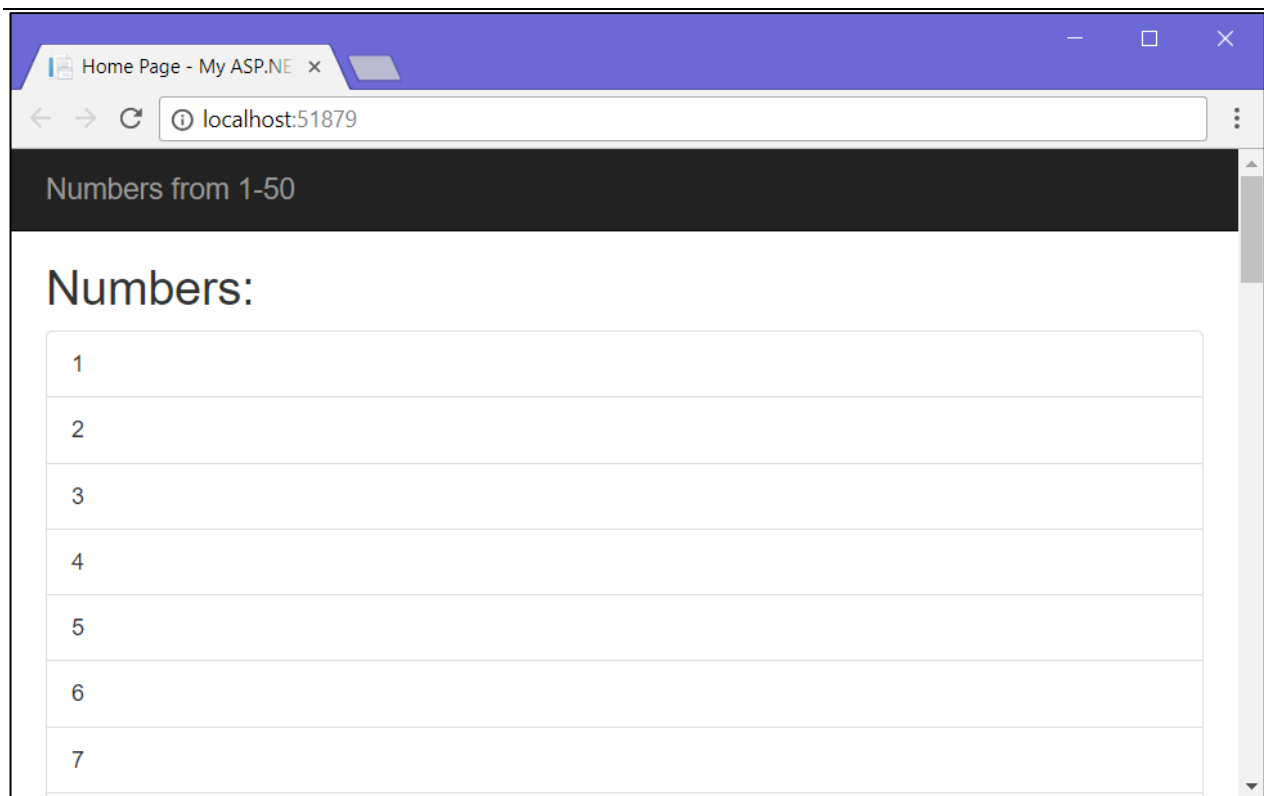
Looks good, but we need to print the numbers.

Write List Logic

Let's go back into the **Index.cshtml** file and create an **unordered list** with a simple **for-loop** inside it:

```
1  @{
2      ViewBag.Title = "Home Page";
3  }
4
5  <div class="row">
6      <div class="col-md-4">
7          <h2>Numbers:</h2>
8          <ul class="list-group">
9              @for (int i = 1; i <= 50; i++)
10             {
11                 <li class="list-group-item">@i</li>
12             }
13          </ul>
14      </div>
15  </div>
```

Refresh the page again and the result should show up:



Looks like it works!

## Data-Driven Apps

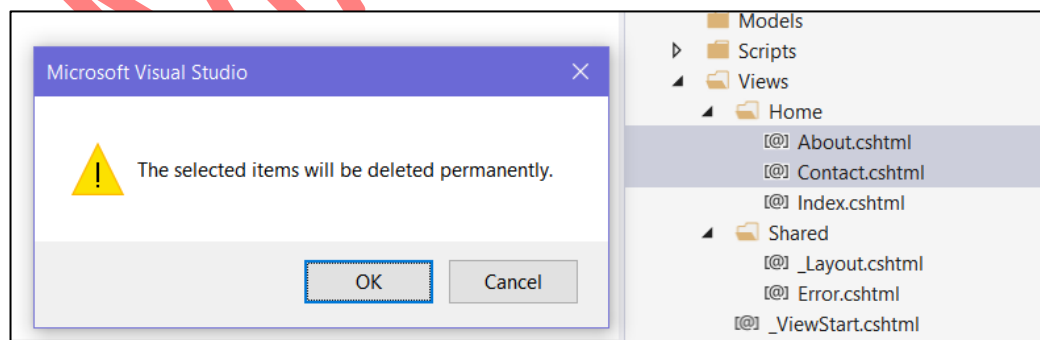
These are apps, which need to store data in a database to work properly.

### TODO List

Create a TODO list application, which keeps track of a person's **tasks** inside a **database**. The application should support **creating** tasks and **deleting** tasks.

### Prepare a New Project

Just as before, create a new project and name it **TODOList**. After you create it, we need to remove any unnecessary views yet again:



After that, go into the **Controllers/HomeController.cs** file and **remove every action except the Index() action**:

```

7 namespace TODOList.Controllers
8 {
9     public class HomeController : Controller
10    {
11        public ActionResult Index()
12        {
13            return View();
14        }
15
16        public ActionResult About()
17        {
18            ViewBag.Message = "Your application description page.";
19
20            return View();
21        }
22
23        public ActionResult Contact()
24        {
25            ViewBag.Message = "Your contact page.";
26
27            return View();
28        }
29    }
30 }

```

After that, this is what the contents of the **Home controller** should look like:

```

1 using System.Web.Mvc;
2
3 namespace TODOList.Controllers
4 {
5     public class HomeController : Controller
6     {
7         public ActionResult Index()
8         {
9             return View();
10        }
11    }
12 }

```

Next, let's go into the **/Views/Shared/\_Layout.cshtml** and **change our application name**. Before leaving, we can also remove the **Home, About and Contact** menu items as well.

At this point, our **\_Layout.cshtml** file should look like this:



```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="utf-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>@ViewBag.Title - My ASP.NET Application</title>
7      @Styles.Render("~/Content/css")
8      @Scripts.Render("~/bundles/modernizr")
9  </head>
10 <body>
11     <div class="navbar navbar-inverse navbar-fixed-top">
12         <div class="container">
13             <div class="navbar-header">
14                 <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
15                     <span class="icon-bar"></span>
16                     <span class="icon-bar"></span>
17                     <span class="icon-bar"></span>
18                 </button>
19                 @Html.ActionLink("TODO List", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
20             </div>
21         </div>
22     </div>
23     <div class="container body-content">
24         @RenderBody()
25         <hr />
26         <footer>
27             <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
28         </footer>
29     </div>
30
31     @Scripts.Render("~/bundles/jquery")
32     @Scripts.Render("~/bundles/bootstrap")
33     @RenderSection("scripts", required: false)
34 </body>
35 </html>

```

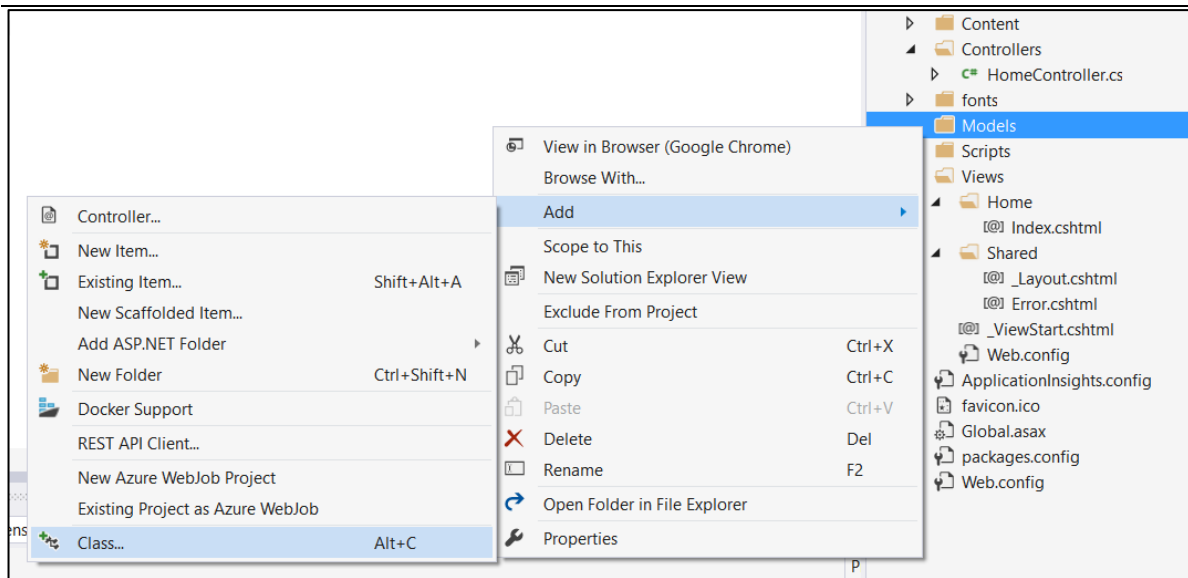
We've removed all the unnecessary stuff in our project and we are ready to actually start writing code!

### Create Task Model

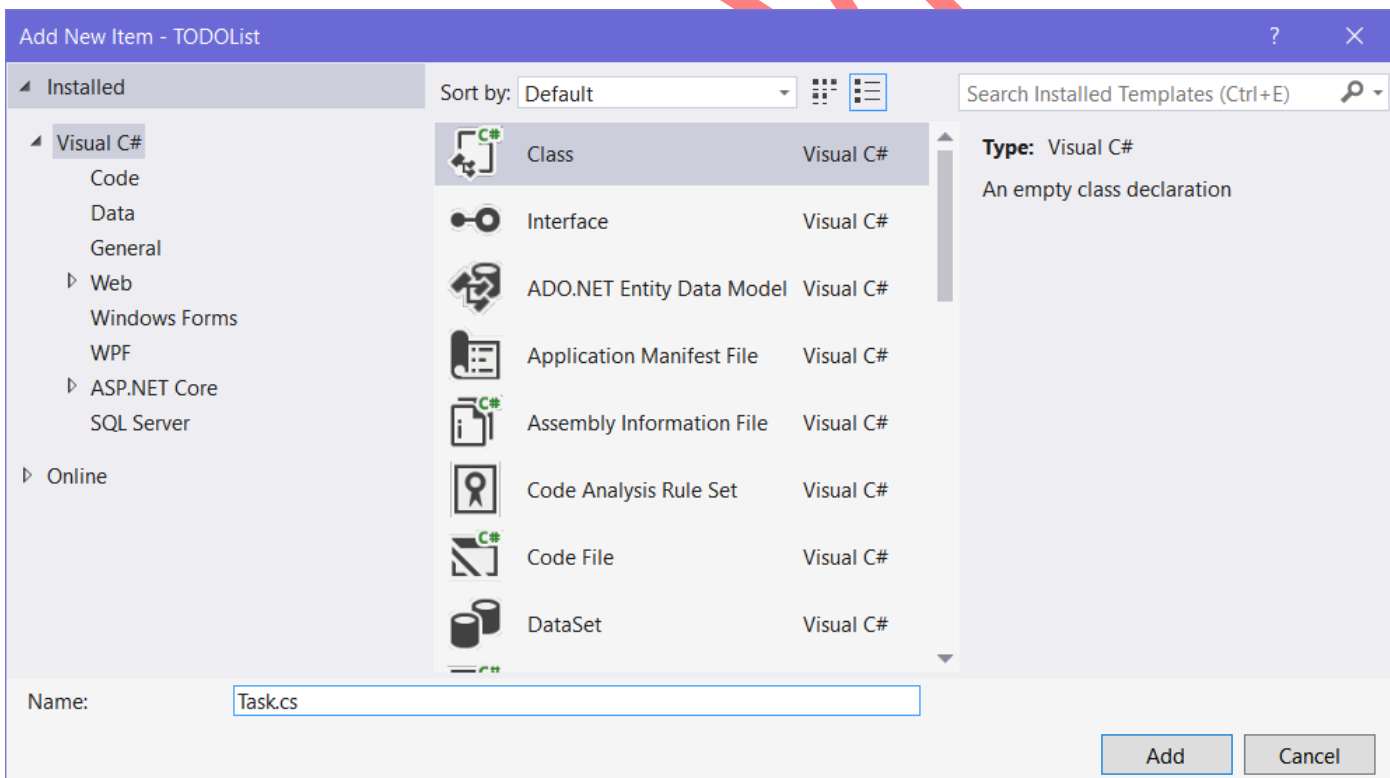
Now it's time to create our **task entity class**. Our **task** will be simple. It will have **2 properties**:

- **Id** – a unique **integer**, with which to differentiate tasks from one another.
- **Title** – the **title** of the task, stored as a **string**.

Let's go in our **Models** folder and **add a Task class**:



In the menu, which popped up, select **Class** and name it **Task.cs**:



All that's left is to **add the properties** into our new file:

```
using System.ComponentModel.DataAnnotations;

namespace TODOList.Models
{
    public class Task
    {
        public int Id { get; set; }

        [Required]
        public string Title { get; set; }
    }
}
```

We're using the **[Required]** attribute on our **Title** property, because we don't want to have **tasks without a title**.

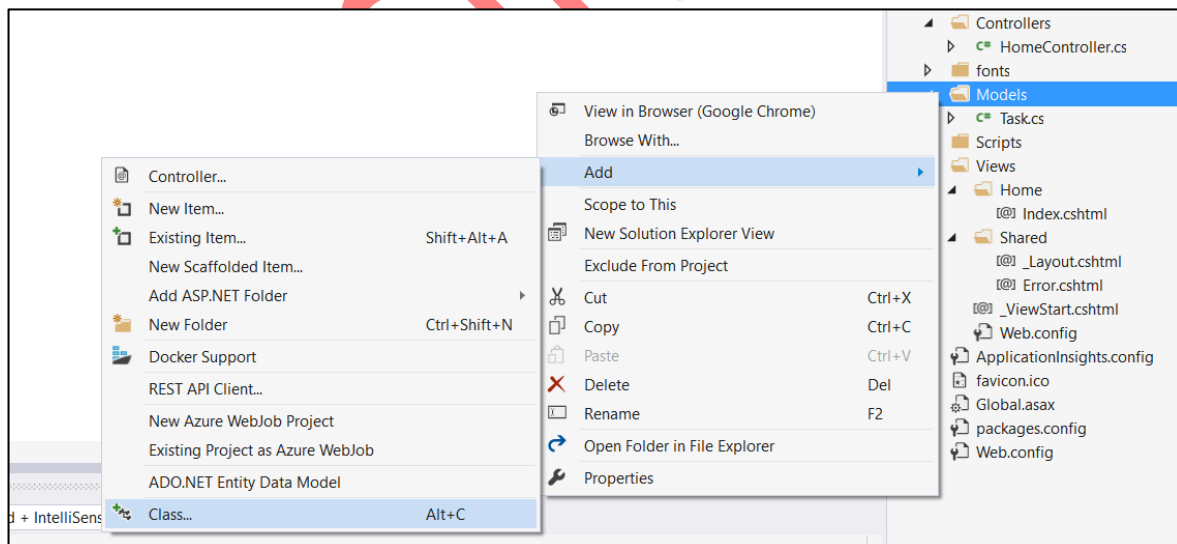
You might have noticed that this **looks** a lot like a **standard C# OOP class**. That's because it's exactly that! Entity Framework works with ordinary classes to achieve its **object-relational mapping**.

We're done here. Let's move on to creating the **database context**.

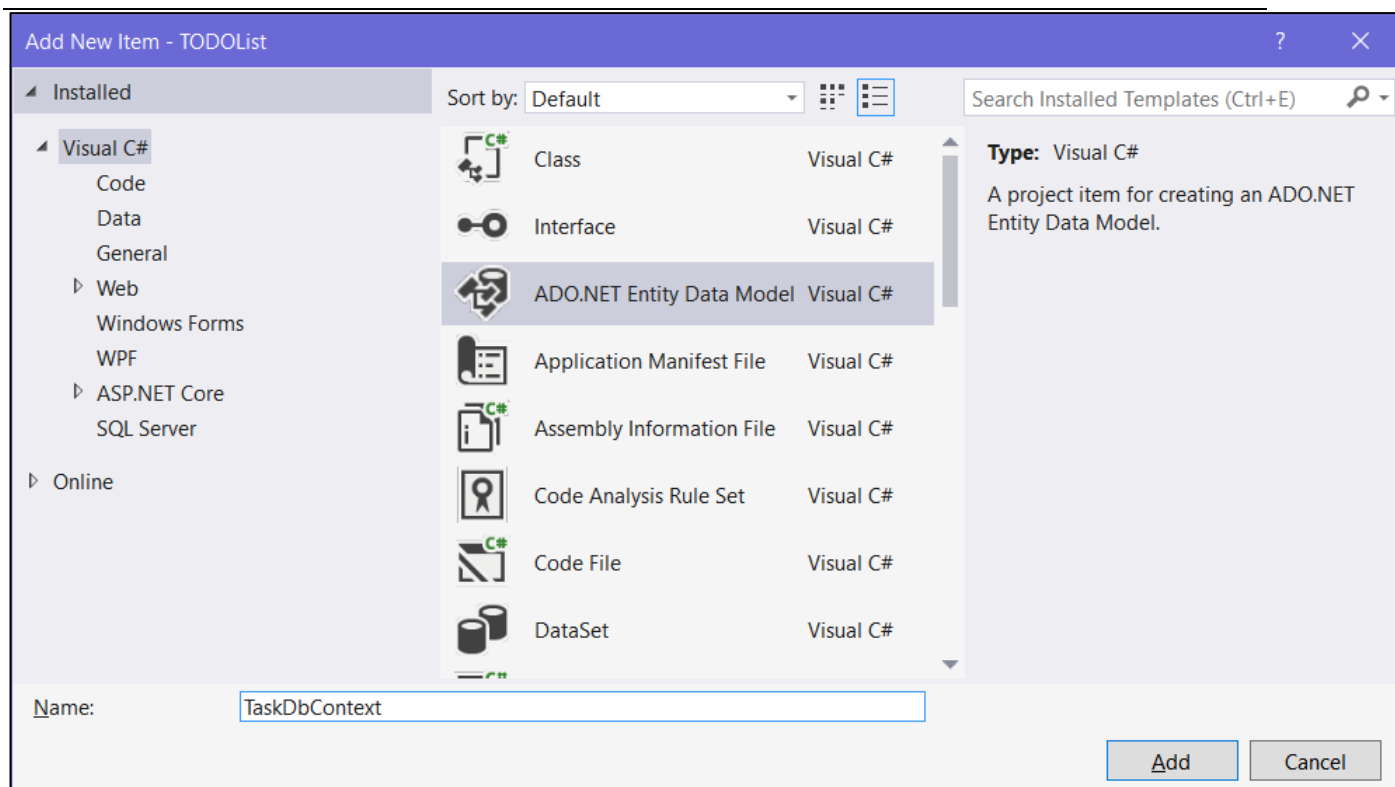
#### Create Database Context

Now, it's time to create our app's **Database Context**. The **database context** is something the **Entity Framework ORM** uses to **communicate** with the **database**. It saves us from writing database queries manually! Let's make one.

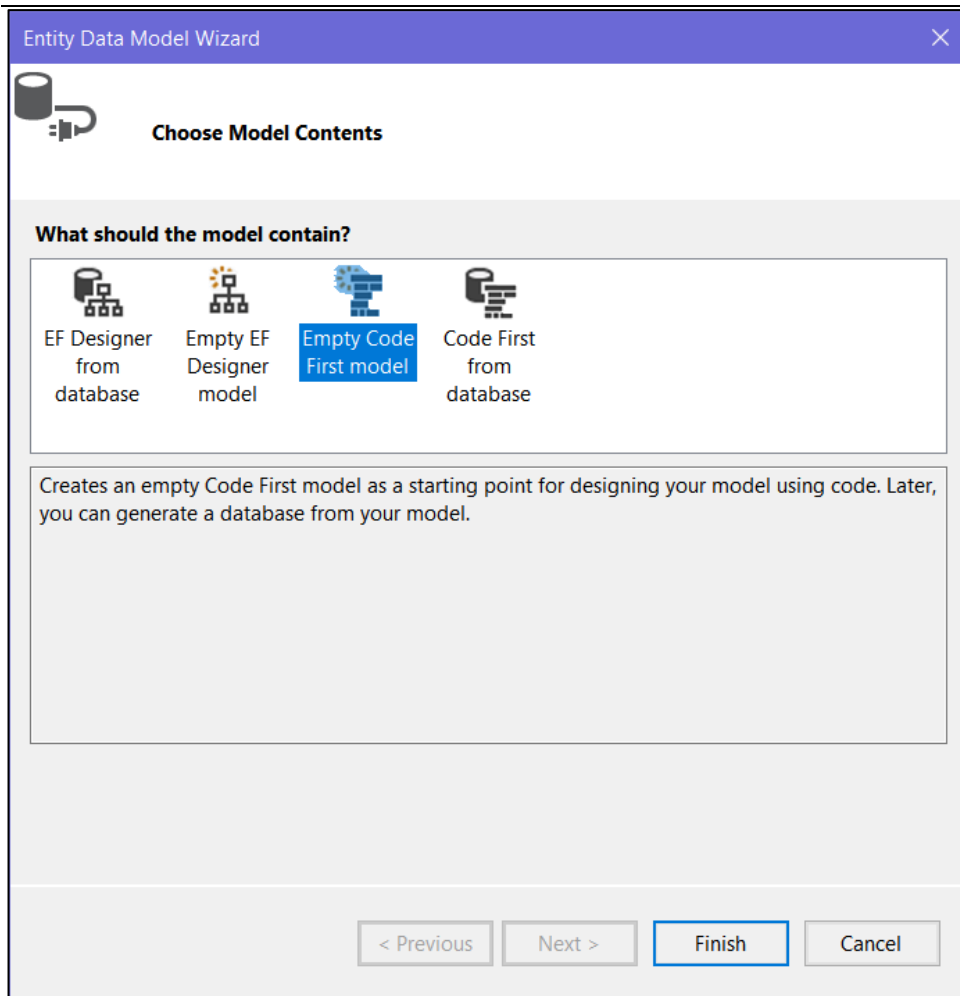
Right click on the **Models** folder and add a new **class**:



From the **Add New Item** menu, choose “**ADO.NET Entity Data Model**” and name it **TaskDbContext**:



After that, a window asking us what kind of data model we want will pop up. Choose **Empty Code First Model**:



In ASP.NET, we get to choose whether we want the **database tables + relations** to be **generated** from the **entity classes**, or for our **entity classes** to be generated, based on what we already have in our **database**. We'll choose the first approach, where we write our **code first**. Hence the name – **Code First** data model.

Visual Studio will generate our **Database Context class** and **automatically** add a **connection string** in our **Web.config** file, so our app can connect to the database.

We need to write some logic into this file, so it knows which classes we want to store in the database. Luckily, **Visual Studio** already adds this logic in the file (albeit **commented** out):

```
23 // public virtual DbSet<MyEntity> MyEntities { get; set; }
```

A **DbSet** works a lot like a C# **List<T>**. **List<T>** accesses items in **RAM**, whereas **DbSet<T>** accesses items in a **database**. It's a little complicated than that, but that's essentially what it does on the surface. It also supports several list-like methods, such as **Add()**, **Remove()** and so on...

So, all talk aside, let's uncomment that line and specify that we want to store a **collection of tasks** in our database:

```
23 public virtual DbSet<Task> Tasks { get; set; }
```

If we remove all the comments in the **TaskDbContext** class, it should look like this:

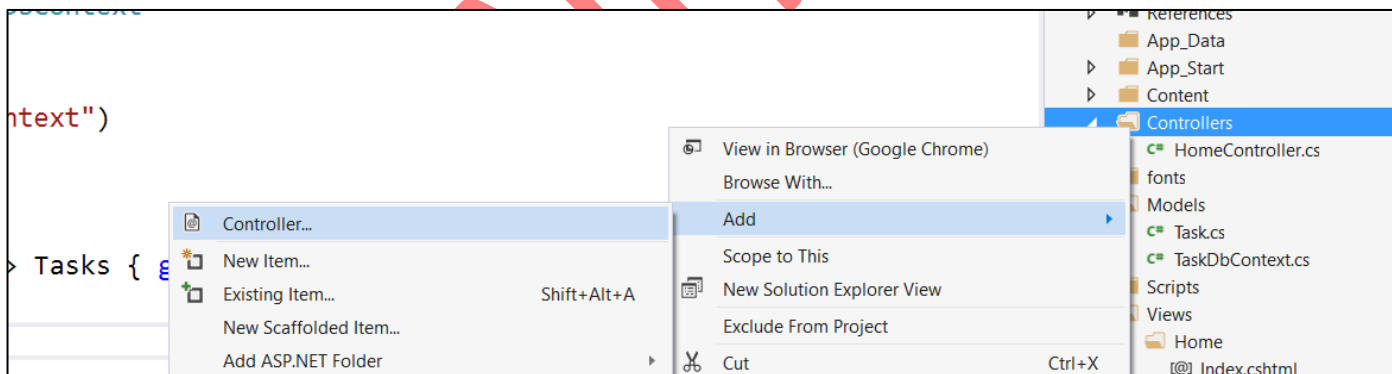
```

1 namespace TODOList.Models
2 {
3     using System.Data.Entity;
4
5     public class TaskDbContext : DbContext
6     {
7         public TaskDbContext()
8             : base("name=TaskDbContext")
9         {
10         }
11
12         public virtual DbSet<Task> Tasks { get; set; }
13     }
14 }
```

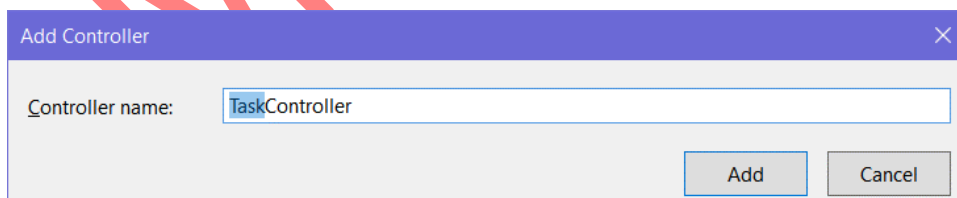
Now, let's write the logic for **adding** and **deleting** tasks.

Create Tasks Controller

Now it's time to create the controller, which **adds** and **deletes** tasks. Right-click the Controllers folder and click on Add ➔ Controller:



In the popup, select “**MVC 5 Controller - Empty**”, then name it **TaskController**:



If we look at our newly-created controller, it looks like this:

```
TaskController.cs
1  using System.Web.Mvc;
2
3  namespace TODOList.Controllers
4  {
5      public class TaskController : Controller
6      {
7          // GET: Task
8          public ActionResult Index()
9          {
10             return View();
11          }
12      }
13  }
```

We don't need the **Index()** action, so just **remove it**, leaving us with this:

```
using System.Web.Mvc;

namespace TODOList.Controllers
{
    public class TaskController : Controller
    {
    }
}
```

Now, it's time to write the logic for both the actions.

```
using System.Web.Mvc;
using TODOList.Models;

namespace TODOList.Controllers
{
    public class TaskController : Controller
    {
        [HttpPost]
        public ActionResult Create(Task task)
        {
        }
    }
}
```

#### Write Logic for Adding Tasks

Let's make the action for **creating** tasks. This action will have a **Task** as a parameter, letting ASP.NET automatically fill in the properties of the task before inserting it into the database:

We're using the **[HttpPost]** attribute, because we're **sending** data to the server, not retrieving it. The first thing we should do is **add some basic validation**:

```
[HttpPost]
public ActionResult Create(Task task)
{
    if (task == null)
    {
        return RedirectToAction("Index", "Home");
    }
}
```

This piece of code checks if the **user actually sent us a task**. If they **didn't**, we can just redirect them to the **Index** action, located within the **Home** controller.

We have our basic validation down, now let's **add the task to the database**:

```
[HttpPost]
public ActionResult Create(Task task)
{
    if (task == null)
    {
        return RedirectToAction("Index", "Home");
    }

    using (var db = new TaskDbContext())
    {
        db.Tasks.Add(task);
        db.SaveChanges();
    }
}
```

What the **using** block does is it allows us to **open** a database connection, then after we're done manipulating the database, **close** that connection and **free any resources** used.

The **db** variable holds all our **DbSets**. We use the **Tasks DbSet** to **add** the task to the **database**, after which we **save the changes** to the database with **db.SaveChanges()**.

Lastly, all we need to do is **redirect the user** to the **Index()** action in the **HomeController**:



```
public class TaskController : Controller
{
    [HttpPost]
    public ActionResult Create(Task task)
    {
        if (task == null)
        {
            return RedirectToAction("Index", "Home");
        }

        using (var db = new TaskDbContext())
        {
            db.Tasks.Add(task);
            db.SaveChanges();
        }

        return RedirectToAction("Index", "Home");
    }
}
```

You might ask yourself why we're redirecting the user twice. We're not! If we see that the **task** sent to us is **null**, we **redirect them** before we could ever **insert an invalid task in the database**. That's why we have one **redirect** for when the task is **invalid** and another **redirect** for when it is **valid**.

Now if our user wants to **add a task**, all they have to do is send a **POST request** to **"/Create"** with their **task title**. Alternatively, they could just use the **HTML form** we'll create in a few minutes.

Almost done, it's time to add the **delete** action as well.

Write Logic for Deleting Tasks

Let's add another method for **removing tasks**. When the user sends a **GET** request to **"/Delete/{id}"**, we want to **delete** the task with that **id**:

```
[HttpGet]
public ActionResult Delete(int? id)
{
}
}
```

Why is the **id** nullable? If the user **doesn't send an id**, we shouldn't try to delete any tasks. Let's write some logic, protecting us from the user:

```
[HttpGet]
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return RedirectToAction("Index", "Home");
    }
}
```

That way, if the user visits let's say **"/Delete/"**, instead of **"/Delete/3"**, we'll just shoo them away to the **homepage**. Now that we're sure our user **gave us an id**, let's **find** the task with that **id** and **delete** it:

```
[HttpGet]
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return RedirectToAction("Index", "Home");
    }

    using (var db = new TaskDbContext())
    {
        var task = db.Tasks.Find(id);
        db.Tasks.Remove(task);
        db.SaveChanges();
    }
}
```

We've **found** the task and **removed** it, but what happens if the user specifies an **invalid id**? Here the problem isn't that the user didn't specify an **id**, the problem is that they specified an **id** of a **task** that **doesn't exist**.

How can we check if the user gave us an invalid **id**? Let's check the [Entity Framework Documentation](#):

## DbSet<TEntity>.Find Method

[Other Versions](#) ▾

*[This page is specific to the Entity Framework version 6. The latest version is available as the 'Entity Framework' NuGet package. For more information about Entity Framework, see [msdn.com/data/ef](https://msdn.com/data/ef).]*

Finds an entity with the given primary key values. If an entity with the given primary key values exists in the context, then it is returned immediately without making a request to the store. Otherwise, a request is made to the store for an entity with the given primary key values and this entity, if found, is attached to the context and returned. **If no entity is found in the context or the store, then null is returned.**

Perfect! If the **Find()** method **can't find** the item, it **returns null**.

Let's add one more check for the **id**:

```
[HttpGet]
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return RedirectToAction("Index", "Home");
    }

    using (var db = new TaskDbContext())
    {
        var task = db.Tasks.Find(id);

        if (task == null)
        {
            return RedirectToAction("Index", "Home");
        }

        db.Tasks.Remove(task);
        db.SaveChanges();
    }
}
```

This way, we'll redirect the user if the **id** is invalid as well.

Finally, let's redirect the user upon a successful removal of the task:

```
[HttpGet]
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return RedirectToAction("Index", "Home");
    }

    using (var db = new TaskDbContext())
    {
        var task = db.Tasks.Find(id);

        if (task == null)
        {
            return RedirectToAction("Index", "Home");
        }

        db.Tasks.Remove(task);
        db.SaveChanges();
    }

    return RedirectToAction("Index", "Home");
}
```

We're nearly done with the controller actions. All that's left is to **retrieve all tasks** and hand them to the **index view**, so the user can see them.

Write Logic for Listing Tasks

Let's go into the **Controllers/HomeController.cs** file:

```
using System.Web.Mvc;

namespace TODOList.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Not much going on here... Let's retrieve all the **tasks** and give them to the **index** view:

```
using System.Linq;
using System.Web.Mvc;
using TODOList.Models;

namespace TODOList.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            using (var db = new TaskDbContext())
            {
                var tasks = db.Tasks.ToList();

                return View(tasks);
            }
        }
    }
}
```

We retrieve all the tasks, using **db.Tasks.ToList()**. Converting them to a list puts all of the tasks into **RAM** and we can render them in the view.

Create View

Let's go into the **Views/Index.cshtml** file and replace its contents with this:

```
@model List<TODOList.Models.Task>
@{
    ViewBag.Title = "Home Page";
}

<div class="row">
    <div class="col-md-4">
```

```
<h3>TODO List</h3>
<ol>
  @foreach (var task in Model)
  {
    <li>@task.Title @Html.ActionLink("[Delete]", "Delete", "Task", htmlAttributes: null,
routeValues: new { id = task.Id })</li>
  }
</ol>
@using (Html.BeginForm("Create", "Task", null, FormMethod.Post, new { @class = "form-inline" }))
{
  <div class="form-group">
    <input type="text" class="form-control" name="title" placeholder="Task Title"
autofocus="autofocus" />
  </div>
  <div class="form-group">
    <input type="submit" class="btn btn-primary" value="Add Task" />
  </div>
}
</div>
</div>
```

Let's break this code down a bit:

The **model** variable at the top of our view defines what the **type** of the model that the view is receiving will be. We have to specify this, because the type information isn't passed between the controller and the view.

**ViewBag.Title** sets the title of the page.

Further down we see a **foreach** block, which iterates through the **model** and adds **<li>** items to our **ordered list**

In the **<li>**, we enter the **title** of the task and **generate an ActionLink**, which takes us to the **delete** page of that **task**, using its **id**. The **ActionLink** accepts several parameters. Let's break them down:

- "[Delete]" – the **link text**
- "Delete" – the **action** for the link
- "Task" – the **controller** to which the **action** belongs
- **htmlAttributes: null** – the **html attributes** for the link. Since we're not using any, we can just leave it as **null**.
- **routeValues: new { id = task.Id }** – an anonymous object, which sets the **id** to the **task's id**, effectively making the link look like this: **"/Task/Delete/2"**

After the **foreach** block, we can see the **Html.BeginForm**, which is the proper Razor way of generating forms. We give it a lot of parameters, so let's explain them:

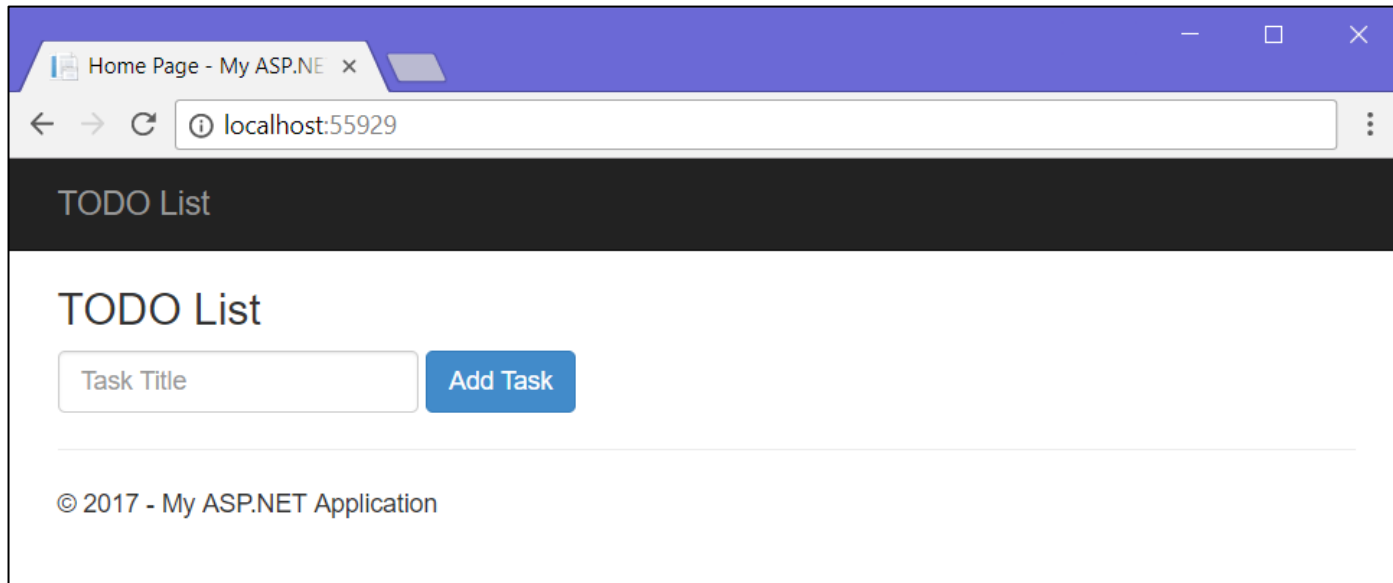
- **"Create"** – the **action name** where this form will send its data
- **"Task"** – the **controller** to which the **action** belongs
- **null** – the **route values** of the form. In contrast to having used route values in the **ActionLinks** above, this form doesn't need to be sent to a particular route. Hence, we set it to **null**.
- **FormMethod.Post** – the **form method** we'll use for this form

- `new { @class = "form-inline" }` – the **html attributes** of our form. In contrast to the **ActionLinks** above, we actually need the form to have html attributes, so we set this to an **anonymous object**, which only contains the **css class** of the form.

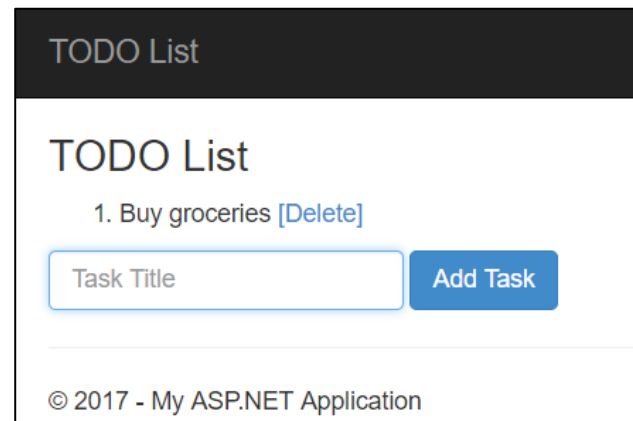
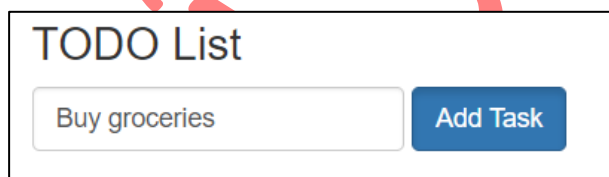
Whew, that was a lot of writing. Let's try to actually run the application.

Test the Application

If we run the application, we should end up with something like this:



No tasks in sight. Let's try adding one:



The task showed up! Let's add a few more:

TODO List

TODO List

1. Buy groceries [\[Delete\]](#)

2. C# Homework [\[Delete\]](#)

3. Make a TODO List [\[Delete\]](#)

They get added successfully! Let's try deleting one:

TODO List

TODO List

1. Buy groceries [\[Delete\]](#)

2. C# Homework [\[Delete\]](#)

3. Make a TODO List [\[Delete\]](#)

<localhost:55929/Task/Delete/3>



TODO List

TODO List

1. Buy groceries [\[Delete\]](#)

2. Make a TODO List [\[Delete\]](#)

It's gone! No more C# Homework!

If you followed all the steps correctly and read all the explanatory text, you should have a working TODO List application.