# Data-Driven Apps

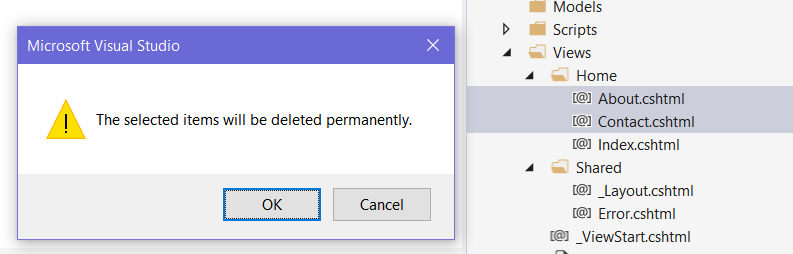
These are apps, which need to store data in a database to work properly.

## TODO List

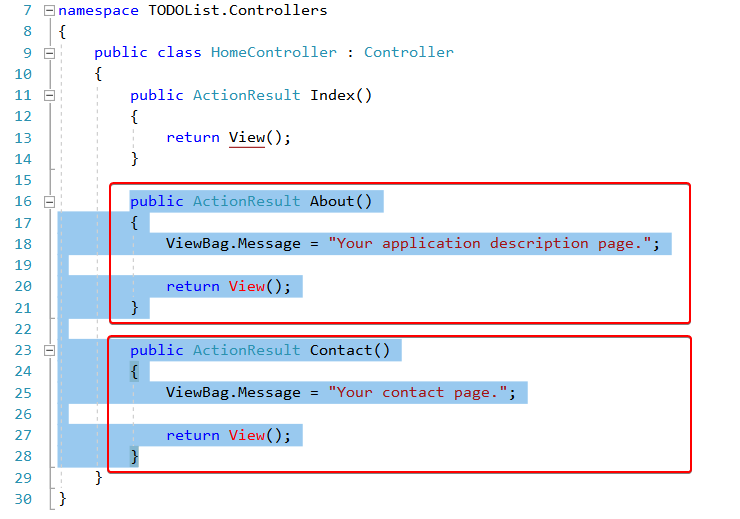
Create a TODO list application, which keeps track of a person’s **tasks** inside a **database**. The application should support **creating** tasks and **deleting** tasks.

### Prepare a New Project

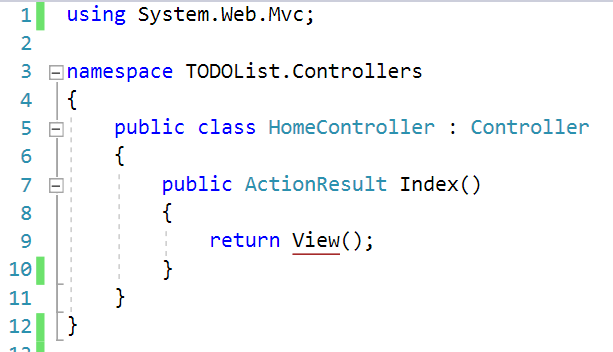
Just as before, create a new project and name it TODOList. After you create it, we need to remove any unnecessary views yet again:



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



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



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** Contactmenu items as well.

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



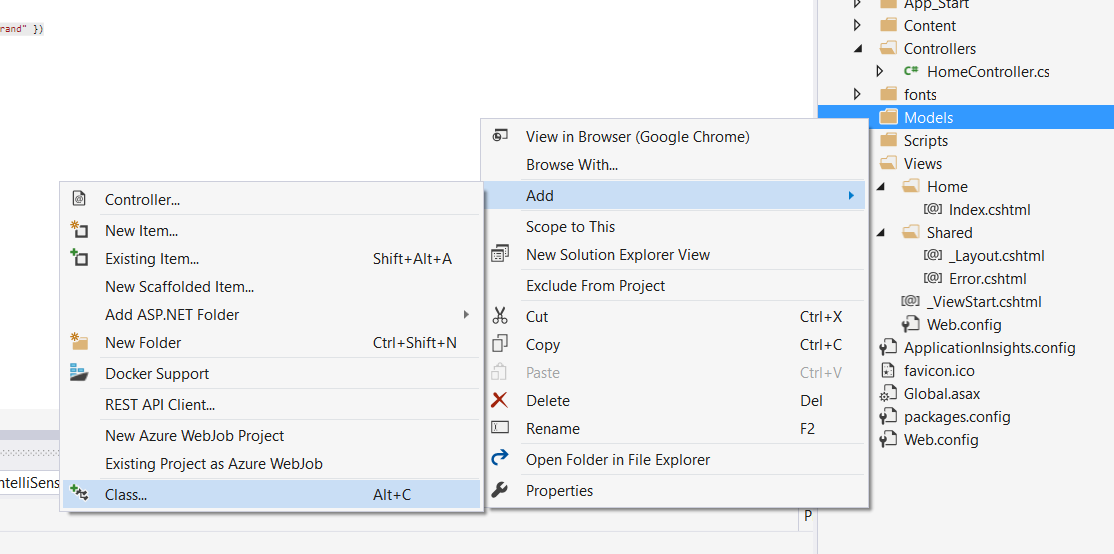
We’ve removed all the unnecessary stuff in our project and we are ready to actually start writing code!

### Create Task Model

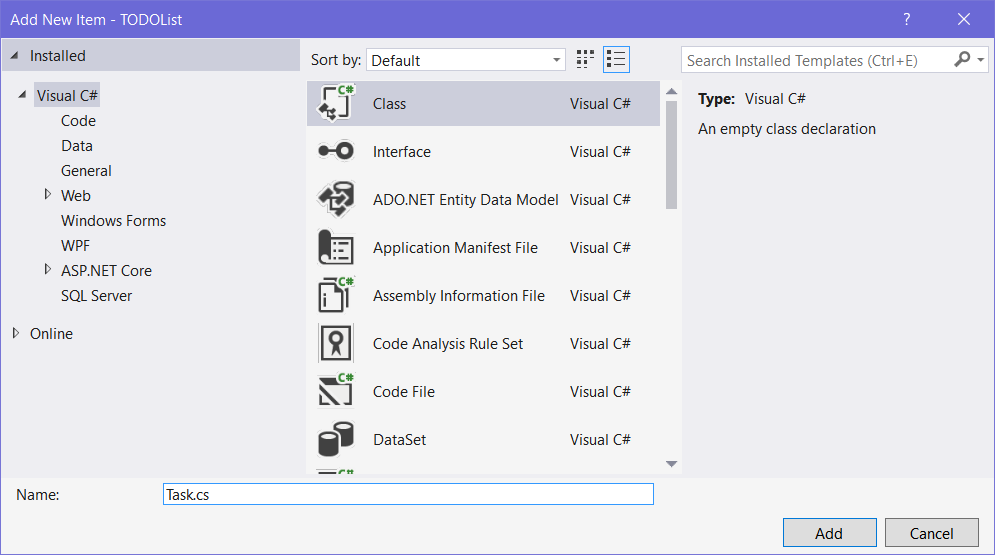
Now it’s time to create our **task entity class**. Our **task** will be simple. It will have **2 properties**:

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

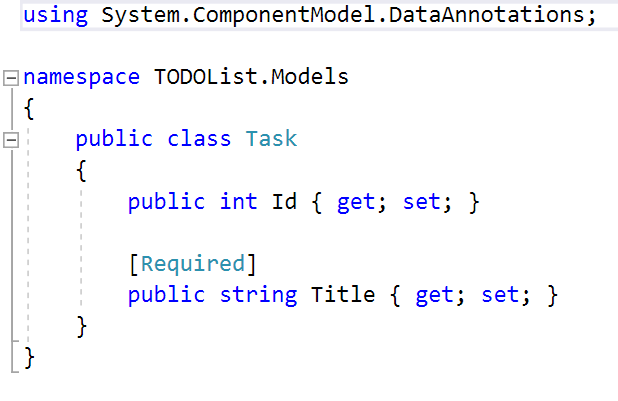
Let’s go in our Modelsfolder and **add a Task class**:



In the menu, which popped up, select Classand name it Task.cs:



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



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

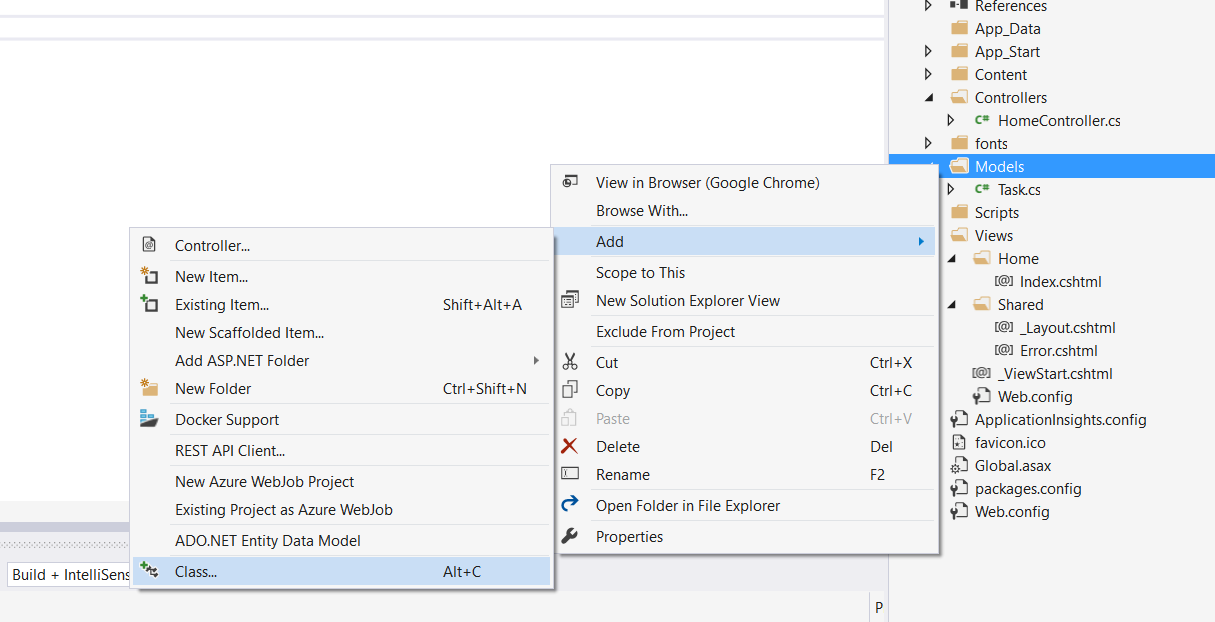
You might have noticed that this **looks** a lot like a **standard C# OOP class**. That’s because it’s exactly that! Entity Framework works with ordinary classes to achieve its **object-relational mapping**.

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

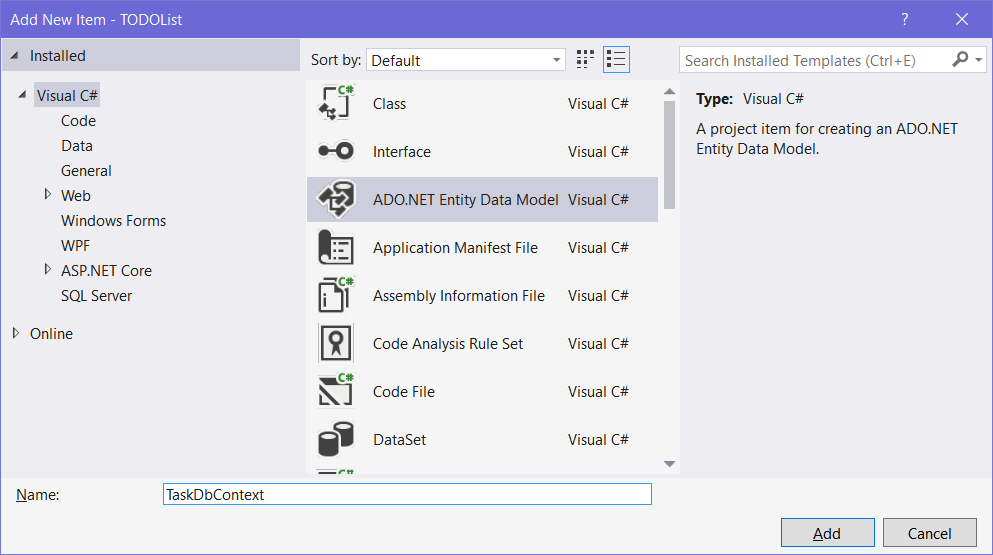
### Create Database Context

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.

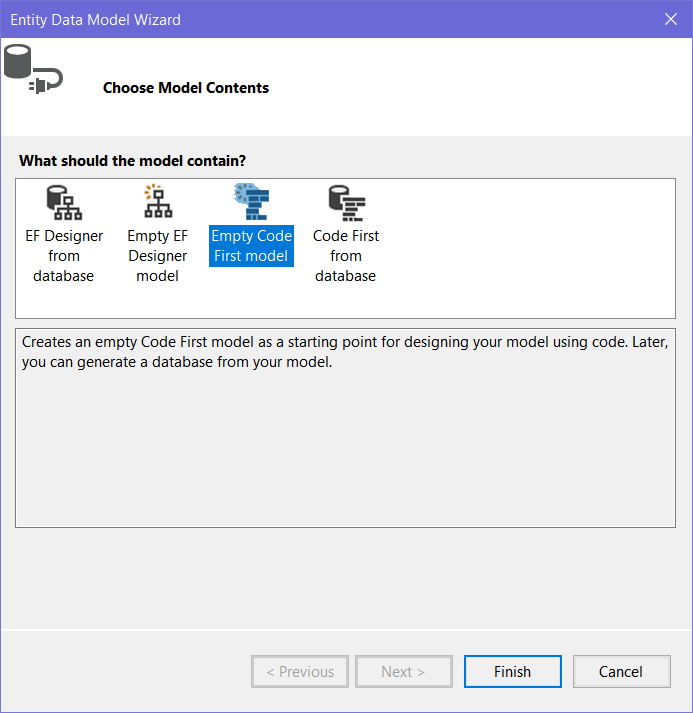
Right click on the Models folder and add a new class:



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.configfile, 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):

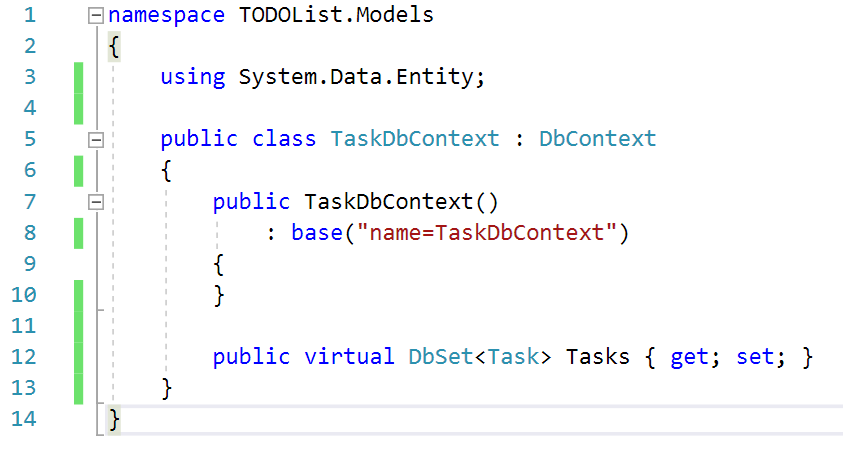


A DbSet works a lot like a C# List<T>. List<T> accesses items in **RAM**, whereas DBSet<T> accesses items in **a database**. It’s a little complicated than that, but that’s essentially what it does on the surface. It also supports several list-like methods, such as Add(), Remove() and so on...

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

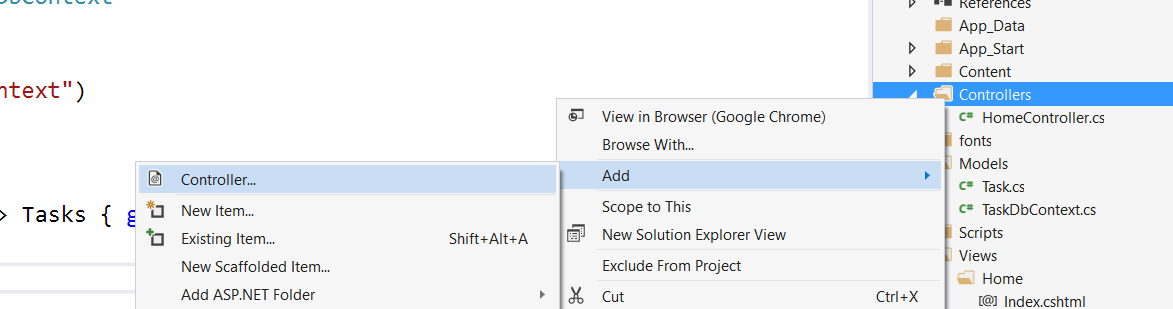


If we remove all the comments in the TaskDbContext class, it should look like this:

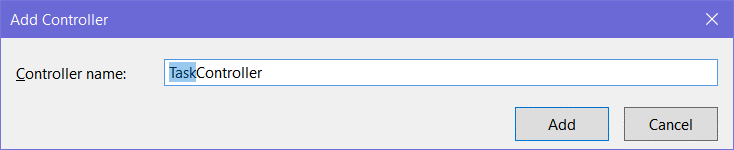


Now, let’s write the logic for **adding** and **deleting** tasks.

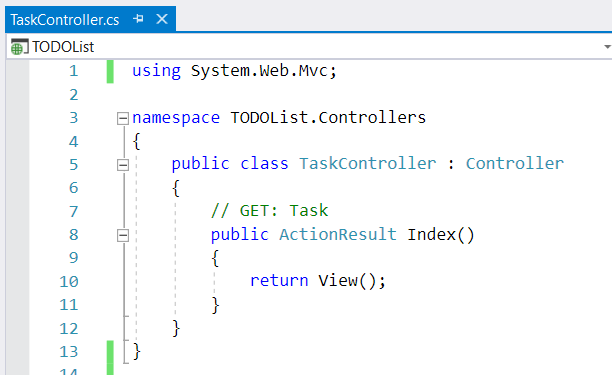
### Create Tasks Controller

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


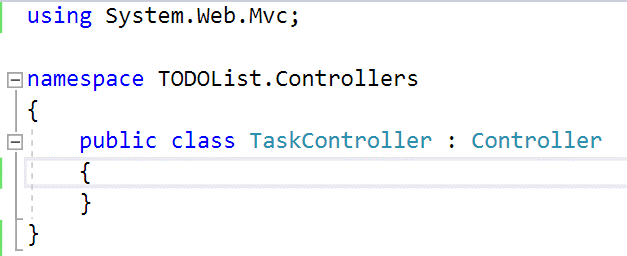
In the popup, select “MVC 5 Controller – Empty”, then name it TaskController:



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

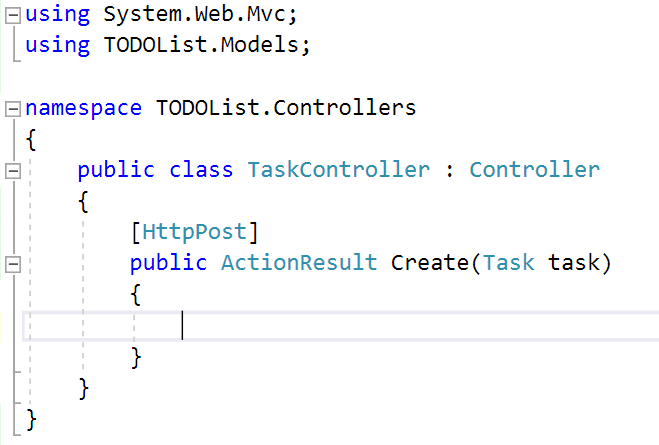


We don’t need the Index() action, so just **remove it**, leaving us with this:

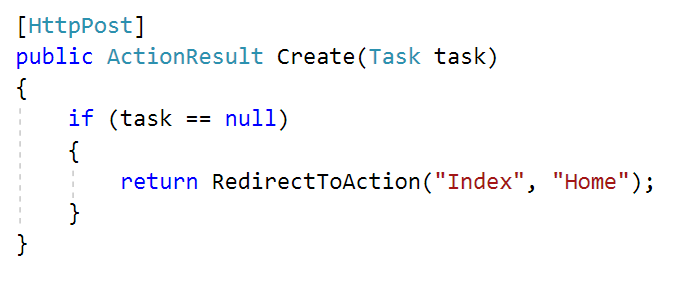


Now, it’s time to write the logic for both the actions.

### Write Logic for Adding Tasks

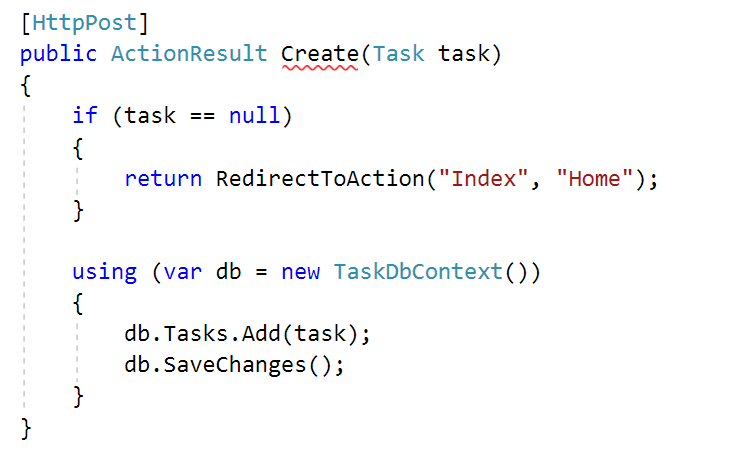
Let’s make the action for **creating** tasks. This action will have a Taskas a parameter, letting ASP.NET automatically fill in the properties of the task before inserting it into the database: 

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**:



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.

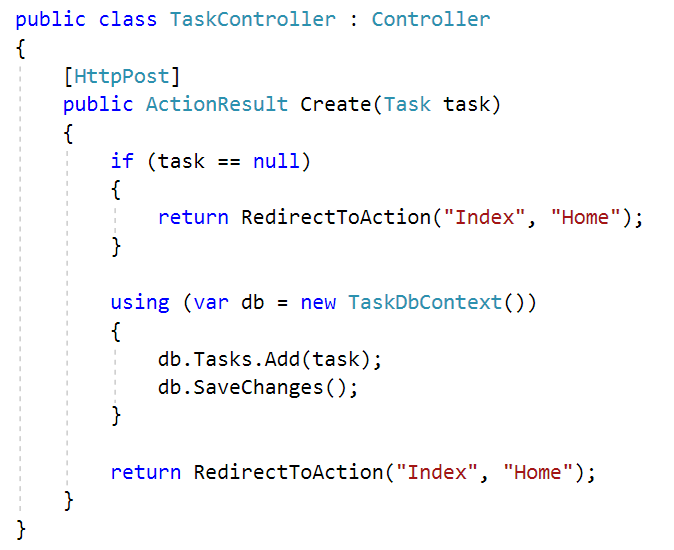
We have our basic validation down, now let’s **add the task to the database**:



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 TasksDbSet 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:



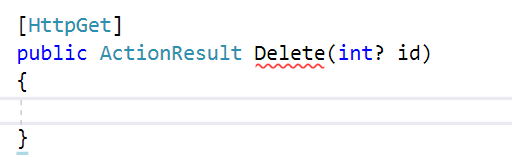
You might ask yourself why we’re redirecting the user twice. We’re not! If we see that the **task** sent to us is **null**, we **redirect them** before we could ever **insert an invalid task in the database**. That’s why we have one **redirect** for when the task is **invalid** and another **redirect**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

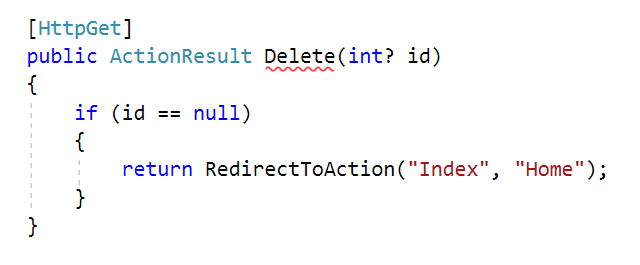
Almost done, it’s time to add the **delete** action as well.

### Write Logic for Deleting Tasks

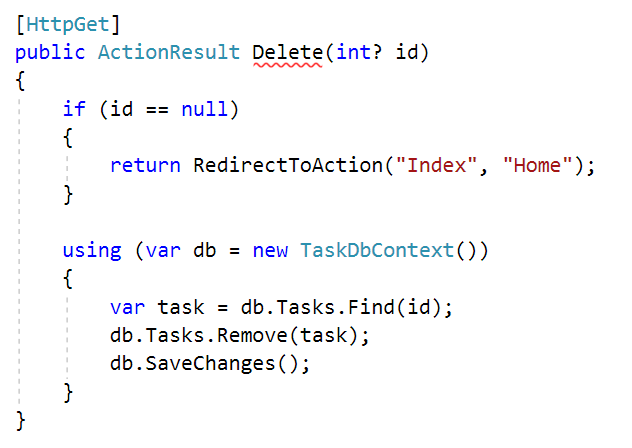
Let’s add another method for **removing tasks**. When the user sends a **GET** request to “/Delete/{id}”, we want to **delete** the task with that id:



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:

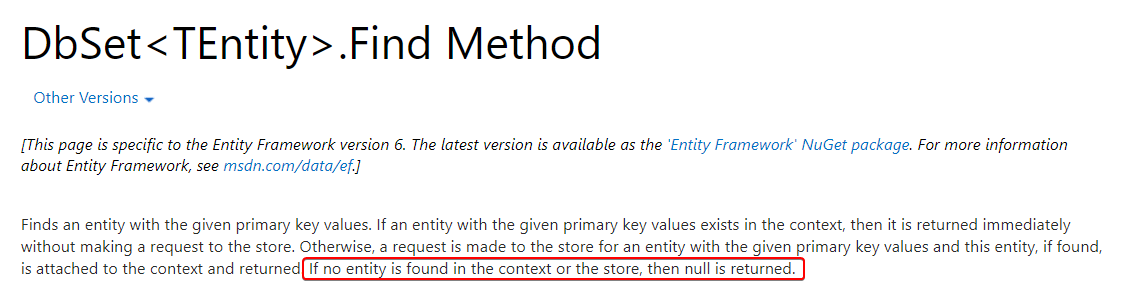


That way, if the user visits let’s say “/Delete/”, instead of “/Delete/3”, we’ll just shoo them away to the **homepage**. Now that we’re sure our user **gave us an** id, let’s **find** the task with that id and **delete** it:



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](https://msdn.microsoft.com/en-us/library/gg696418(v=vs.113).aspx):



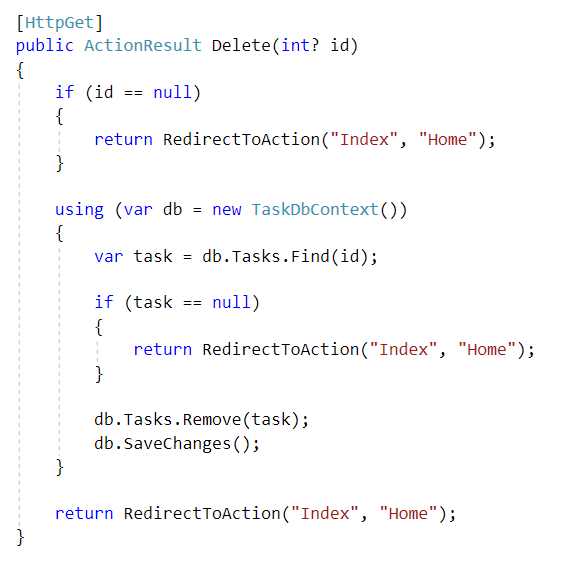
Perfect! If the Find() method **can’t find** the item, it **returns null**.

Let’s add one more check for the id:



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

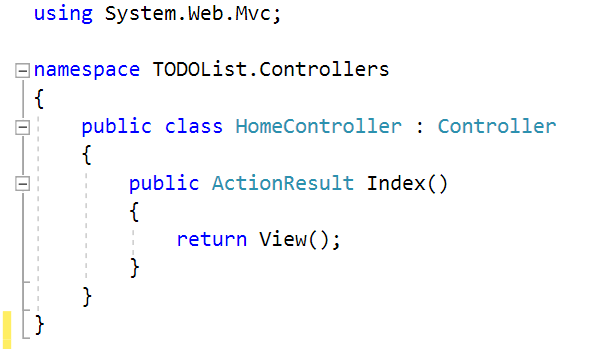
Finally, let’s redirect the user upon a successful removal of the task:



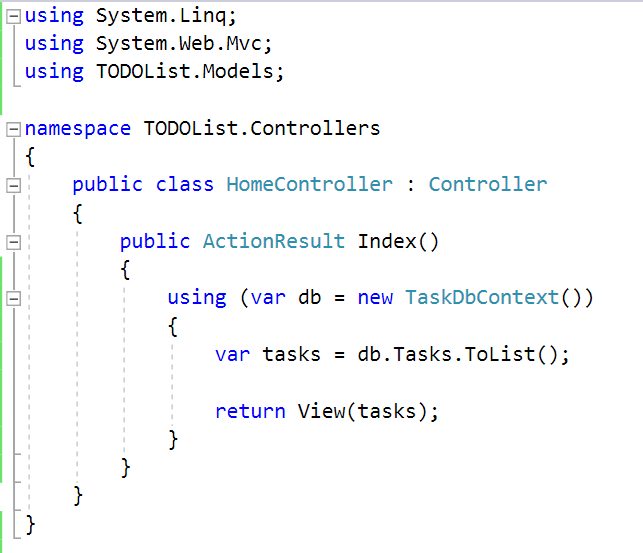
We’re nearly done with the controller actions. All that’s left is to **retrieve all tasks** and hand them to the **index view**, so the user can see them.

### Write Logic for Listing Tasks

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



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



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”

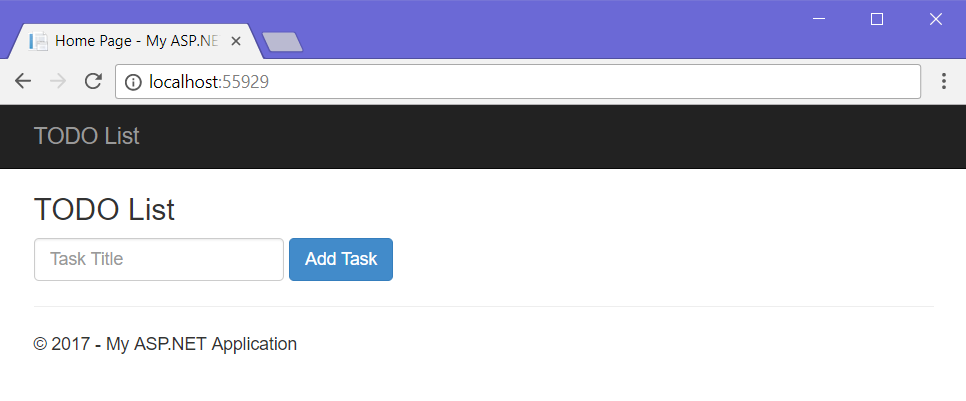
After the foreach block, we can see the Html.BeginForm, which is the proper Razor way of generating forms. We give it a lot of parameters, so let’s explain them:

* "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

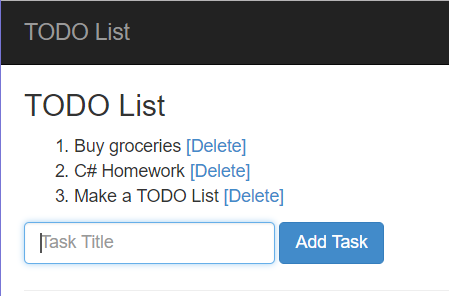
If we run the application, we should end up with something like this:



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

|  |  |  |
| --- | --- | --- |
|  | 🡺 |  |

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



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

|  |  |  |
| --- | --- | --- |
|  | 🡺 |  |

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.