# C# Basic CRUD: ToDo List

## Problem

You have been tasked to create a simple **TODO List** application. The application should hold **tasks**, which are the main app **entities**.

The functionality of the application should support **listing, creating**, **updating** and **deleting** tasks.

The application should **persist** the data into a **database**.

## Overview

### Requirements

* **Visual Studio 2017**
* **SQL Express 2017**
* **SQL Server Management Studio (SSMS)**

### Data Model

The Task entity holds **3 properties**:

* id – int
* title – non-empty text
* comments – non-empty text

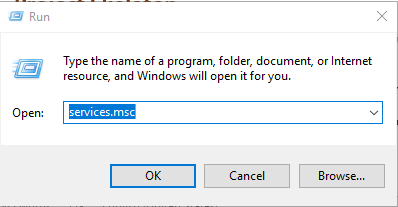
## SQL Express 2017 and SSMS Installation

### SQL Express 2017

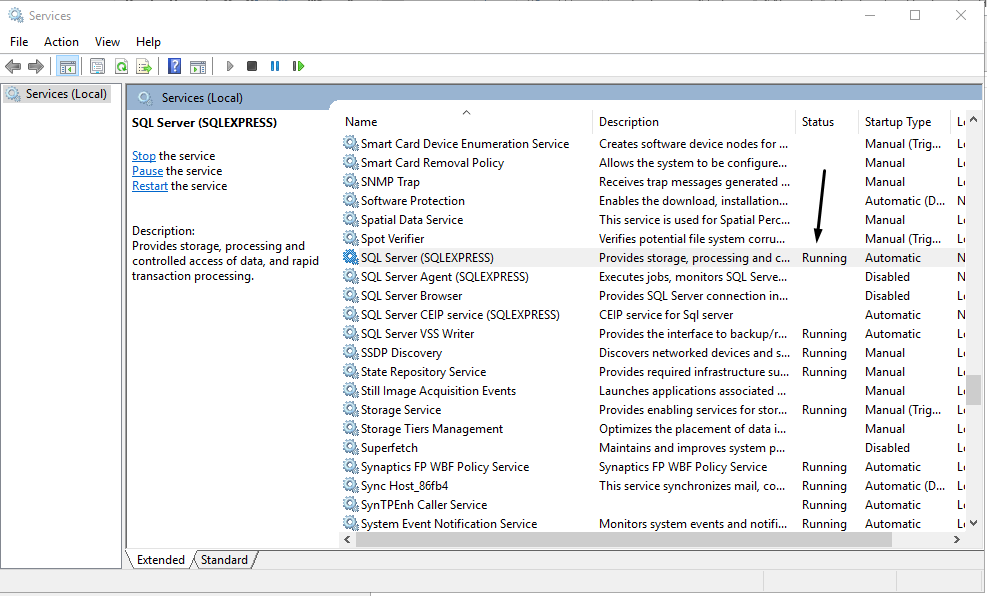
Download the installation file using this link - [Download](https://go.microsoft.com/fwlink/?LinkID=799012).

Select Basic installation and follow the installation process.

After Installation process open **Run.exe**. And open **services.msc**



Make sure **SQL Server (SQLEXPRESS)** process is Running, if it’s not: **right click-> start.**



### SQL Server Management Studio

Download the installation file and follow installation process - [Download](https://go.microsoft.com/fwlink/?linkid=2014306)

## Project Skeleton

You will be given the application’s skeleton.

You must implement the **Model and Controller**.

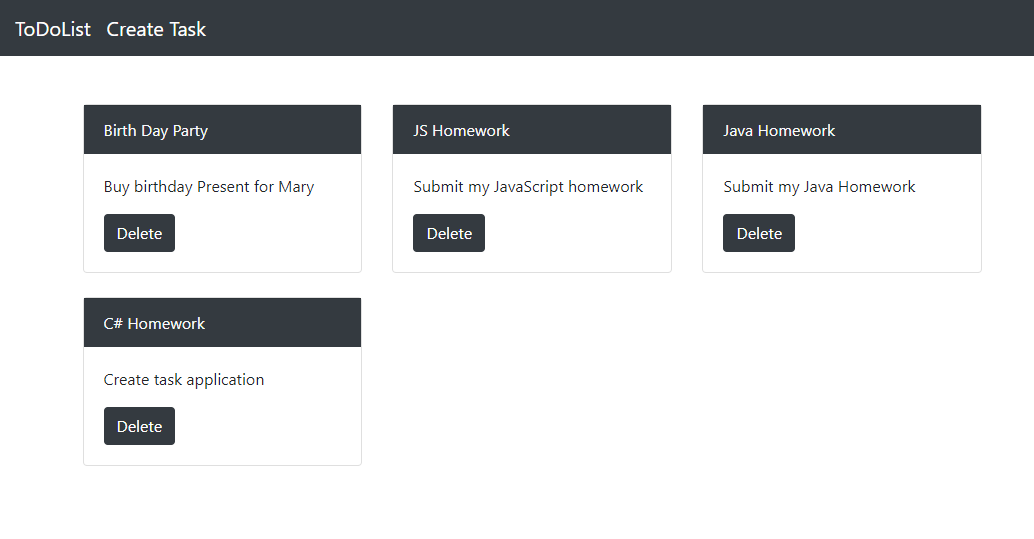
The application’s views will be given to you fully implemented. You only need to include them in your business logic.

Everything that has been given to you inside the skeleton is **correctly implemented** and if you write your code **correctly**, the application should work just fine. You are free to change anything in the Skeleton on your account.

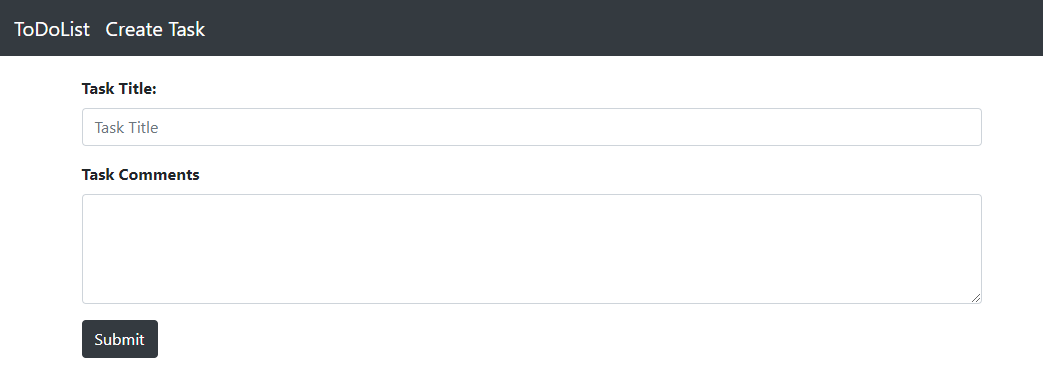
### User Interface

This is the user interface or how the application’s pages should look in their final form (fully implemented). You have several pages, described below:

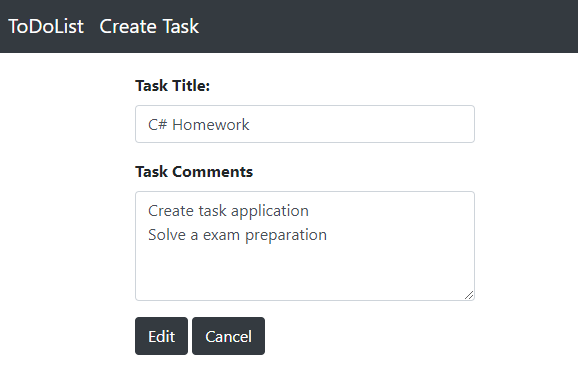
#### Index Page



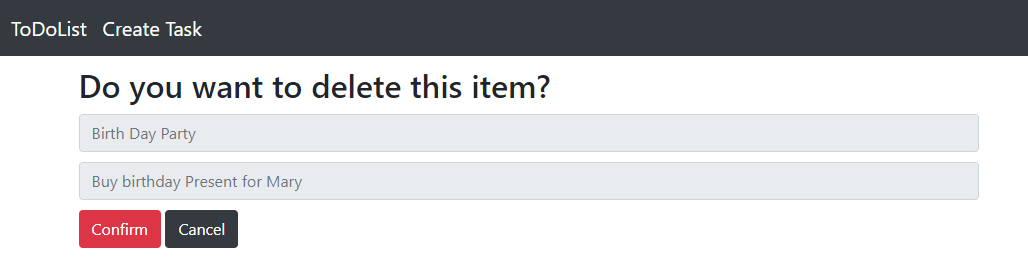
#### Create Page



#### Update Page



#### Delete Page

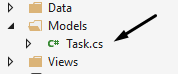


## Task Entity

It's time to create our first entity. We are using **Entity Framework Core** for **ORM**.

For this task we have only one simple entity, this means that we have no relations so it will be really easy job for you.

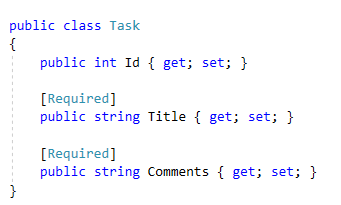
Create new C# class called **Task.cs** in **Models** folder.



You should describe the entity as following:

* id – int
* title – non-empty text
* comments – non-empty text

So the class should look like this:

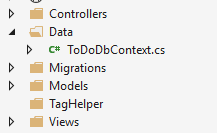


By default when you have property **int Id** in your class, **Entity Framework Core** will map it to **Primary Key** in the table.

## Creating DbContext

To save our tasks in the database, first we must create **DbContext** class, which will map the **Task** entity to table.  
**DbContext** class will provide us with functionality to work with our database table and records in that table.

Create new folder in your project called **Data**. Inside the folder create **ToDoDbContext.cs**.



Our class must inherits the **DbContext** class which is provided by **Entity** **Framework**.

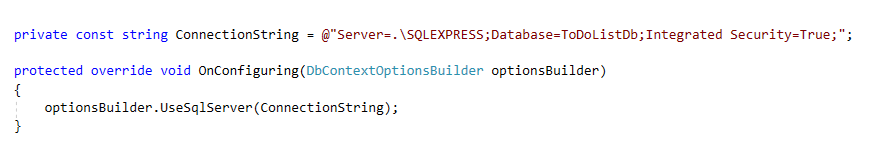
**Note:** Include "**using Microsoft.EntityFrameworkCore;"** in your **ToDoDbContext.cs**



To map our entity into table **Entity** **Framework** needs a **DbSet<Task>**   
**Note that Task is used in C# base classes so you must be sure that you are including the right using: "using ToDoList.Models;"**

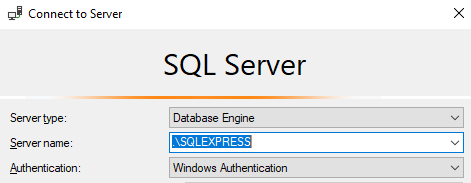


Now we must configure our connection to the database.

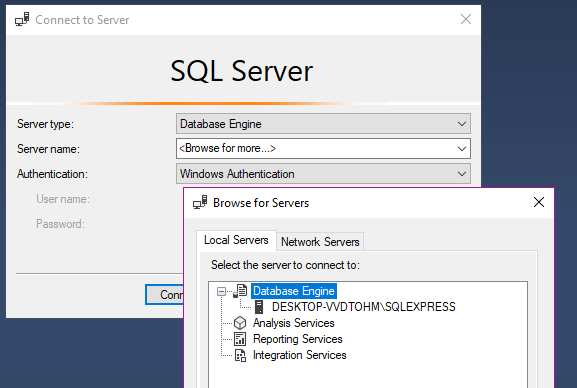


**ConnectionString** is used by Entity Framework to connect to our local Database.

If you have installed **SQL Express** you should be able to connect with the given string.  
We can always find server name in SSMS.



Or you can open <**Browse for more…>**



On the given computer both "**DESKTOP-VVDTOHM\SQLEXPRESS "** and "**.\SQLEXPRESS"** will connect to the local server.

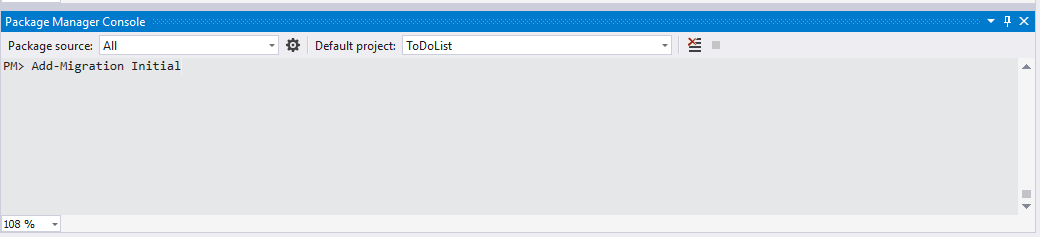
Database name is set to **ToDoListDb**,but you can change it to whatever you want.

## Migrations and Database Update

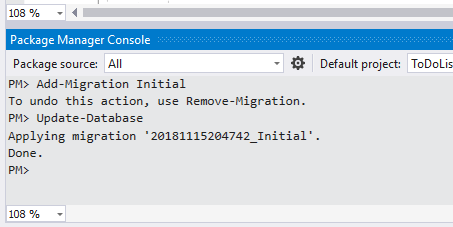
Now we must map our model to database. This is done through **migrations** which are something like SQL Queries but written on C#.

To **add** new **migration** go to **Tools->NuGet Package Manager->Package Manager Console**

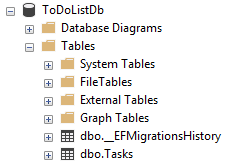
Write the following command in the console: **Add-Migration Initial.**



After that you have to execute: **Update-Database**Both commands successfully executed should look something like this.



After this you should be able to find your database with **Tasks** table in **SSMS**.



## Task Controller

To handle user's interactions the application must have controllers.

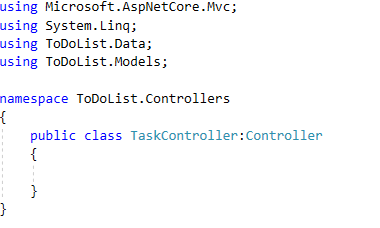
Let's create Task controller which will be responsible for

* Listing the tasks
* Creating the tasks
* Updating the tasks
* Deleting the tasks

Create new **TaskController.cs** in **Controllers** folder

The class should look like this.

Include all the **usings** on the picture you will probably need them later.

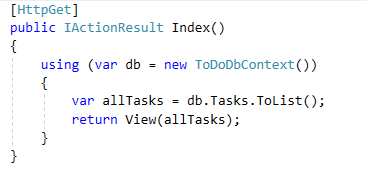


### Listing the tasks

In the task Controller create Index Action.



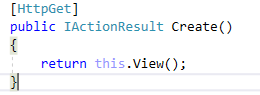
Index Action will be called when the user open our web application.  
All our index should do is to get all the tasks in the database and pass them to the View as **List<Task>.**



### Creating the tasks

To create a single task first we must open the **Create page**, after that we must fill the form and post the information about the task.  
That means that our controller will handle **GET** **request** and will return the View. After that the application will handle the **POST** **request** with the given information to create new task.  
Probably you already know that this will require two actions.

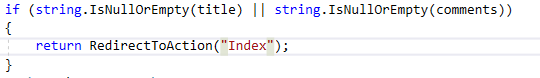
The first action is just simple **Create()** action with no other functionality but to return **View()**



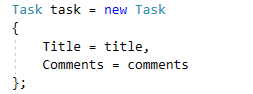
Absolutely nothing special.

Now the action which handles the **POST** **request** is a bit more complex. The action receives two string parameters **title** and **comments**. Using those parameters we have to create Task and add it to database.

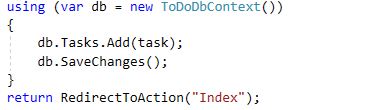
First we must check if both parameters are not **empty or null**. If one of the parameters is null or empty we will simply ignore the request and redirect to **Index** action.



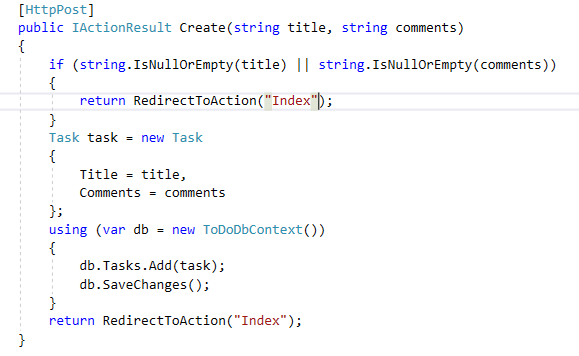
Now once we made a simple validation, we can create a **Task object.**



The last step is to add the new Task to tasks, save the changes and redirect user to Index Page.



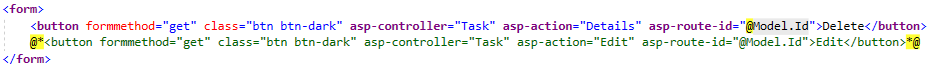
The entire action should look like this.



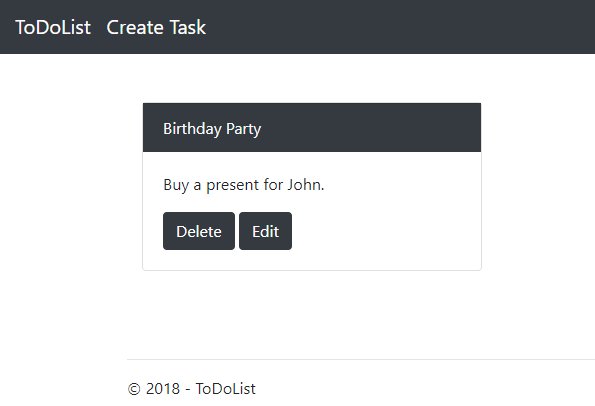
### Editing a task

Imagine if you misspell a word, it's stupid if you need to delete the task and create a new one.  
So let's **add** a **functionality** to our app, which will make **editing** of tasks possible.  
Now you might be wondering, that you have no **edit** **button** or any edit view in your application.  
Well the views are right there ready for use, you just have to "unlock" them.

Open \_Task.cshtml file, which is located in Views\Shared\. This is the partial view, which represents each of the tasks on the index page.   
We can make changes to our task view without even touching the Index.cshtml file  
and the changes will be applied over every single task in the page.  
In the file you will find that we have one button in comments.



Remove the comments and try to start the page again.

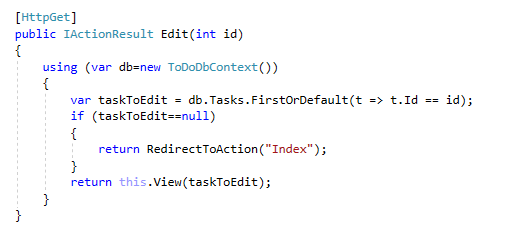
Now the application should look like this.  


Although nothing will happen if we press the **Edit** button. As you can see the button will generate a **GET** **request** on the given route **Task/Edit/{id}**, but we have no **Edit** action to handle the request.

Open the Task controller and create new action called **Edit**. The action will receive **int Id** as parameter and will handle **HttpGet** requests. Something like this:

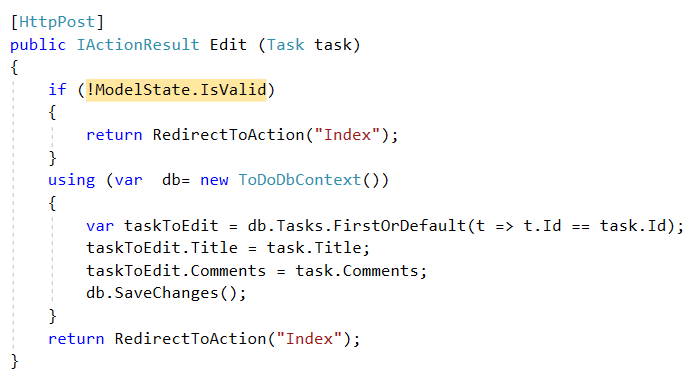


The logic behind the **Edit** action is simple. You have to find the task by given Id and send it to the View. Keep in mind that this is **GET** **request** so any kind of Id can be sent through the **Url**.



Now we can press our **Edit** **button** and if everything is fine we will generate a page similar to the Create page, but of course with populated form.



The final step is to create Edit action, which handles **Post** request with the data.  
The action will receive a **Task** as parameter. Validate the mapped **Task** from the **POST Request** using **ModelState.IsValid**. You have to find a **task from the database** with the **exact same id** as the parameter and assign the new values.  
Here is the example.

### Deleting a task

Looking at the pictures at the start you've probably noticed that when we want to delete a task, first we are going through a page that shows task title and comments.  
On the same page we can delete the task forever or just cancel the request.

#### Details action

So similar to the creating process before implementing the **Delete** action first we must implement the action that handles **GET** request to the Details Page.  
The **Details** action receives **id** as parameter. The **id** represents the tasks' id which we have to get from DB and send to the View.  
Following all these steps you probably already know how to implement the action so try it by your own.

#### Delete action

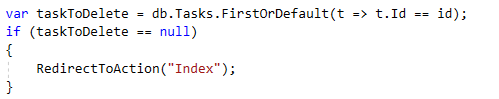
It is time to implement the last action. It is the action that provides the functionality to delete tasks.

The action works as it follows.

You will receive **id** as parameter.



You must find the tasks which has equal id to the parameter.



Remove it from **tasks.**



Don’t forget to **save** the **changes**.  
And **redirect** to the **Index** action.

Both methods should look like this:



Alright that's enough for our application to work just fine. Feel free to implement new functionality on your own. ☺