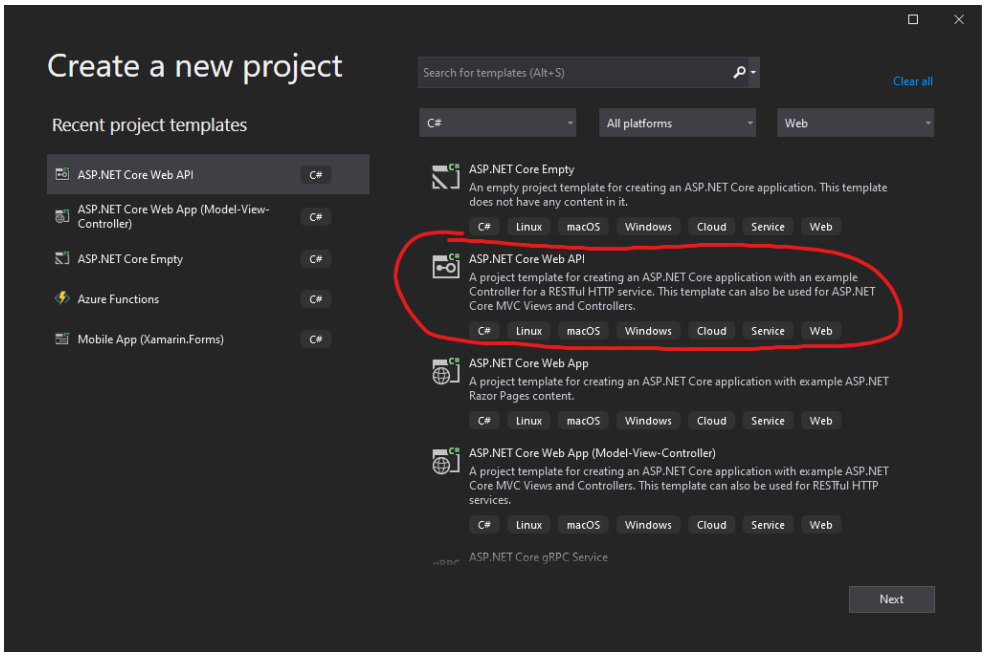
Making a Todo List API with a Postgresql Database

Software I'll need for this Project:

1. Visual Studio 2019
2. C#
3. Asp .Net SDK
4. Docker and Docker CLI

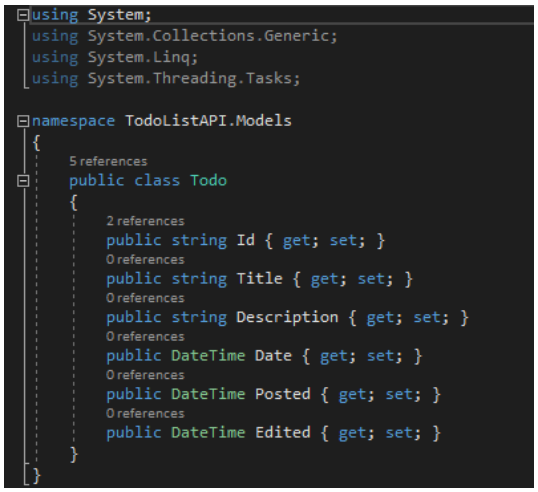
I'm going to create an API that will manage the data for a Todo List application. The first thing I'll do is create a .Net Web API default project:



For this project I'm going to name it TodoListAPI. This will give me the scaffolding to start working from. The first thing that I need to get functioning on this application is a database to store data to and pull data from. There are a number of ways of going about this, many of which could be completely valid. Due to familiarity, For this project I decided to create a Postgresql database that will be running on a docker container, that I will then connect to from my API. Before I can create that connection I obviously need the container running so I'll open a powershell terminal and enter the following command:

```
docker run --name=postgres  -p 5432:5432 -e POSTGRES_PASSWORD=[my_password] postgres
```

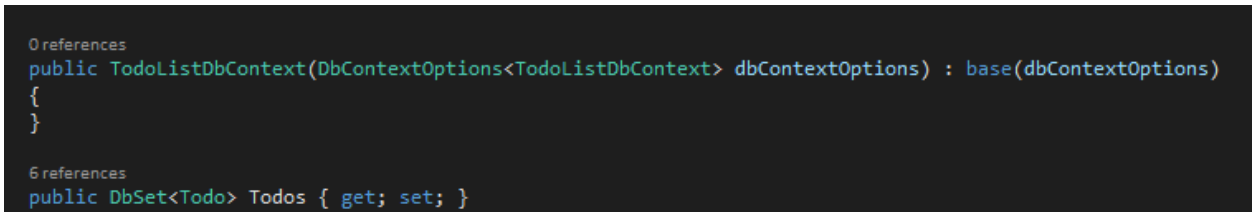
This command will build the latest postgres image and run it in a container named postgres exposing it on port 5432 on my localhost secured by ‘my_password’. Once this is done, I successfully have a postgres docker container running on my system that is ready to be connected to. Now, I need to set up my application to connect to it. To do this I’m going to start by downloading a couple of necessary NuGet Packages. I’ll right click my solution file and click on the <Manage NuGet Packages for Solution…> option. This will bring up the NuGet window in Visual Studio. I’ll then proceed to download EntityFramework6.Npgsql, Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Design, and Npgsql.EntityFrameworkCore.PostgreSQL. I now have all the packages I need now I need to set up the class objects that will represent the tables of my Database and the Database Context class object. To do this I will create a couple of folders in my Project. I’ll make a folder named Models and another folder named Database. Inside my Models folder I’m going to make a class called Todo.cs. This class will contain the attributes I need for my Todos. It will have an Id, a Title, a Description, a date for the item to be done, a date posted, and a nullable date Edited. Once I’m done my class will look like this:



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

namespace TodoListAPI.Models
{
    5 references
    public class Todo
    {
        2 references
        public string Id { get; set; }
        0 references
        public string Title { get; set; }
        0 references
        public string Description { get; set; }
        0 references
        public DateTime Date { get; set; }
        0 references
        public DateTime Posted { get; set; }
        0 references
        public DateTime Edited { get; set; }
    }
}
```

Next, I’ll be adding another class to my Database folder called TodoListDbContext.cs. Within this class I will add two using statements. “Using Microsoft.EntityFrameworkCore” and “Using [Project_Name].Models” in my case my Project Name is TodoListAPI. Next, I will have my class inherit from DbContext. Now, I’m going to add the following code:



```
0 references
public TodoListDbContext(DbContextOptions<TodoListDbContext> dbContextOptions) : base(dbContextOptions)
{
}

6 references
public DbSet<Todo> Todos { get; set; }
```

This will tell the database to create a table called Todos and to populate its columns based off of my Todo class.

Now, I'm ready to set up my Database connection. I need to open up my Startup.cs file and inside the ConfigureServices() method I'm going to add the using statement "Using Microsoft.EntityFrameworkCore. Then add the following code:

```
services.AddDbContext<TodoListDbContext>(options => options.UseNpgsql(Configuration["DATABASE_URL"]));
```

I'm accessing my connection string inside my appsettings.json file however I could have just as easily put it directly into my AddDbContext call. The following is my connection string:

```
"DATABASE_URL": "host=localhost; port=5432; database=postgres; user id=postgres; password=[redacted]",
```

Inside my Program.cs file I'll include the same using statement as before and then within my Main() method add the code:

```
0 references
public static void Main(string[] args)
{
    var host = CreateHostBuilder(args).Build();
    using(var scope = host.Services.CreateScope())
    {
        var db = scope.ServiceProvider.GetRequiredService<TodoListDbContext>();
        db.Database.Migrate();
    }

    host.Run();
}
```

Now for the final ingredient to set up our database we need to run a dotnet ef add migrations command from our terminal:

```
dotnet ef migrations add [migration name]
```

The migration name can be anything. For my first migration I usually call it "initial." Once this runs and it returns successfully I'll see a Migrations Folder in my Project that should contain two files. A file that will be called [some id]_[migration_name].cs and a [Database Context Class Name]ModelSnapshot.cs.

At this point I have everything set up to connect to my database and populate it with a Todos table that is modeled after my Todo class object. It is ready to have data stored to it and read from it. Now, I need to create my API controller that will utilize the database. I'm going to start by going to the Controllers folder and creating a Controller Class object named TodoListController.cs. I can do this by right clicking on the Controllers folder → Add → Controller → API Controller with read/write actions and naming it TodoListController. This will give us the following scaffolding code:

```
// GET: api/<TodoListController>
[HttpGet]
0 references
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}

// GET api/<TodoListController>/5
[HttpGet("{id}")]
0 references
public string Get(int id)
{
    return "value";
}

// POST api/<TodoListController>
[HttpPost]
0 references
public void Post([FromBody] string value)
{
}

// PUT api/<TodoListController>/5
[HttpPut("{id}")]
0 references
public void Put(int id, [FromBody] string value)
{
}

// DELETE api/<TodoListController>/5
[HttpDelete("{id}")]
0 references
public void Delete(int id)
{
}
```

So right off the bat I need to add “using TodoListAPI.Database”, and “using TodoListAPI.Models” so I have access to my class objects and database context. Then I need to declare my vital database context object that I will use to access my database. I will do this by writing at the top of my TodoListController “private readonly TodoListDbContext _context;” I will then initialize this object in a class Constructor like so:

```
public TodoListController(TodoListDbContext context)
{
    _context = context;
}
```

Alright, I have my context object now I'm ready to start setting up my endpoint functions. First, I'm going to make all my functions asynchronous so I can call Async methods on my database to ensure that there are no race conditions when my database is being accessed. I will also need to change the return type from string or void to Todo on my GET methods and a IActionResult on my other methods and wrap the return types in a Task<>. When I have done this my endpoints will look like this:

```
public async Task<IEnumerable<Todo>> GetTodos()
public async Task<Todo> GetTodo(string id)
public async Task<IActionResult> PostTodo(Todo newTodo)
public async Task<IActionResult> UpdateTodo(Todo newTodo)
public async Task<IActionResult> DeleteTodo(string id)
```

My GetTodos() will return an IEnumerable of Todo data types so I will accomplish this by the following code:

```
public async Task<IEnumerable<Todo>> GetTodos()
{
    return await _context.Todos.ToListAsync();
}
```

My GetTodo() will take in an id of type string and use this string to search the table of todos for a result that matches and then return that match. I will accomplish this by the following code:

```
public async Task<Todo> GetTodo(string id)
{
    return await _context.Todos.FirstOrDefaultAsync(i => i.Id == id);
}
```

My PostTodo() will take in a Todo class object and add that object to the Todos table and save the changes then return a 200 ok object result. I will accomplish this by the following code:

```
public async Task<IActionResult> PostTodo(Todo newTodo)
{
    await _context.Todos.AddAsync(newTodo);
    await _context.SaveChangesAsync();

    return new OkObjectResult(newTodo);
}
```

My UpdateTodo() will take in a todo object and match the todo id with the existing row that is to be updated and then the row will be updated with the new object data. I will accomplish this with the following code:

```
public async Task<IActionResult> UpdateTodo(Todo newTodo)
{
    _context.Todos.Update(newTodo);
    await _context.SaveChangesAsync();

    return new OkObjectResult(newTodo);
}
```

My DeleteTodo() will take in an id of type string and then find an appropriate match and return the match to a deleteTodo variable then that result will be removed from the Todos table. I will accomplish this with the following code:

```
public async Task<IActionResult> DeleteTodo(string id)
{
    var deletedTodo = _context.Todos.FirstOrDefault(t => t.Id == id);

    _context.Todos.Remove(deletedTodo);
    await _context.SaveChangesAsync();

    return new OkObjectResult(deletedTodo);
}
```

I now have a fully functional API that can take in HttpRequests and perform full CRUD operations on its data using a postgresql database.