**Introduction:**

This is a document containing notes regarding Les Jackson’s Entity Framework video series found on Youtube here: <https://www.youtube.com/user/binarythistle/videos>.

Entity Framework (EF) is an object-relational mapper that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write. (<https://docs.microsoft.com/en-us/aspnet/entity-framework>)

A lot of the notes here will be directly taken from the video series and shouldn’t be considered as my own work. This is purely a learning project.

**User Story:**

As a Developer

I want to map my application objects seamlessly to my database

So that I don’t have to worry about manual CRUD operations.

**What you’ll learn:**

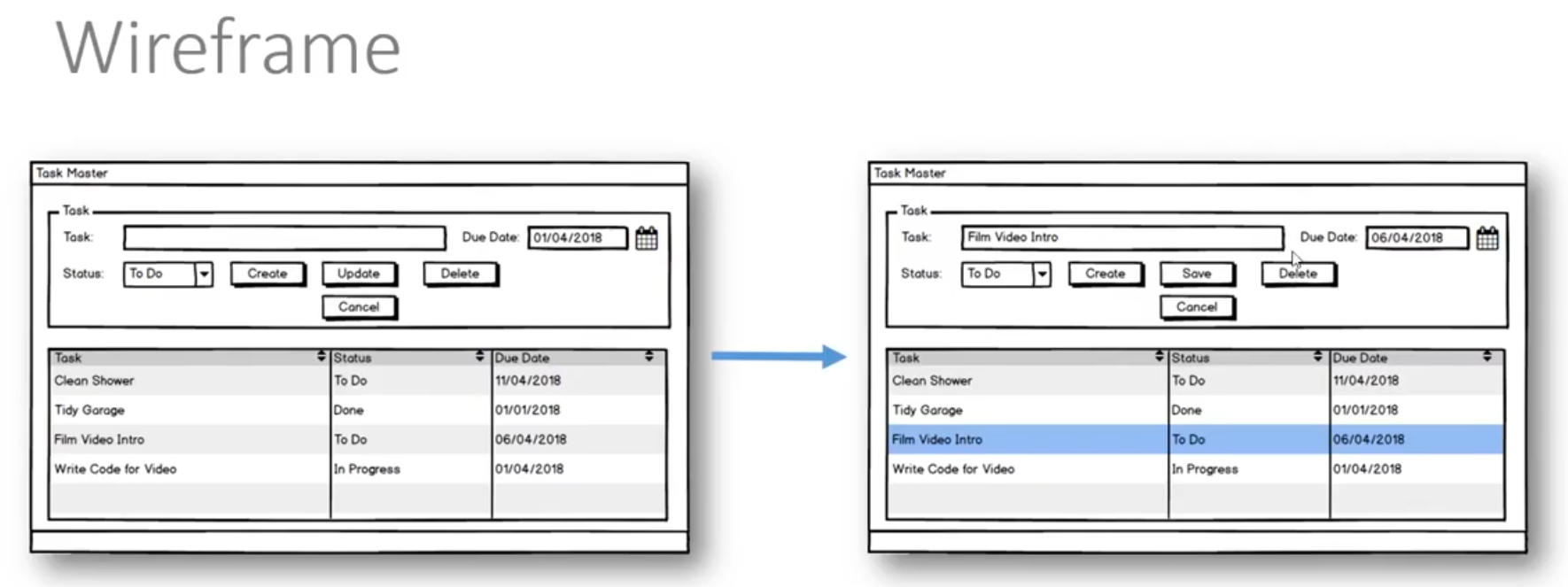
* Understand what Entity Framework (EF) is
* Understand what “code first” and “database first” approaches are
* Perform CRUD operations on a DB using C# and Entity Framework
* Understand what LINQ is and how it relates to EF.

Visual Studio Enterprise 2019 will be my IDE.

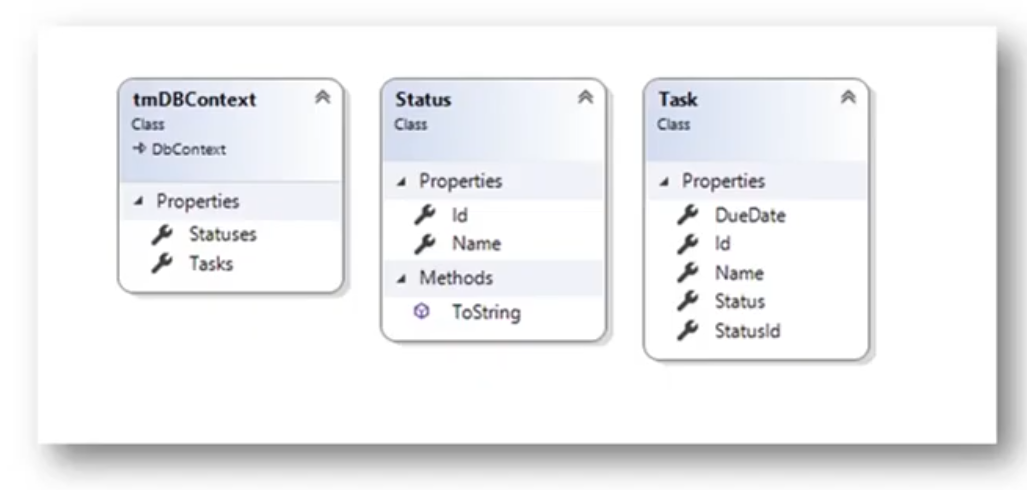
SQL Server Express will be used as my Database.

**Wireframe**

Les Jackson’s Wireframe for the project. We will be creating an application that allows its users to input tasks that will be added to a database so that they can be tracked.

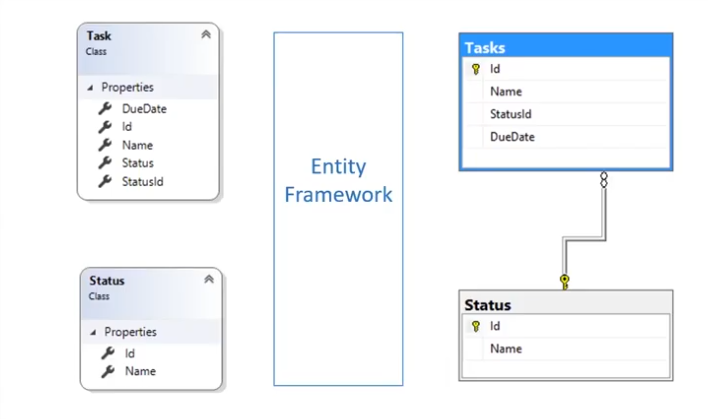


**Class Diagram**



* tmDBContext class: Very important. DBContext will be a reoccurring theme with Entity Framework (?). DBContext was used in MVC API
  + Notice Statuses and Tasks appear in this class.
* Task: Models our tasks
* Status: Models our status

**Database Schema**

****

**What is Entity Framework?**

* A “Persistence” Framework
* A layer of abstraction between the application and DB
* May also be referred to as an Object Relational Mapper (ORM)

**Advantages:**

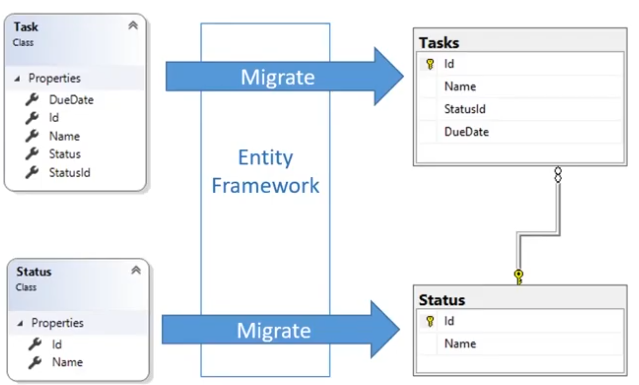
* Productivity (once you learn how to use them!)
* Application Design
* Code Reuse
* Maintainability

**Code First vs Database First**

Code First:

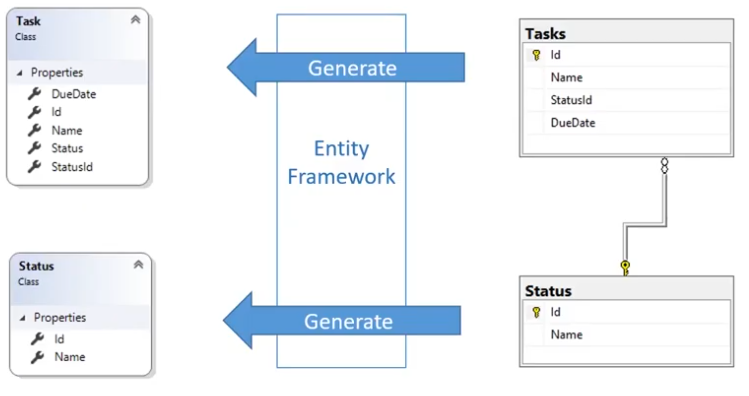
This scenario includes targeting a database that doesn’t exist and Code First will create, or an empty database that Code First will add new tables to. Code First allows you to define your model using C# or VB.Net classes. Additional configuration can optionally be performed using attributes on your classes and properties or by using a fluent API. (<https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database>)

Basically, as the name implies, you create (code) your model classes first and then migrate the classes to tables using the entity framework. (my words from listening to video)



Database First:

Basically, as the name implies, has the database already created, and is essentially the reverse of code first. You could be part of a company with a database already created, or created by a team that specifically works on databases, or you could be working with a third party database you want to access and use. You generate your code (model) classes from the database schema.



**Which is Best?**

A bit of a preference and situational discussion. As with most things with coding, it depends. In this video we will be taking the code first approach.

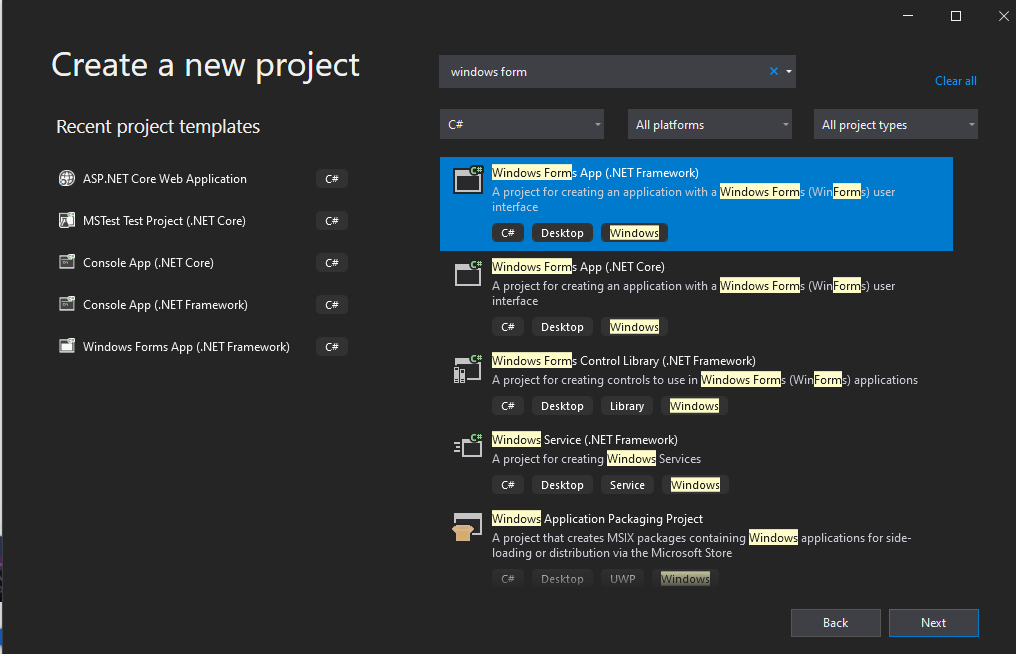
**Alternatives to Entity Framework:**

* LINQ to SQL
  + Also, from Microsoft.
  + Team developing LINQ to SQL ended up merging with Entity.
* nHibernate
  + Widely used
  + Object-relational mapping for .NET platform.
  + Port of Hibernate
  + Open source under GNU Lesser General Public License

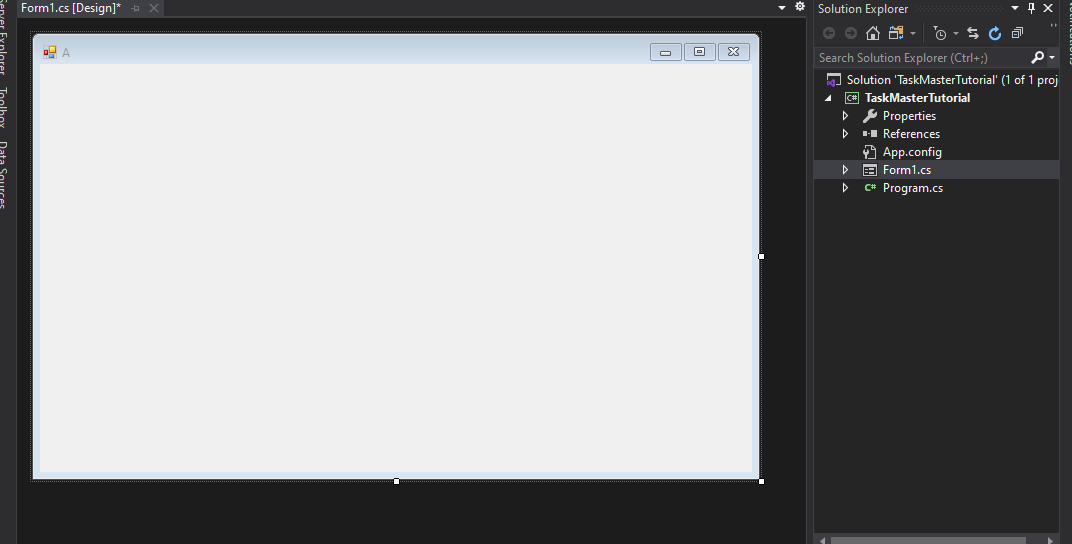
**What is LINQ?**

* Language Integrated Query
* Is NOT a persistence framework
* It’s a ubiquitous way to query data from within your code.
* You can use the same syntax to query different data sources, such as
  + XML
  + Relational DBs

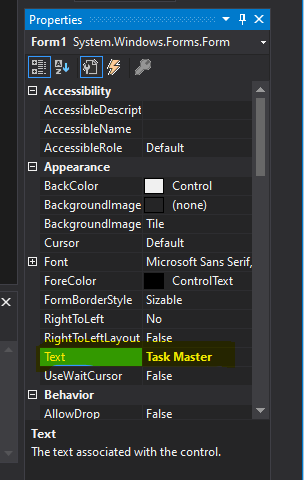
In Visual Studio



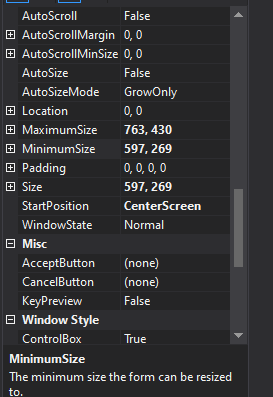
Using the .NET Framework project type, in order to follow video. Will look into trying the .NET Core type seen in screen shot later.



Clicking on the form should open up a Properties window seen below.

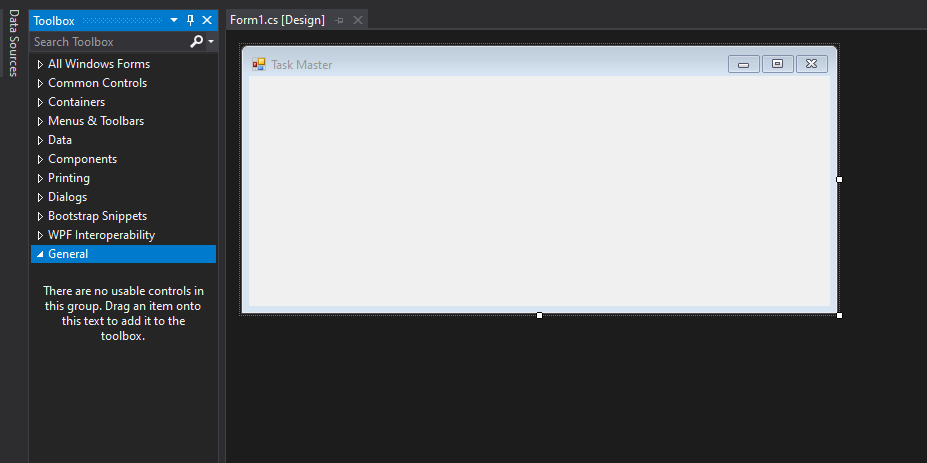


Change the Text box to Task Master

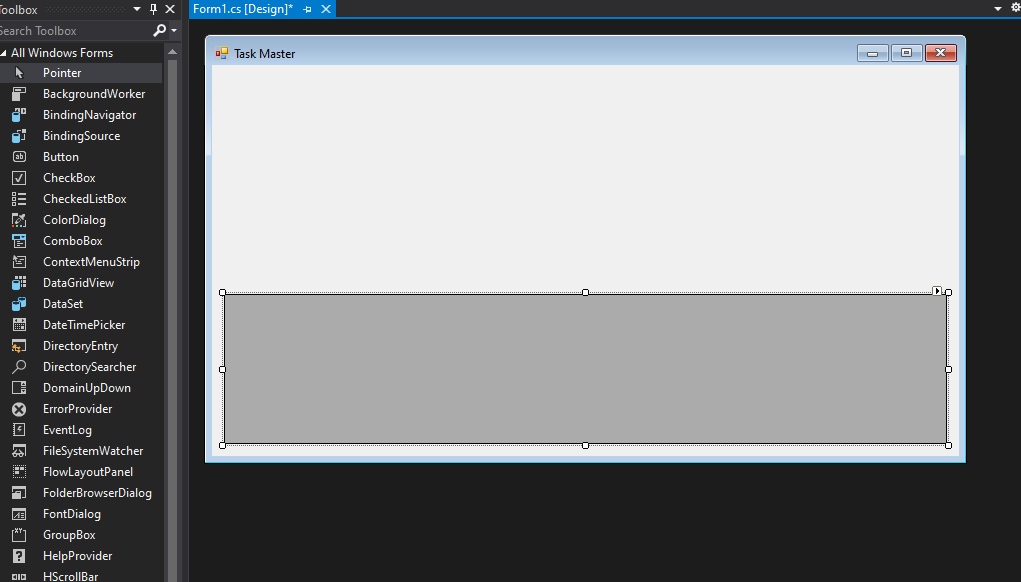


Following along with the video I set start position to center screen and set a minimum and maximum size. I didn’t set the exact same size as tutorial just arbitrarily chose a size, I thought decent.

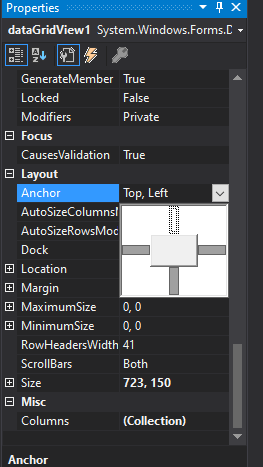
Open up the toolbox



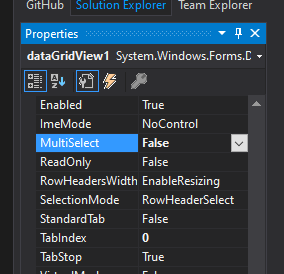
“Data grid view”



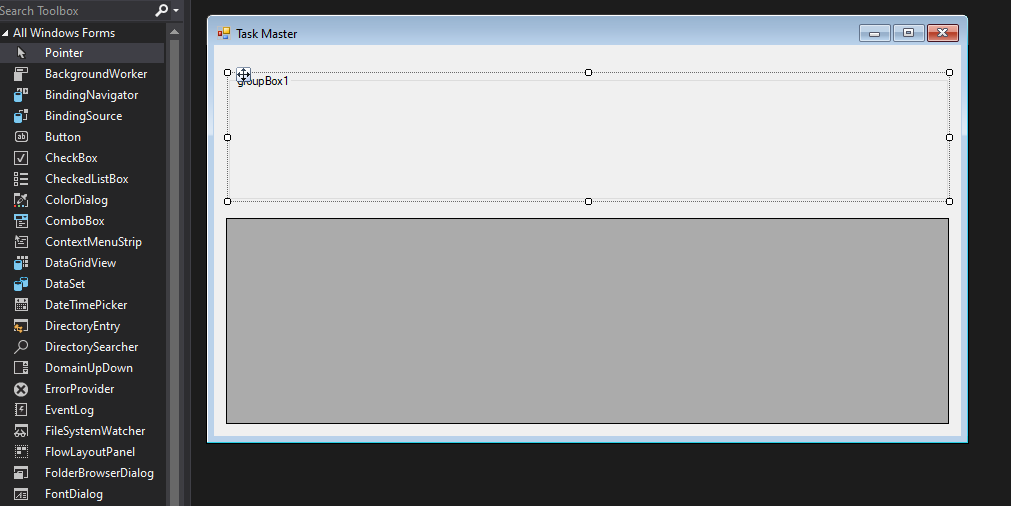
In data grid view properties set anchor options left, right, bottom



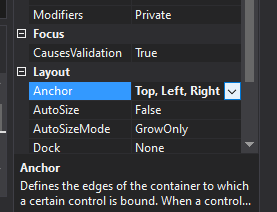
Multi select to false



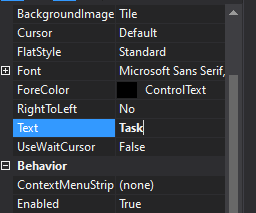
Add “group box “now

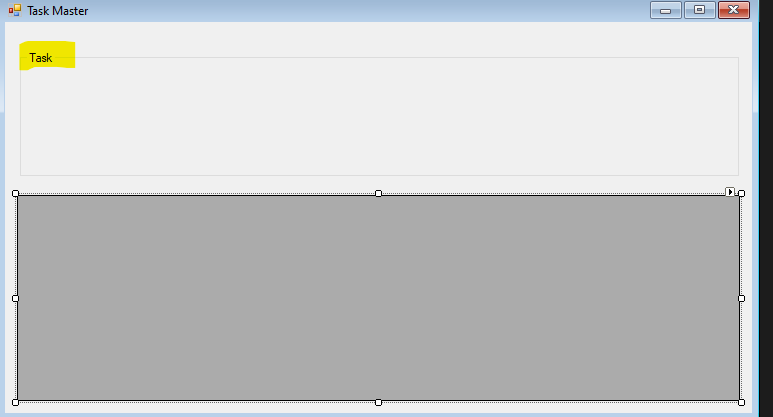


Anchor top left right

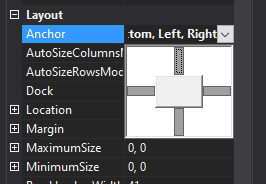


Set Text to “Task”

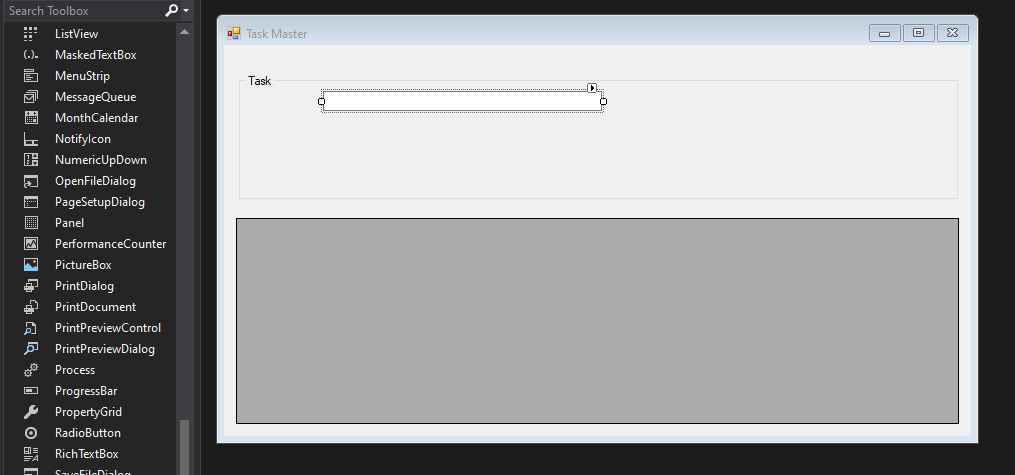




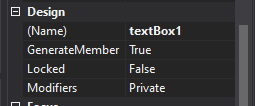
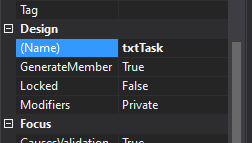
Add the top anchor to data grid view



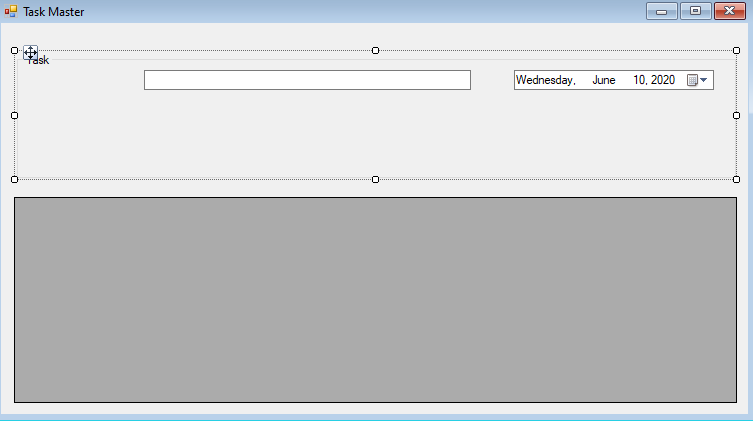
Add Text box to group box.



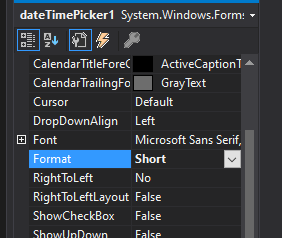
Change (name) row to txtTask. In the video his properties looked slightly different. I think I changed the right property.

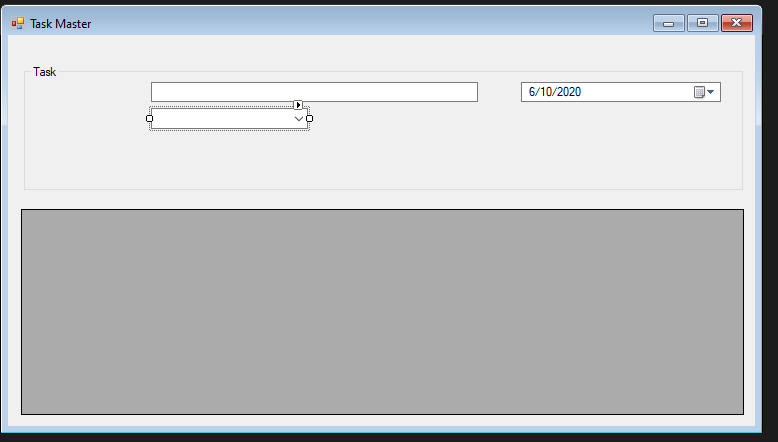
Now add “DateTimePicker”



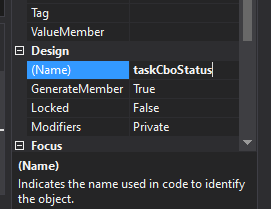
DataTimePicker property “Format” from Long to Short



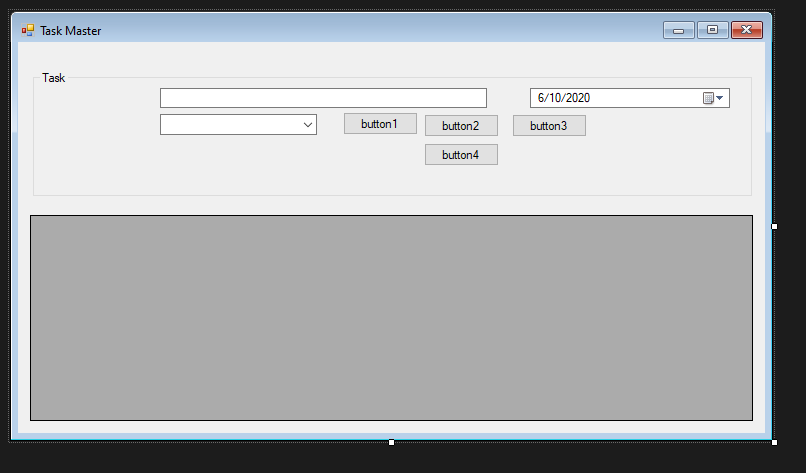
Add “combobox”



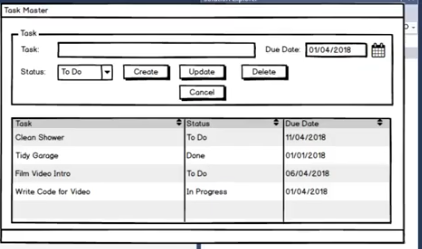
Propert “(name)” to taskBboStatus



Add 4 buttons

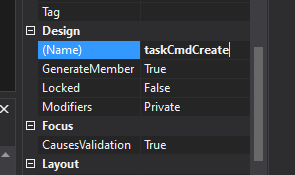
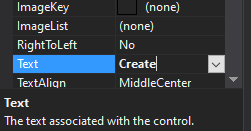


We are creating the buttons from this wireframe…

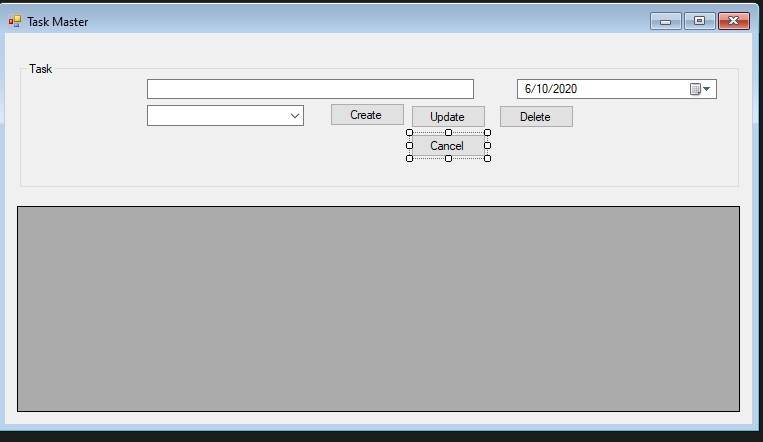


(name) will be taskCmdCreate, taskCmdUpdate, etc.

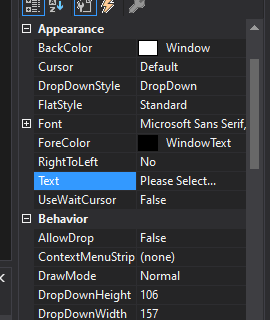
Text will be Create, Update, etc.

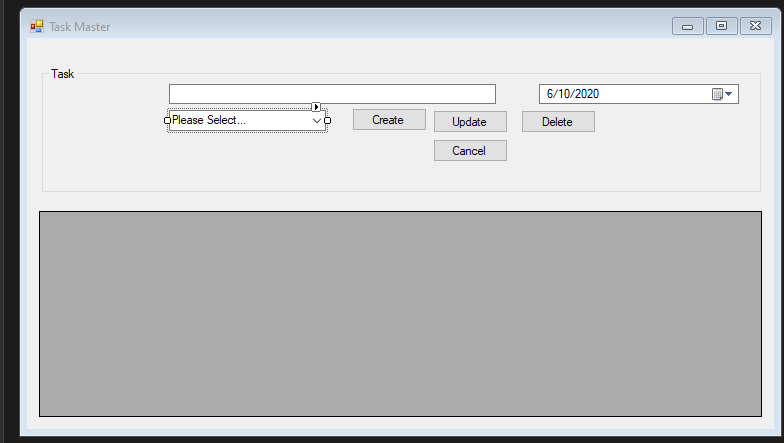
 

Set properties for all buttons

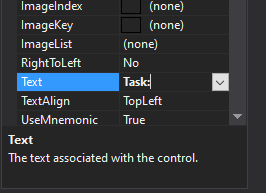


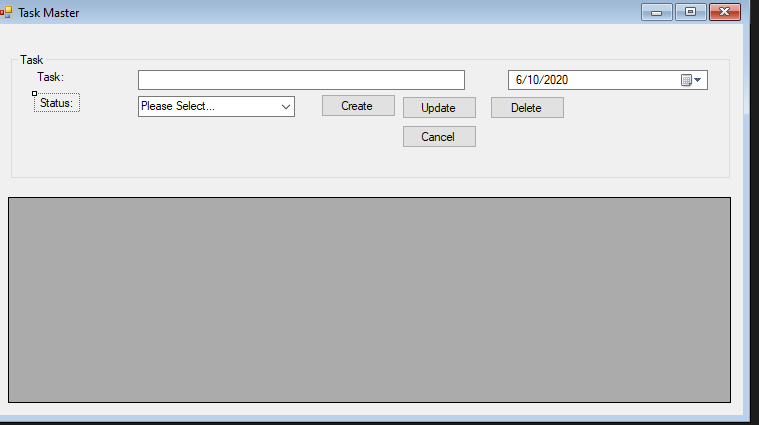
Select the combo box and change Text field





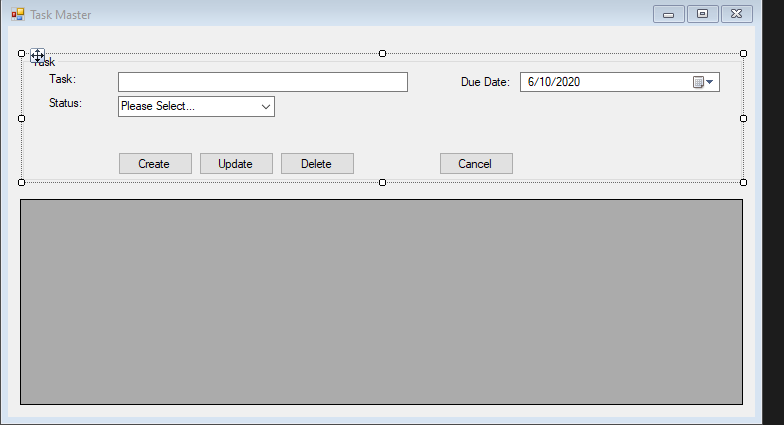
Add 2 labels. One for Task: and one for Status:





Add third label for Due Date:

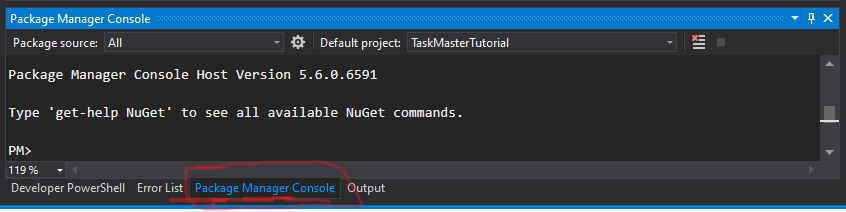
I readjusted some items.

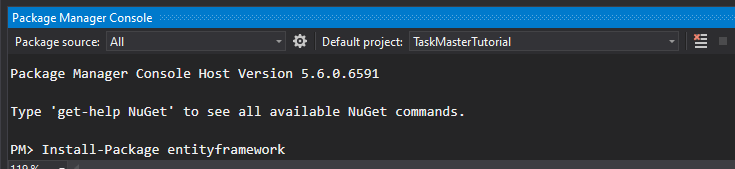


**Set Up Entity Framework**

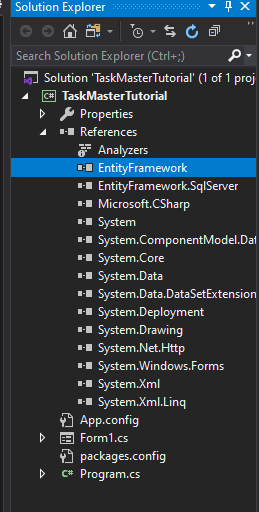
* **Install Entity Framework**
* **Create DB Context Class**
* **Update app.config file**
  + **Holds connection string to database**
* **Enable Migrations**

Open up the package manager console. (Mine was tabbed near the bottom. Could also find it in Tools -> NuGet Package Manager -> Package Manager Console)



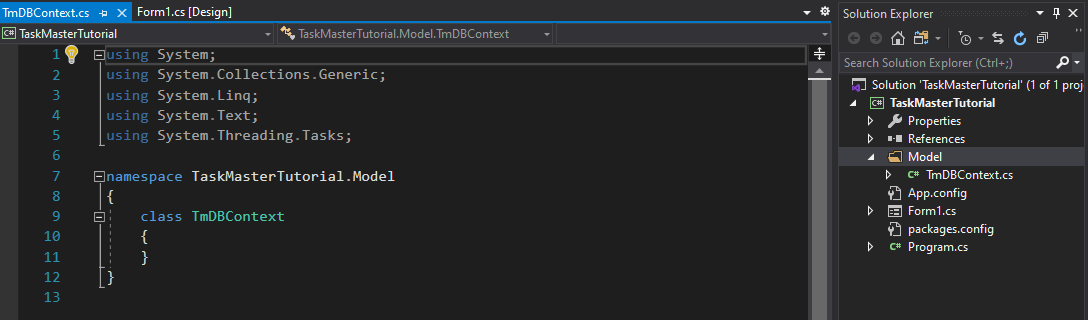


And we can now see entity framework in our references folder section

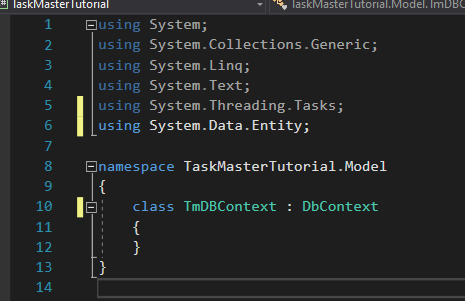


Now create a new folder named “Model”. Model will hold our DB Context class. Model folder will be added to “TaskMasterTutorial”.

Create a new class in Model folder named TmDBContext.cs. Tm stands for Task Master (fyi)

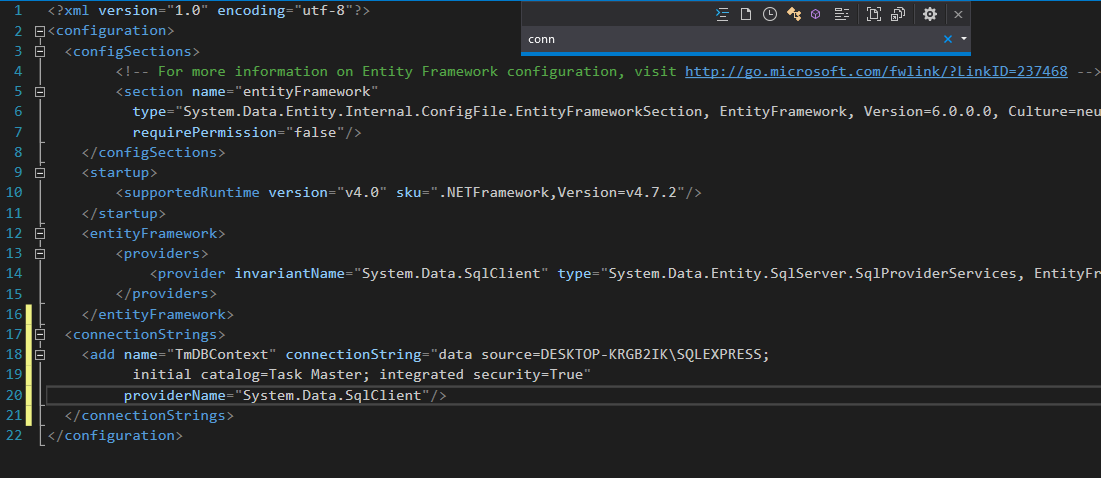


The initial set up. Inherit from DbContext



Next go into app.config file.

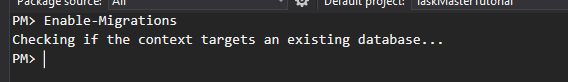
We added the <conncectionStrings> section



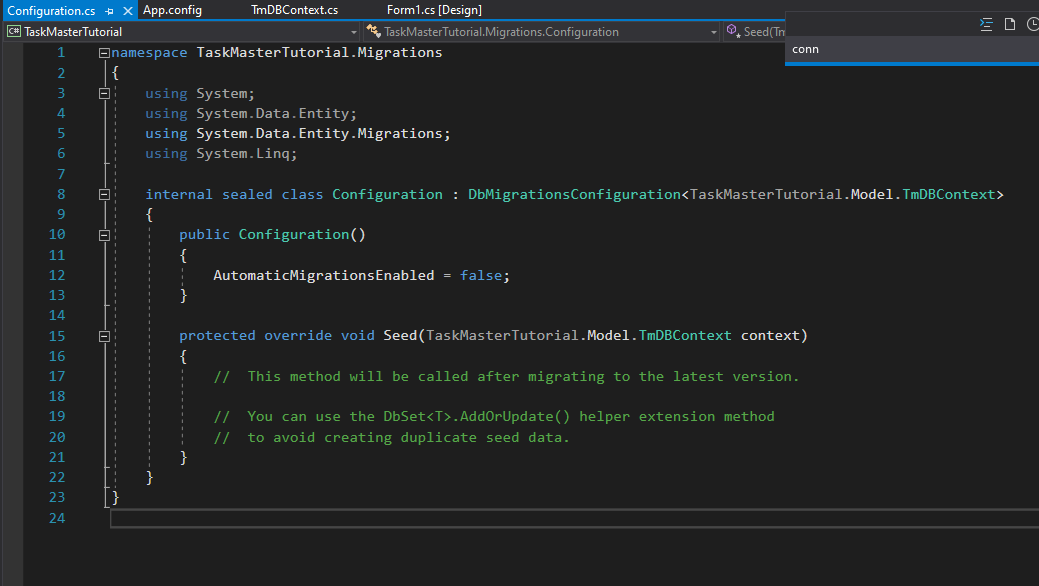
Data source will be unique to you. Integrated security = true just means it logins in with your current credentials, and could be changed or may need to be changed depending on your system and database.

Now go back to the Package Manager Console and “Enable-Migrations”. This will allow us to migrate our model class into the database to create our Database and tables. Initial Catalog from the Connection String part will name the database (aka catalog).

I believe if the connection string is wrong this part will fail. Not verified. Worked on first go.



This class should appear in a Migrations folder.



No catalog should appear in your server.

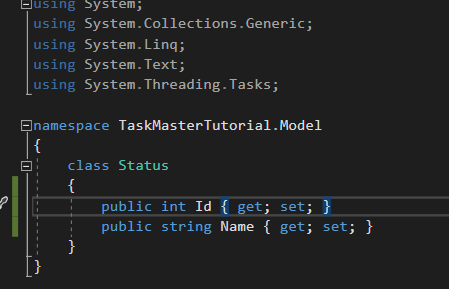
Only need to enable migration once.

**Create Status Class**

* Create Status class
* Add Status Class property to DBContext
  + DBContext can be thought of as a representation of the data base
* Create a Migration
* Update our database
  + by running migration

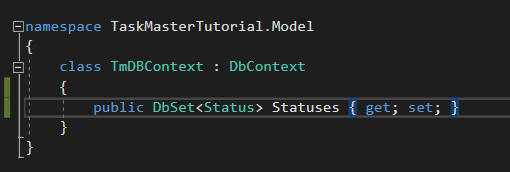
Add a new Class file names Status.cs to the Model folder.

Our initial class…



Now we want to add our Status class as a property, to our DBContext class (TmDBContext)

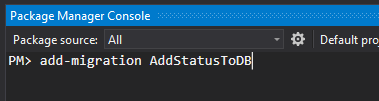
Add the following line to the DBContext class. DbSet can be thought of as a representation of a table from our data base.



Now we will migrate our data to the database. This is workflow of migration is and will be repeatable when working with entity framework.

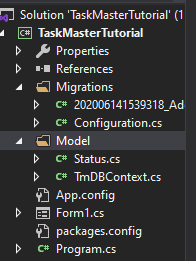
In Package Manager Console type command “add-migration <name of migration”

We want to give our migration a meaningful name. Think of it as self-documenting name in a way.

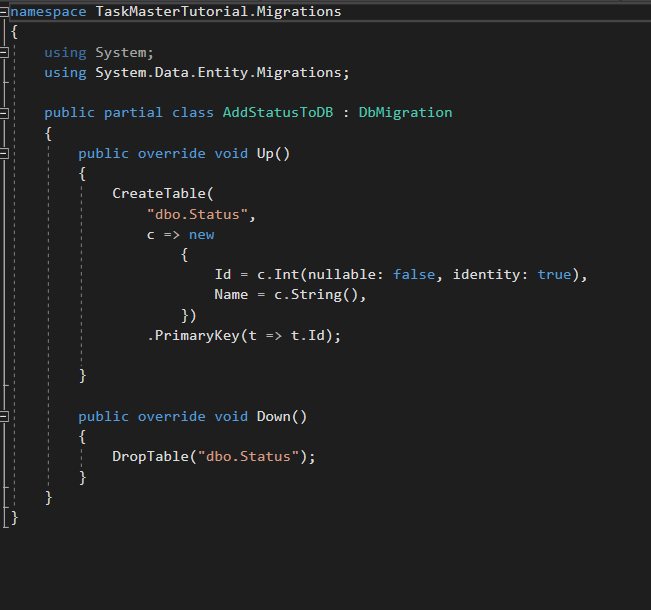


SQL Server must be running for the migration to work.

A Migration folder will appear.



The Migration file (which is started with the creation date in this case 20200614….) contains the information to make the changes of our migration to our database.



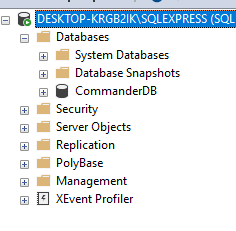
In this case the Up, creates a table with its two fields, and the Id is the primary key. Kind of self-explanatory from the naming conventions.

We still have to run the migration after adding the migration.

To do this, in the Package Management Console, enter command “update-database”

You can look at your database and see the before and after.

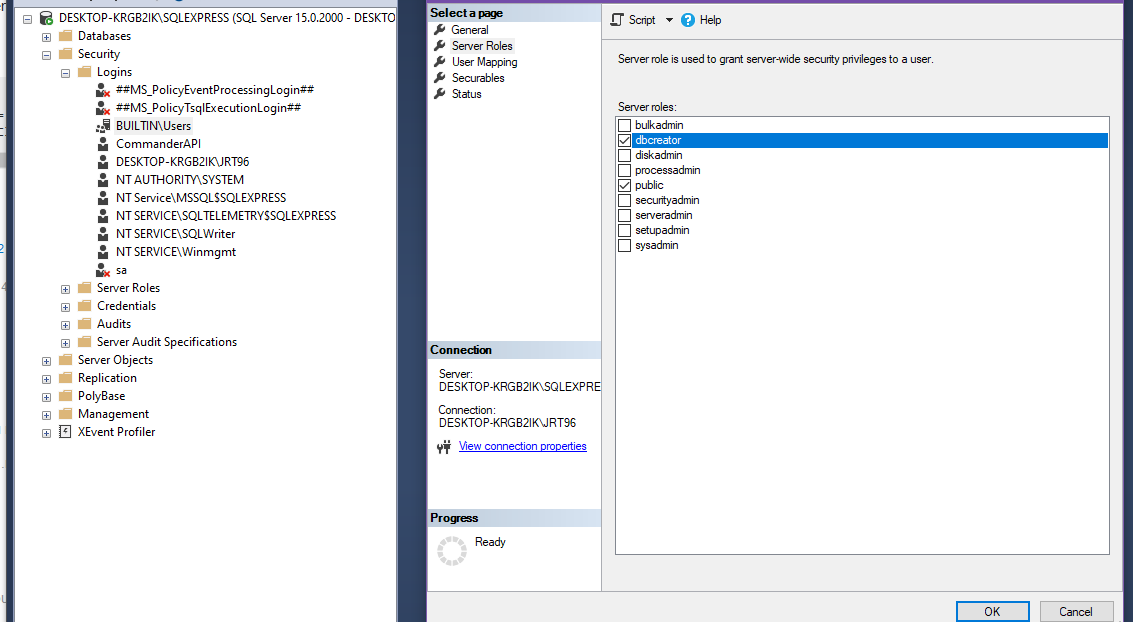
Currently, no migration has occurred, so no catalog is present for this project.



Failed with error

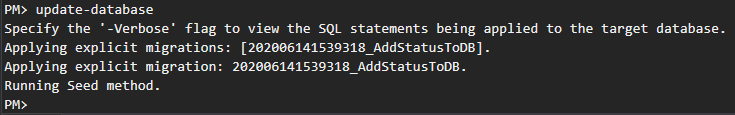


I had this similar issue on the Web API project I was working on (also Les Jackson tutorial).

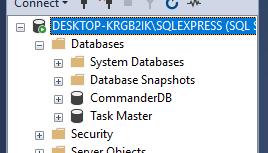


Run SQL Manager as admin, and login with Window authentication.

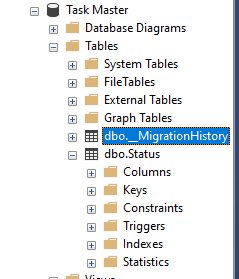
On the left hand, we opened Security folder, then Login folder. I actually changed Server Roles for both Desktop…\JRT96 and BUILTIN\Users. I think I just needed to change the roles for BUILTIN\Users since I am running as the non admin account (JRT96 is admin). Added the check mark for dbcreator (seen on right hand side.). This enabled us to run “update-database”



And we now have our Task Master catalog (database) available



We can see our dbo.Status which is our Status table (database object).



This is our first initial workflow for Code First migrations. We:

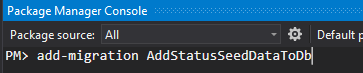
* Created a Class
* Added to DB Context
* Created Migration
* Run that migration

**Seed Data**

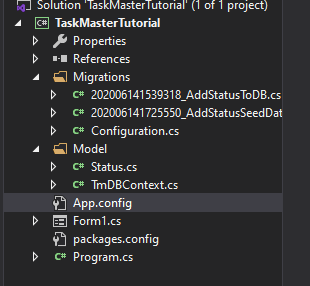
* Create empty Migration
* Add SQL INSERT commands
* Update our Database

We could use our SQL manager to insert items into our database, but we are going to embrace our Code First approach, and add a series of items, and gradually build up our database. This will allow us to run the same code against another database and populate it the same as ours. It also means any reference data is already there.

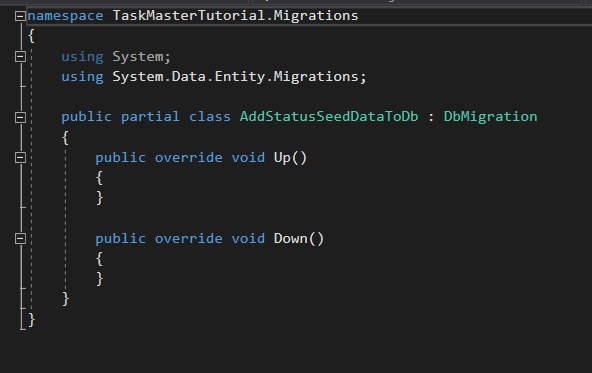
So, to do this, we want to add another migration, remember to give it a meaningful self-documenting type name. In the Package Manager Console “add-migration <name>.



And in our solution explorer we can see the new migration file



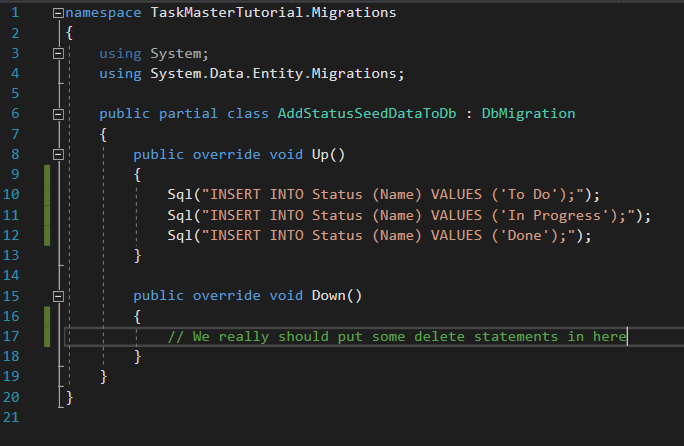
Notice in our new migration file (class?) that the up and down methods are empty. That is because there have been no changes made; our previous migration had been run and since no changes were made.



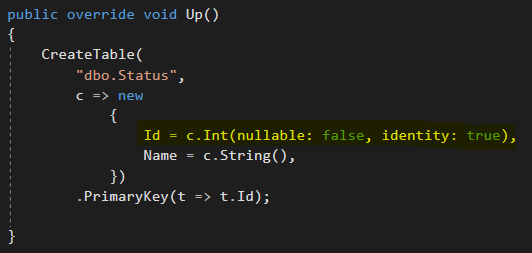
This allows us to manually enter information into our migration class. This give us some advantages over using our SQL Manager such as being able to quickly put up databases such as in the case of replication for development or whatever.

This is “leveraging our power of migrations”

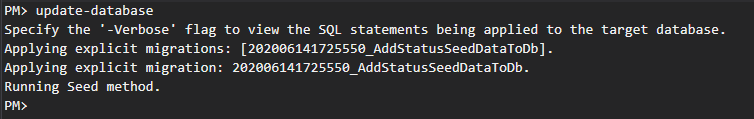
Our current migration class is as follows.



Side note: Look at our previous migration. The line highlighted line makes it so that every item entered into the database is given and Id and is incremented by 1, so you don’t manually have to assign an Id. Specifically the identity = true I believe sets this attribute (I believe nullable = false would also be required but is not verified.)

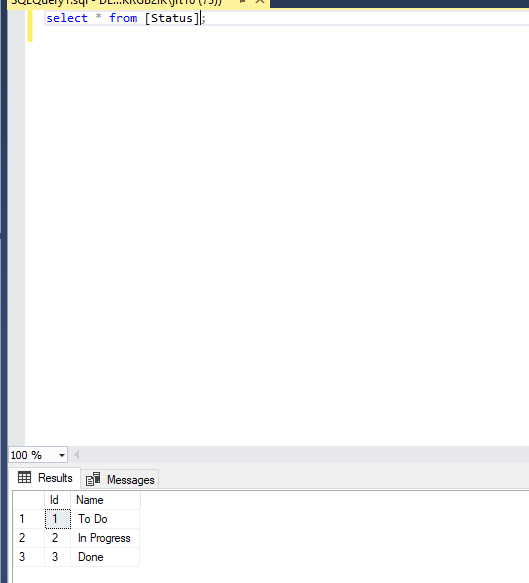


Now we need to update our database.



Now taking a look at our Manager we can see our entries into the Status table.

Status is actually a keyword in SQL, so we delimited it with brackets. I actually tried without delimiting the word, and it still worked, but I am following along with Les Jackson so I will do as he does.



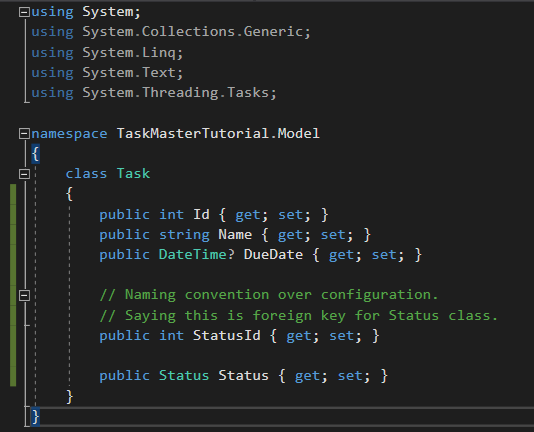
**Migrations Gone Wrong**

* Create our Task class
* Create a migration
* Delete a migration
* Overwrite migration
* Roll back migration

We want to practice how to work with migrations when they go wrong.

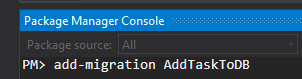
Add new class “Task” to our Model folder.

Our Task class will look like the following

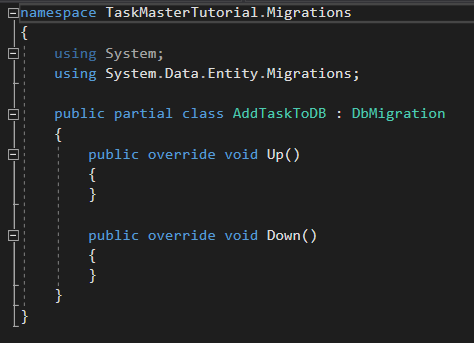


We actually need an attribute for both Status and StatusId. There is a technical reason for this according to Les Jackson, and he has a link on his blog that goes into the detail on the reasoning. He says it is confusing so he doesn’t try to explain it, and that is why he provides a link. Not personally going to research the details, just going to assume that it is dues to need to have a separate variable for the Id for reference purposes, instead of just using Status.id.

Now “add-migration <name>”



Should see the migration in the migration folder, and the class for the migration.

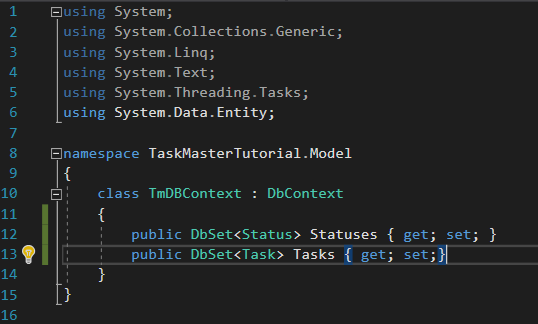


**But why is nothing in the Up method??**

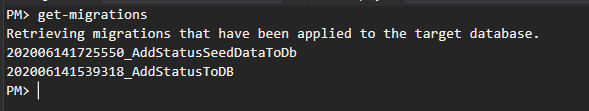
The reason is because we did not add it to our Db Context class.

This is the important of the DB Context class. DB Context says what classes need to be added (or manipulated). This distinguishes which classes are referenced in our database and which classes are not referenced in our databases (my own words, so may not be technically correct, but is how I am thinking of it).

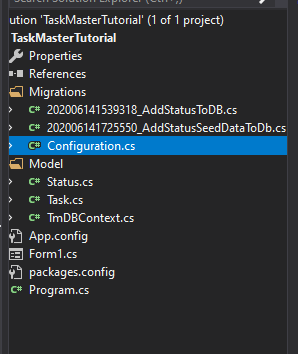
Add the Task class to our Db Context class. DBSet represents a table which will be of the type which our class references. (my own words)



Now what do we do with our previous migration. In this instance we can enter “get migrations” into the package management console.

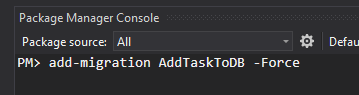


We can see the migration has actually been applied. It has only been created. So all we need to do this simply delete the migration (AddTaskToDb) from our folder (in the solution explorer).

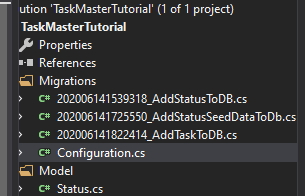


Now generate a new migration. I used to up key to get previously run commands on command line (PMC).

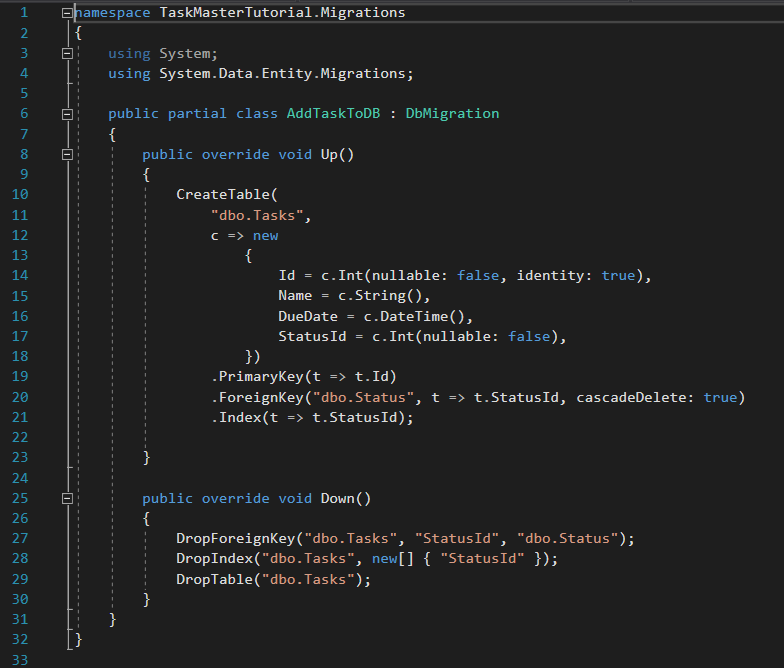
If we had not deleted the previous migration, we could use the -force keyword to over write the migration. This is necessary since it was deleted but we will leave it in there, as it won’t hurt anything.



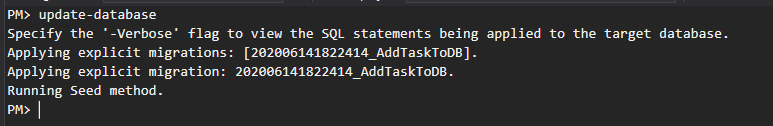
And we have our new migration.

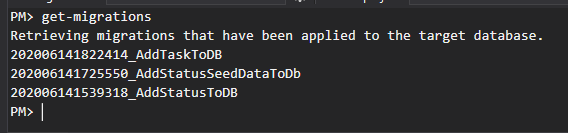


Notice our Up and Down methods are now populated.



Now “update-database”



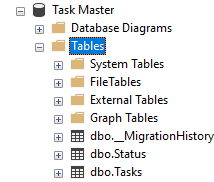


If we wanted to rollback a migration we would “update-database -targetmigration <name>.

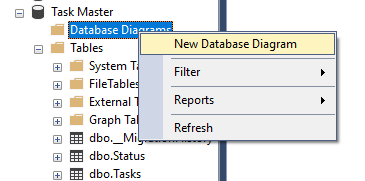
So in this example if we wanted to rollback to the migration before the one we just ran i.e. AddStatusToDb, we would issue command “update-database -targetmigration AddStatusSeedDataToDb”

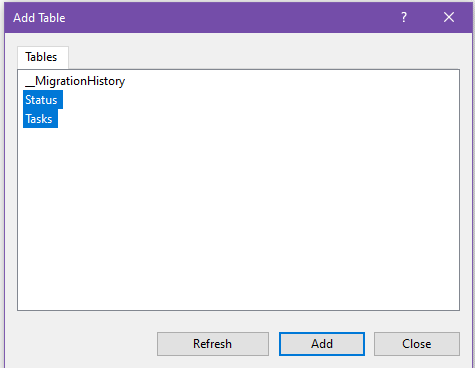
Les Jackson says it would effectively run the Down() method.

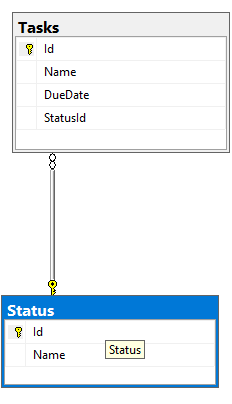
And we can see our new table in our database manager



Now we will utilize the managers “create database diagram” so we can visually see our database (schema?).





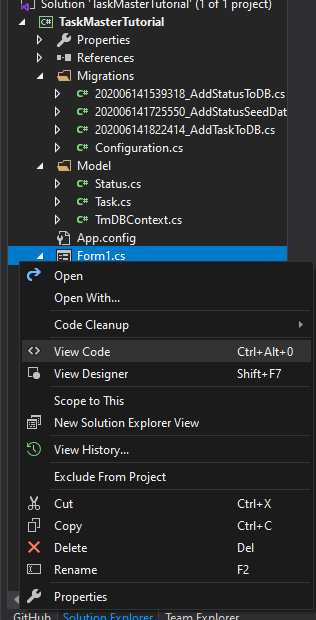


And we can see our relationship between our two tables.

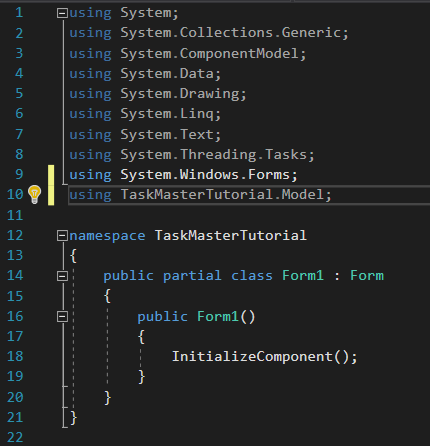
**Reading Data From our DB**

* Add a TmDbContext instance to our form
* Get a list of our statuses
* Write them into our Drop Down

First we need to add to our form1.cs source code. View Code



Add using TaskMasterTutorial.Model so that we can reference them in our form.



Les Jackson says Status and Task are probably not the best names for self-created classes as they are reserved words “all over the place”.

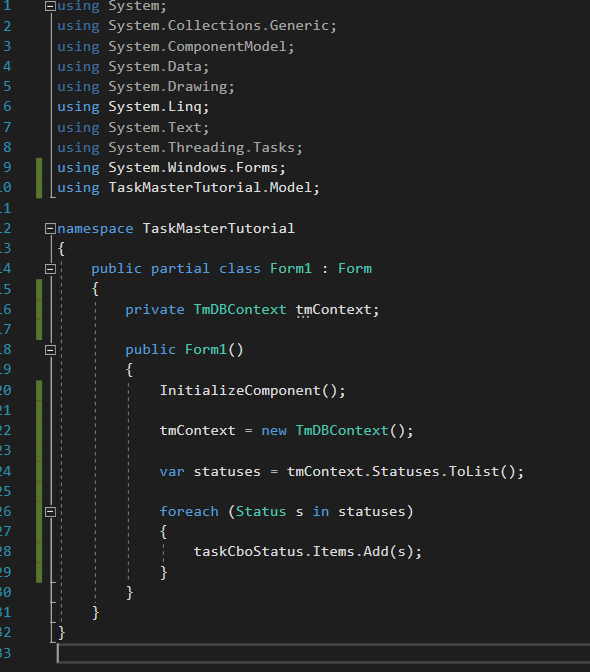
In our form1.cs source code we need to add some lines of code.

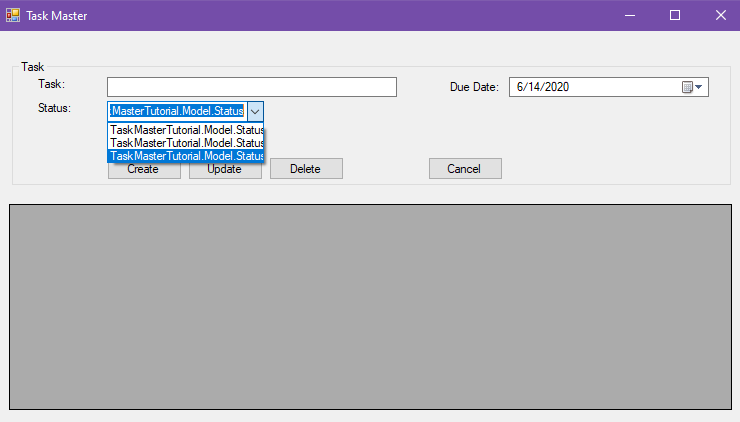
We create a private object that references our DBContext.

We initialize our DB Context instance in the constructor.

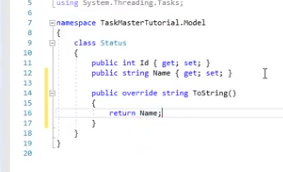
We create a var to hold a list of the statuses from our Status class, and for each status in the Status table we add it to our combo box.

Now we can see in our actual form when it is ran we get the name of the instance and not the actual status. So we need to make an adjustment to our code to get the name.

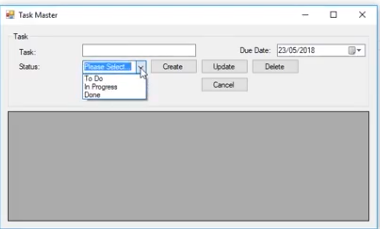




Les Jackson gets the name by adding an override ToString() method to his Status class.

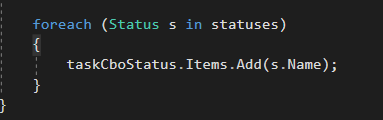


Which yields this result.



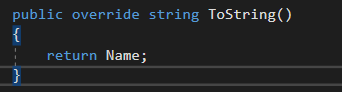
Maybe my method will run into issues further down the road, but I prefer to just reference the object attribute with s.Name, rather than overriding the ToString method like Les Jackson.

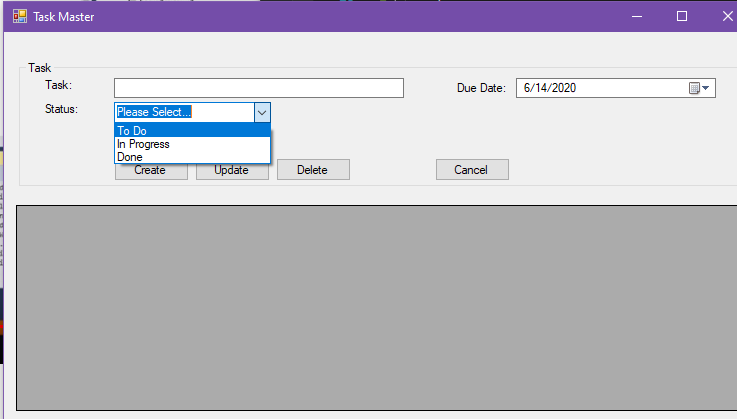
In form1.cs source code my foreach loop looks like



And my results (i.e. form when the program is run)

Ok, so the override was done for a specific reason. You could possibly work around this in a different was such as not needing the ID of the status, since status is essentially an Enumerable. I’m not really sure why it was done this particular way. Regardless we want to add s not s.Name, because we want the entire object added to the list so we can reference its Id later. Add the following to the Status class.



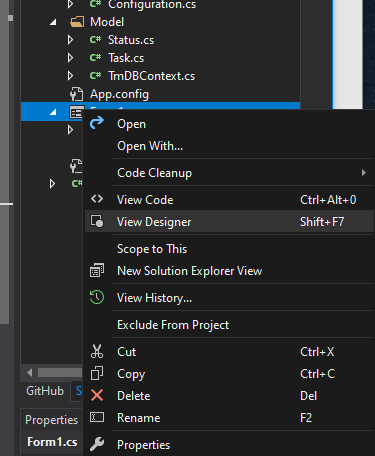


Appears to be the same results. I may have to make some revisions later, but I will stick with this for now.

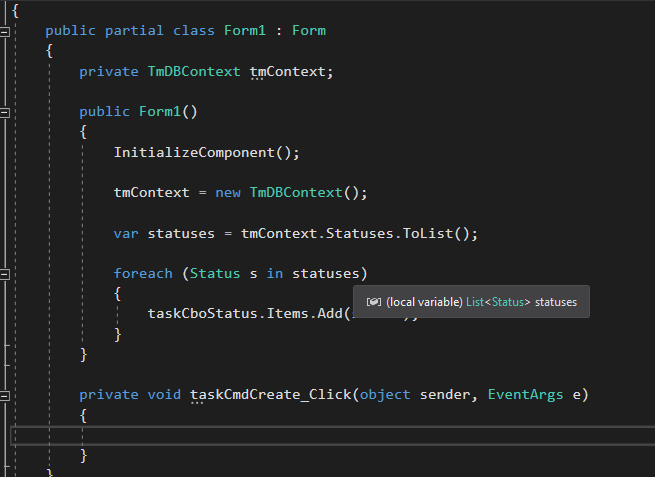
**Creating Data In Our DB**

* Wire up our Create button event handler
* Create a new task instance
* Update and save to database

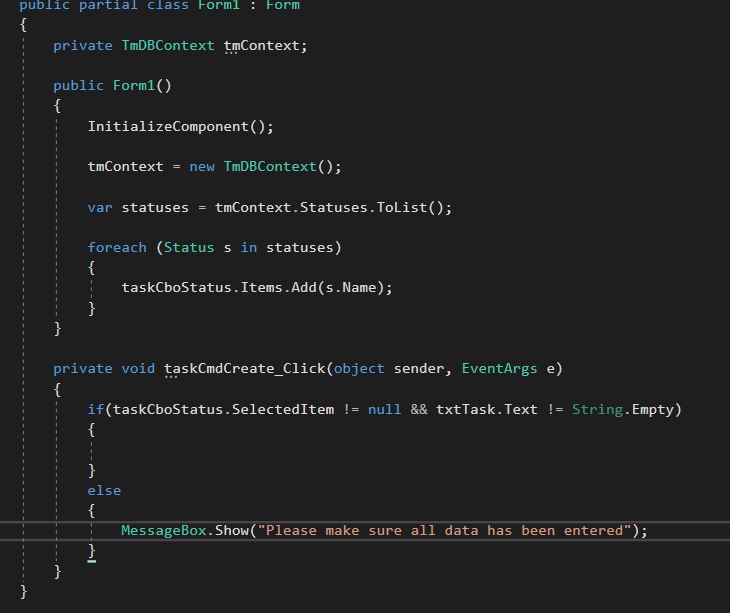
Now go to our form1.cs view designer similar to how we opened our form1.cs view code



Double click the Create button to open up our Create click event view handler.

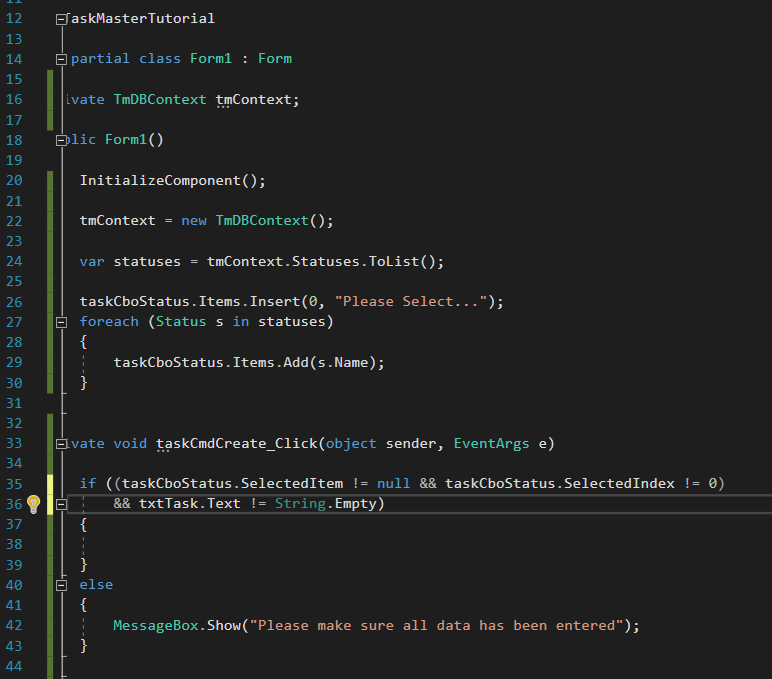


Add the following if else statement



And run the program to test the else portion (i.e. select item is null and string is empty).

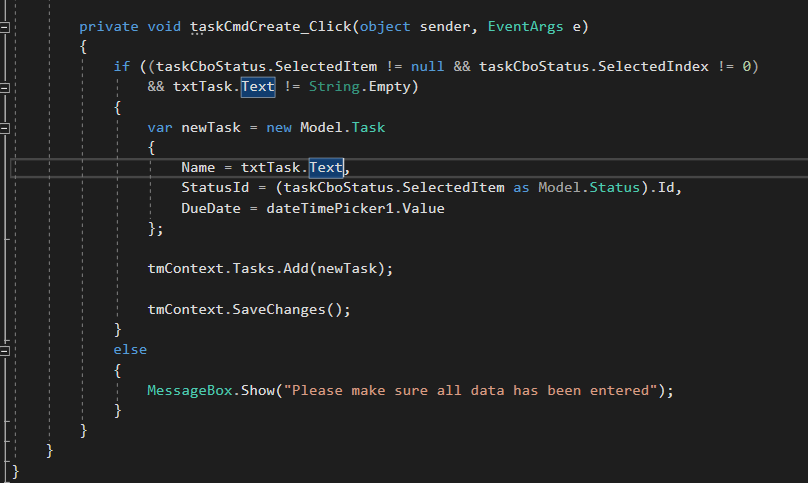
Made some slight modifications to the logic.



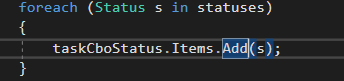
Now let’s handle the when true condition result.

And our method looks like the following.

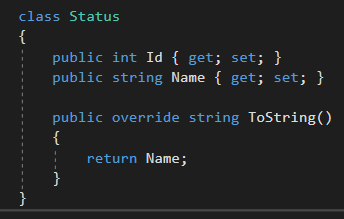
We create a new Task object and assign values to the fields using the information submitted in the form. The StatusId is a bit confusing. Not sure how it would get an Id



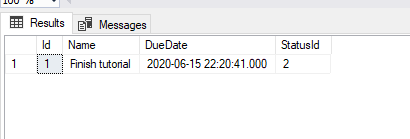
So earlier when adding items to combo list I added s.Name instead of s. I went back and changed this because we weren’t able to reference the Id since we only added the name. So add(s)



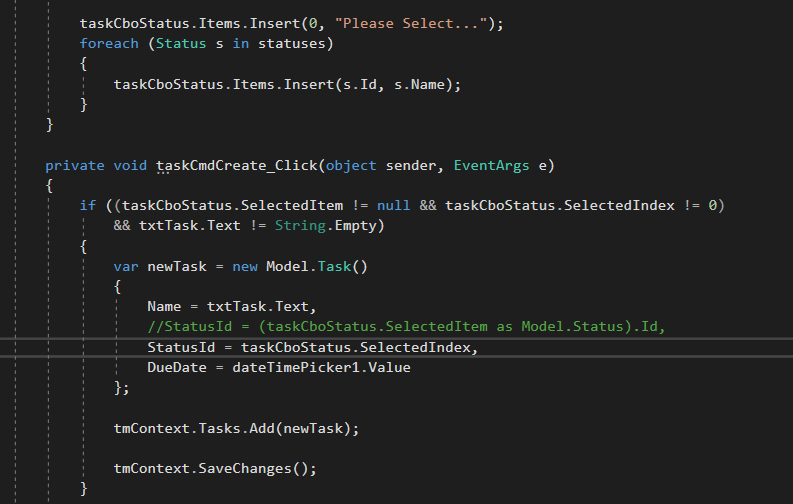
And we need an override ToString() for our Status class.



And we can see our task in the Tasks table from our database



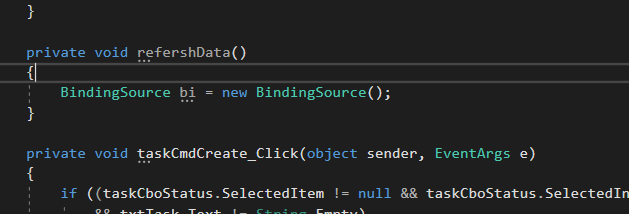
I actually used cboStatus.Items.Insert(s.Id, s.Name) and then StatusId = cboStatus.SelectedIndex and removed the override ToString() method.



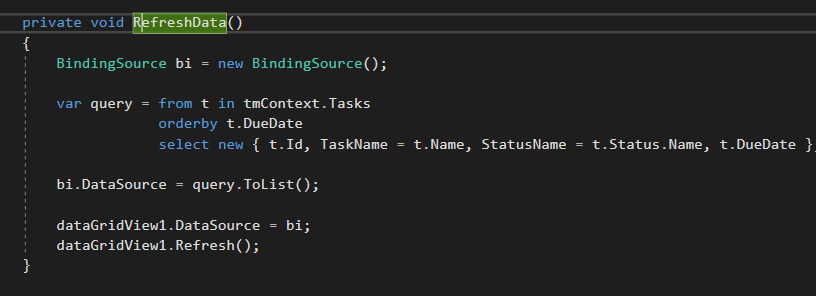
**Reading Data from our DB (pt2)**

* Create a Binding Source
* Create a Linq Query (Query Method)
* Bind to our DatagridView
* Refresh
* Linq: Query Method vs Extension Method

Add a new method for refresh in the form1.cs class. In it we will create a binding source which acts as an intermediary for our data to bind to and be worked with. This is a recommend by Microsoft, and further reading is encouraged.



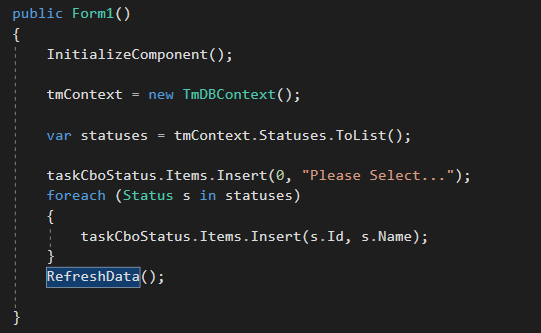
Now for the rest of our RefreshData() method. (Made a spelling error and capitalization error in the above picture.

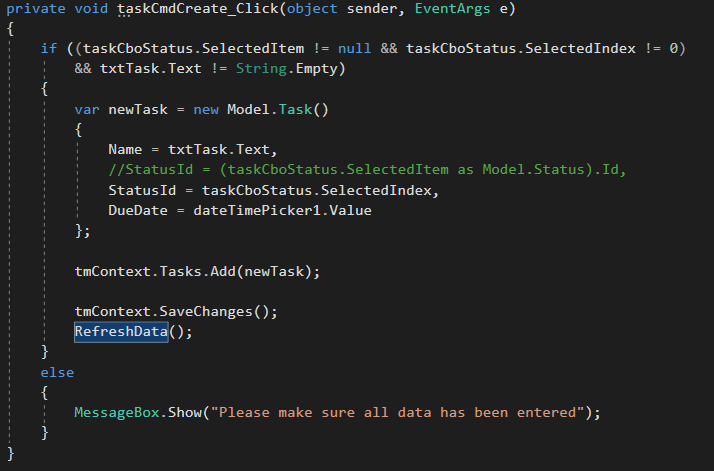


We create a query. One thing to notice is we have the select statement last compared to traditional SQL statements. Not exactly sure why that is except that is the precedence when writing in C# or Linq (not exactly sure). Also notice we use TaskName = t.Name and StatusName = t.Status.Name this is because the statement found using .name ambiguous.

After that we create our binding data source. Essentially putting our list, we generated from our query using our DBContext from Entity framework to our binding data source. Afterwards we add it to our dataGridView and then refresh it.

We also call our RefreshData method when we first construct our form, and we create a new task. It will also be utilized when we delete from our DB which will be seen shortly.



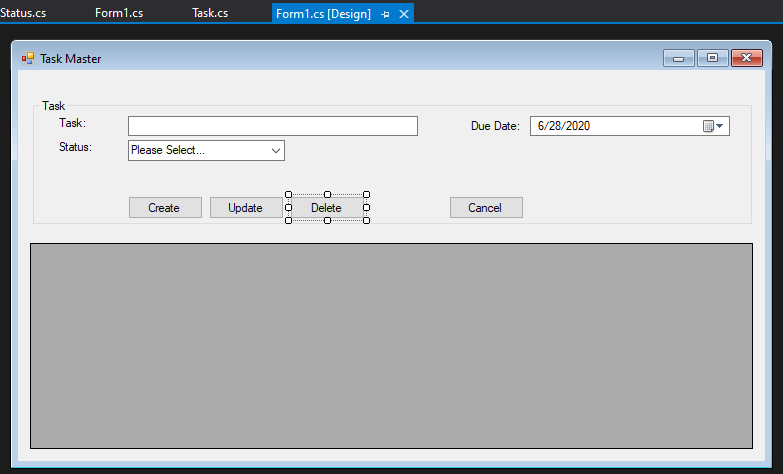


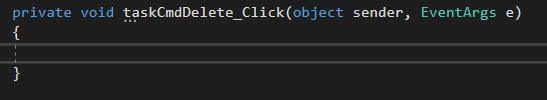
**Delete Data (pt 2)**

* Find our selected Task in the DataGridView
* Remove from our DB Context
* Save

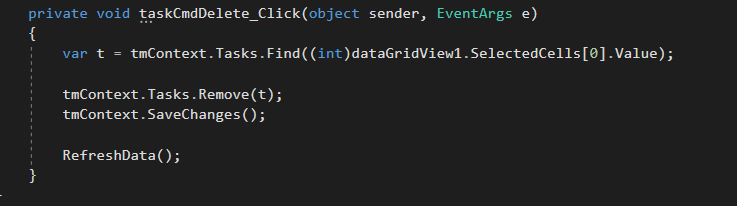
Remember our DB Context is an intermediary, and changes are not saved to our database until we savechanges().

Open up the form1.cs[Design] class and double click the delete button to view the code.





Pretty self-explanatory. Create a variable t and we find the corresponding task with the selected rows Id value with is column 0. We then remove(t) from our DB Context class, save changes and then refresh the data in our data grid view.



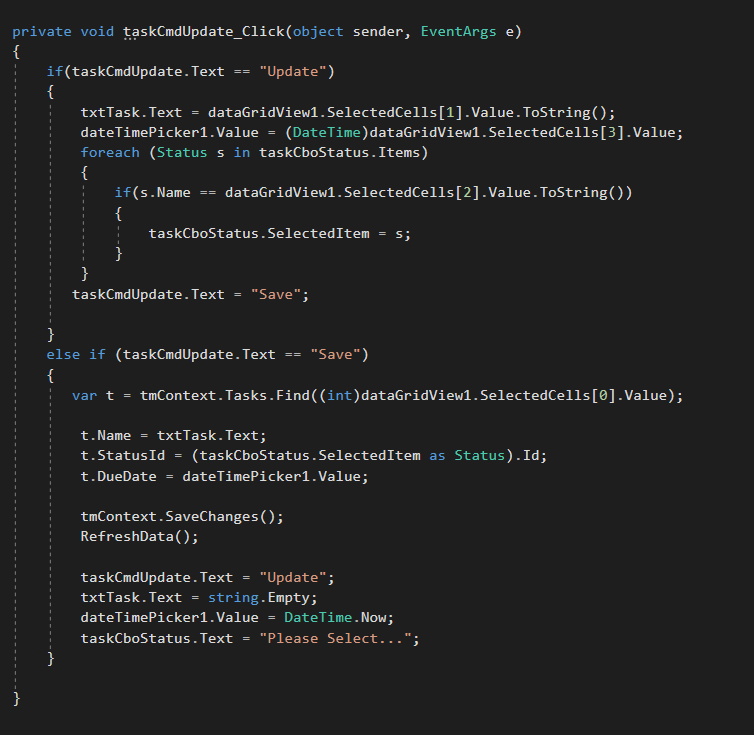
**Updating Data in Our DB (pt2)**

* Find our selected task in the data grid view
* Populate our form attributes
* Update attributes
* Save

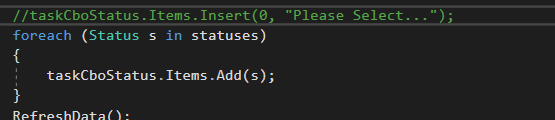
Double click the update button to generate the method in our form1.cs class.

The changes I made earlier where I Inserted s.Id and s.Name instead of add(s) caused some problems here. I ended up changing it back to how Les Jackson had his. I’m sure there was a work around to implement it following the original changes I made, but I don’t see the need to do so at this time and it’s easier to make the changes to match his working program. This caused some problems such as some rows in the data grid view won’t work with the update. I need to go back over the program and smooth out the details.

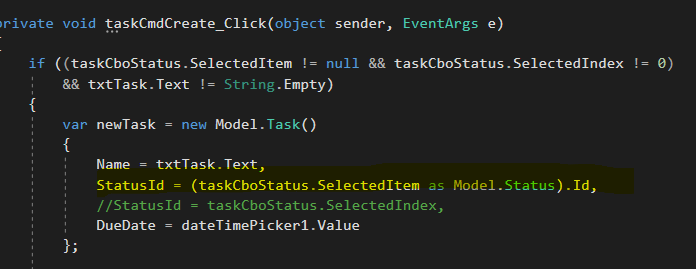
Our update method.



We had to make some changes here



And here



Possibly other places but this worked for now.

Still needs to smooth out validation checking, and some bounds handling such as deleting when no items are in data grid view, the cancel button and more.