# Why I chose to use Naked Objects Framework

I chose to use Naked Objects Framework for multiple reasons. The biggest advantage for using it is its UI. The client side of NOF is brilliant for database orientated projects as the UI will automatically display objects in a tabulated way similar to how it would look in a database. It also separates object methods and adds them to the actions button on the UI. This is all customisable from inside the objects. The UI’s colour and layout can all be customized, this includes individual objects and everything.

NOF also contains the ability to set up user profiles using Auth0, this can be done via a developer-built profile system or by using pre-existing identification such as a google or facebook login, this became useful in the auditing section.

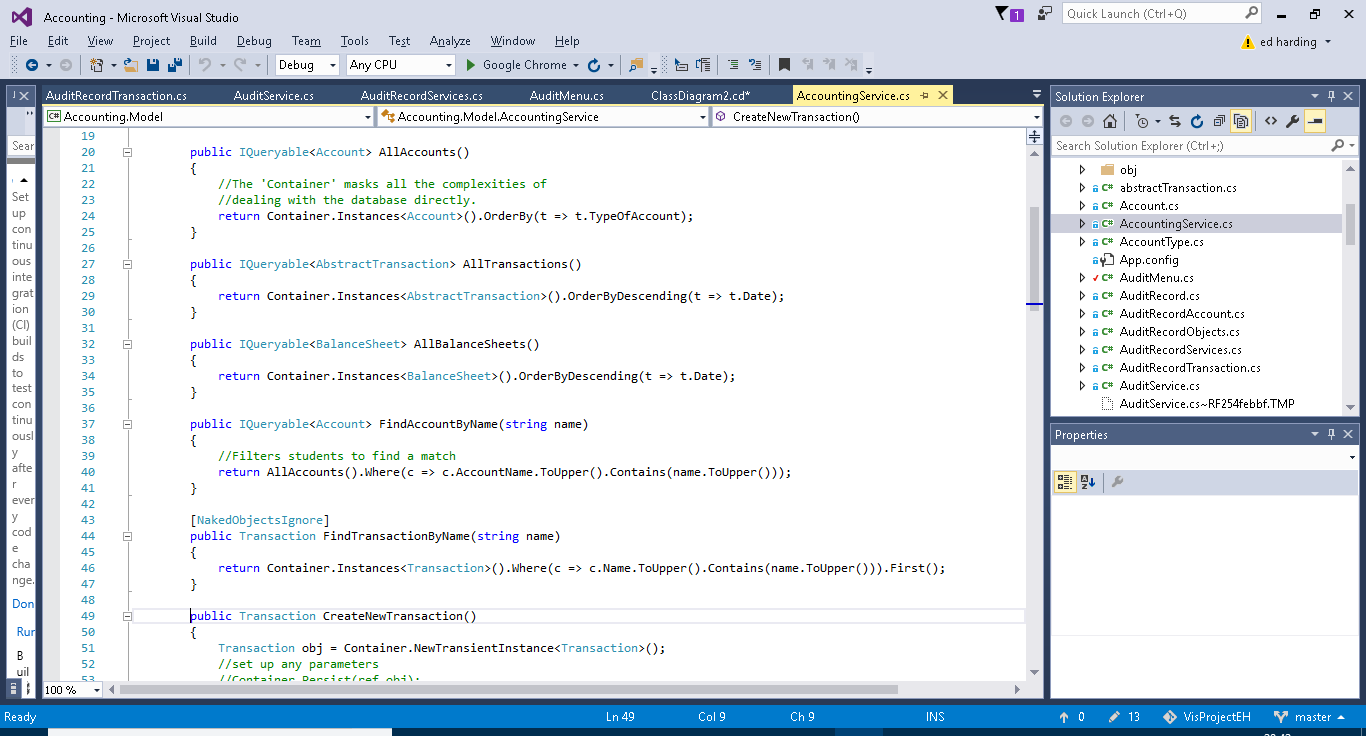
Another advantage I found when researching NOF was that it provides ‘hooks’ which can be used for auditing. A hook will pick up certain user actions and they allow for me to write code to determine what happens when they are picked up. This allowed for me to implement and auditing system. Using the hooks and the NOF Auth0 user profile system I was able to create an Audit service which would use the ‘hooks’ to audit the user by saving information about the user’s profile and the action they performed into an Audit Record object class.

As you can see the advantages of using Naked Objects Framework allowed me to direct my focus towards building a more complex object model and better overall solution as I didn’t have to worry as much about these other ‘tid-bits’.

# The Advantages of LINQ

LINQ is a uniform query syntax which is integrated into C#. It is very similar to SQL in the way that it is used to save and retrieve data from a database. The significance of LINQ is that because it is integrated into C# it can easily interact with databases created from within my project.

There are many advantages to using LINQ over SQL. Structured Query Language (SQL) has been around since 1974 and in this whole time it has never truly been revamped it has just been extended. As you can probably guess this has caused it to become very clunky in comparison to today’s languages. LINQ is a lot more compact compared to SQL, below is an example of a LINQ statement from the Account Services class.



As you can see this is a lot more compact compared to the traditional SELECT FROM WHERE SQL statement which can get very bulky the more complicated the query gets. Another advantage of LINQ is that it is written in a way which is very similar to C# it uses the “.” syntax and C# and user made methods can be incorporated into the query. For general coding it is easier to stick to one form of coding rather than switching between two.

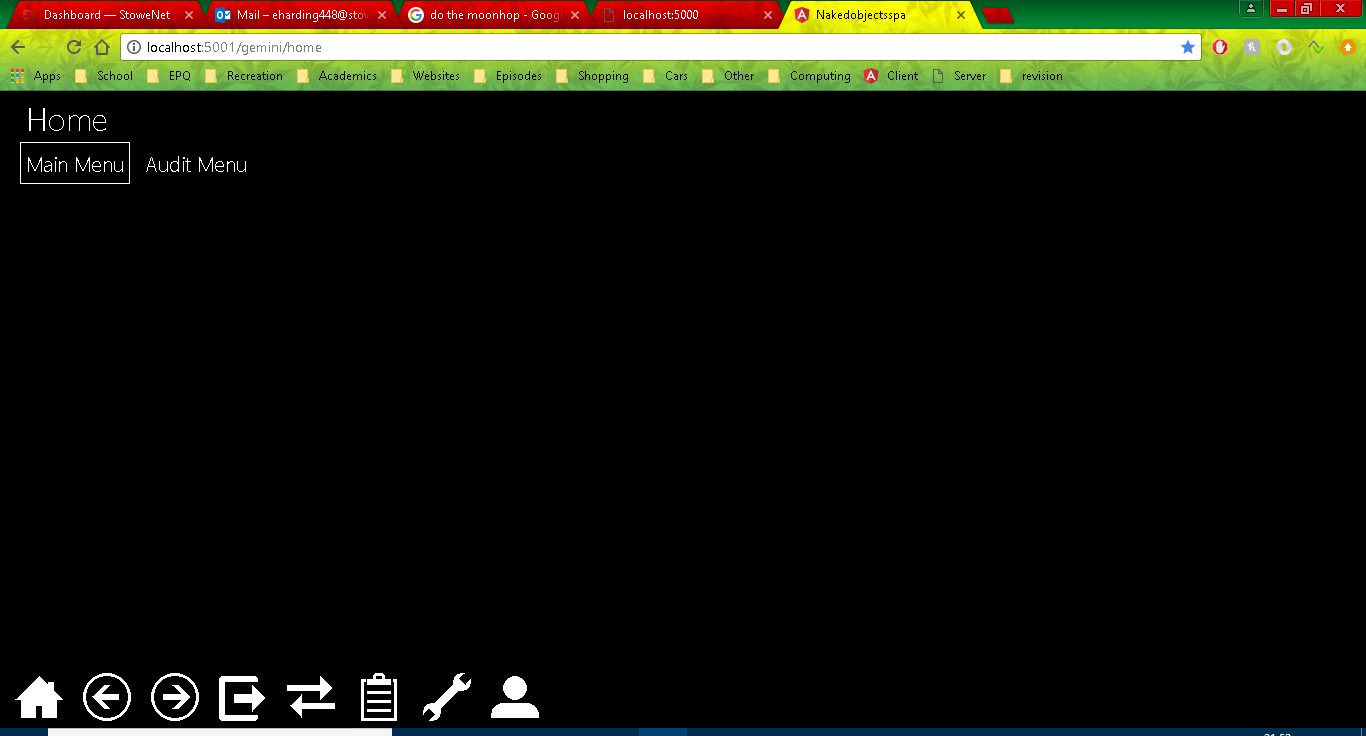
A final point I would like to make about LINQ is that it is technically a form of functional programming. Functional programming is a relatively new programming paradigm which does not allow for any sequential command execution and instead uses functions made up of one single expression. LINQ achieves via the use of a lambda expression. This can be seen in the previous example of code; the lambda expression is the “c => c…….” part.

# Solution Layout

The Solution consists of four separate projects. The Client, Model, Server and Database projects. This format is a way of using the Naked Objects Framework, it works using the client-server model by this I mean that everything the user does via the UI (provided by the client project) goes through the server via requests and responses. I started my project by downloading the NOF Template Solution which comes with the four projects included. The NOF Template solution can be downloaded by anyone and is referenced in the NOF user manual. Below is a breakdown of the projects into more detail.

## Client

The client project provides the User Interface for the solution (pictured below).



The client uses the Objects and repositories created in the Model Project to create the interface. The actual interface itself can be customized by the programmer in additional ways, I personally decided only to change some of the colours of the objects (shown below Needs to be done)

[]

The UI also supports simple usability functionality like drag and drop (shown below can be added once colour change is done)

[]

## Model

The Model project is where the majority o

For objects / classes and services. (where objects from the design section are coded, show AN example). Auditing is also in there. Model is where I wrote most of my code.

Objects and services/repositories.

Show Programming techniques (here’s my code, here’s what it’s doing in the UI) Show examples of actions. (show collections etc. lots of examples); Show validation (robustness)(add validation); User-friendliness (auto Complete); Attributes(using [] hiding objects changing layout etc. etc.) LINQ & repositories.

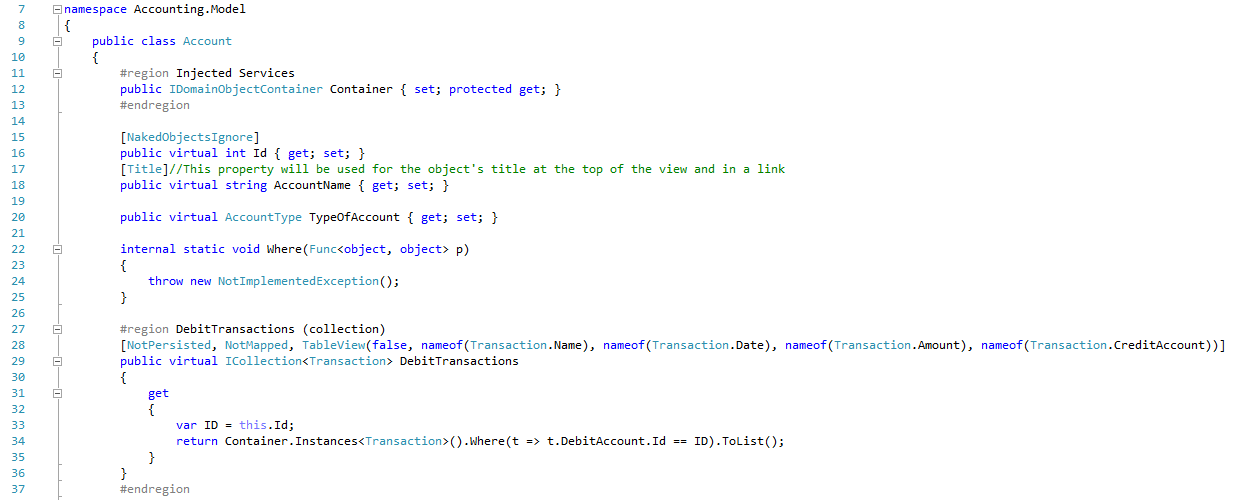
### Functionality

#### Objects

In this section I will analyse some example classes from my code to demonstrate what is going on under the covers. 3451

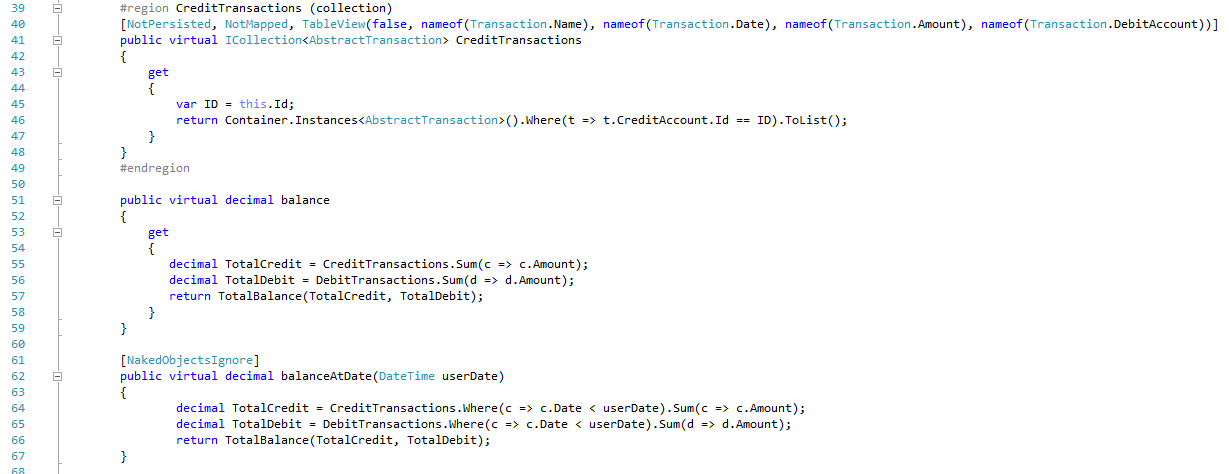
Object Example – Account

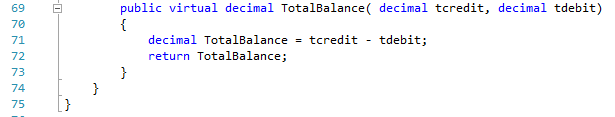
Below is an example of one of my objects, I have chosen to use my Account model as it is one of the most used objects in my project.



Line 11 is a region which contains any injected services used in this class, a lot of my classes have these regions. The Container service (a NOF service) is used in account and therefore must be injected.

Above (line 27) and below (line 39) are two collections used to hold the list of debit and credit transactions applied to the specific account. As you can see the get clause uses a LINQ statement to get the data for the collection. Further down is a screenshot of the UI showing this.

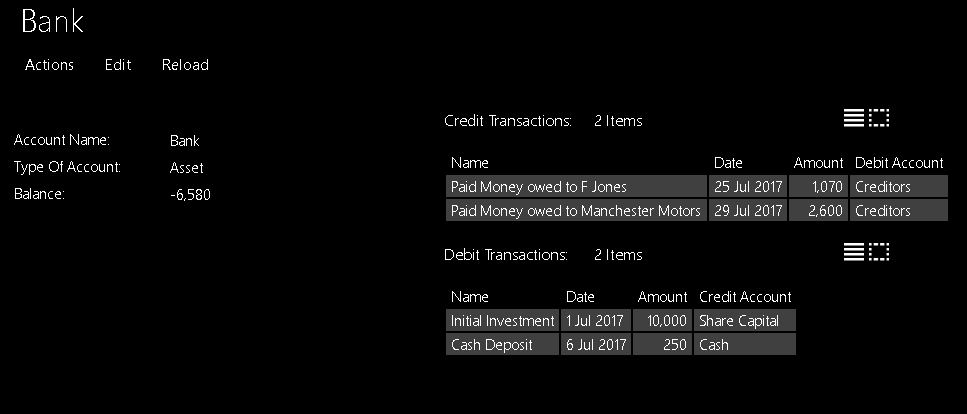




Above (line 51) is a property which gets the balance from the collections of transactions mentioned earlier.

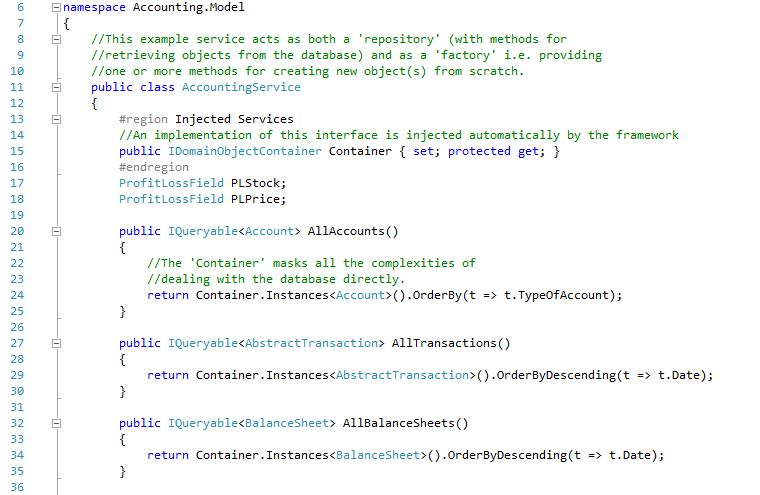
Lines 62 and 69 show methods which are used on the balance property.

Below is a screenshot of an account, on the left you can see the collections of transactions, and on the right you can see the accounts name, type and balance.

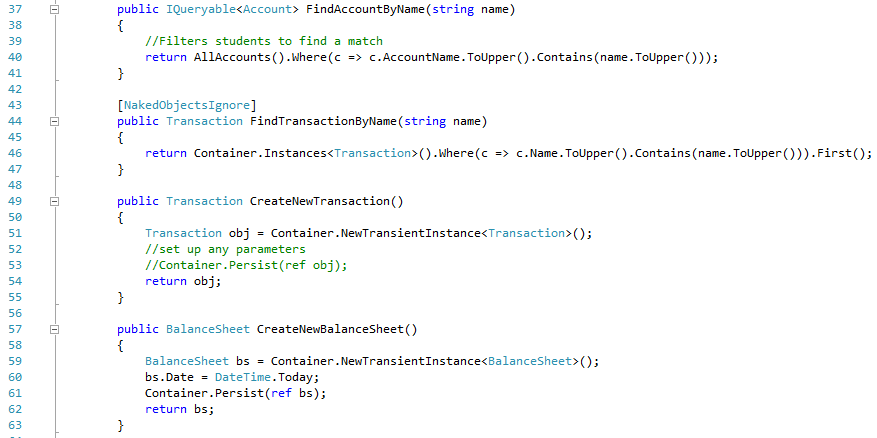


Repository Example – Accounting Service

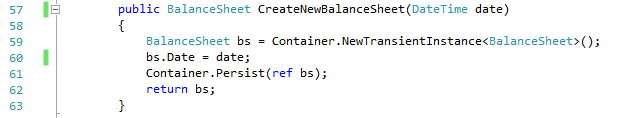
Most methods I talk about will be demonstrated one way or another in the testing section.

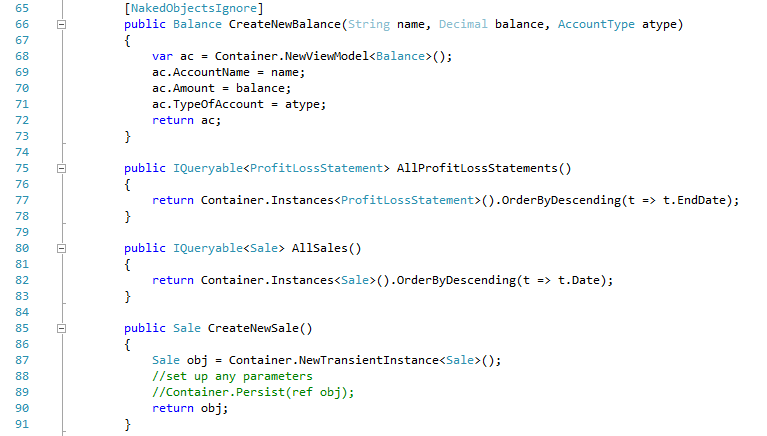


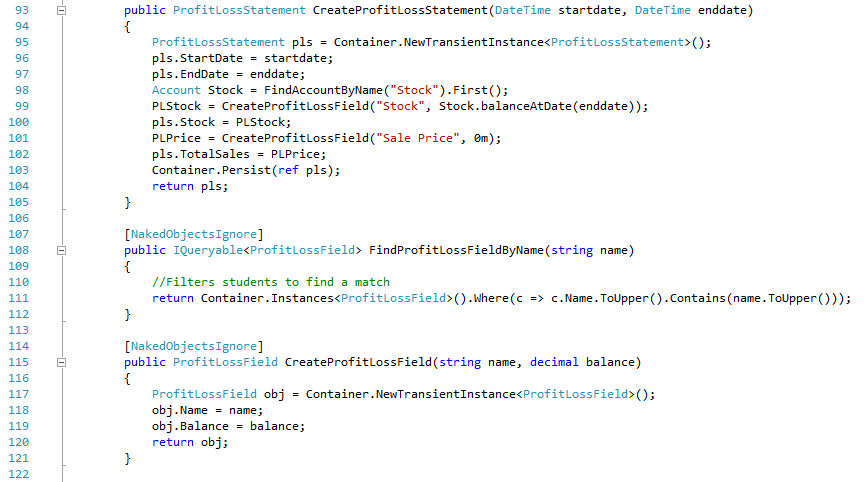
Line 20, 27, 32, 75, 80: All of these lines contain different methods which all follow the same pattern. These methods use LINQ statements to get all of the instances of a specific object, e.g. Account, these are then returned to the user as a list. The list is ordered in different ways depending on the object, as you can see The AllTransactions Method lists them by date in descending order.



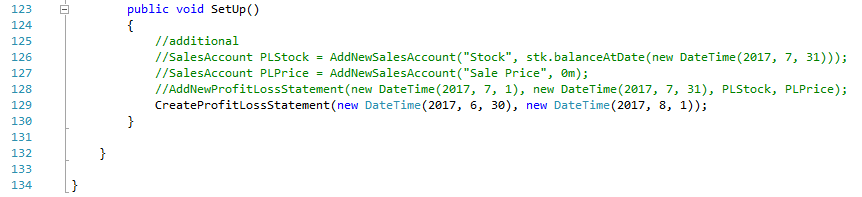
Line 43 is an attribute for the method on line 44, the Naked Objects ignore attribute means the method below it is not displayed to the user in the menu, it can however be used in other objects. The method on line 44 is another query linq statement.







Lines 66, 85, 93 and 115 are all methods used for creating other objects, again they all follow a similar pattern. They use the container service (injected at the start of the code) to create a new instance of the desired object. Each method has different parameters depending on the different properties which need to be defined by the user. The user entered properties are then assigned to the corresponding properties on the newly created instance of the object.



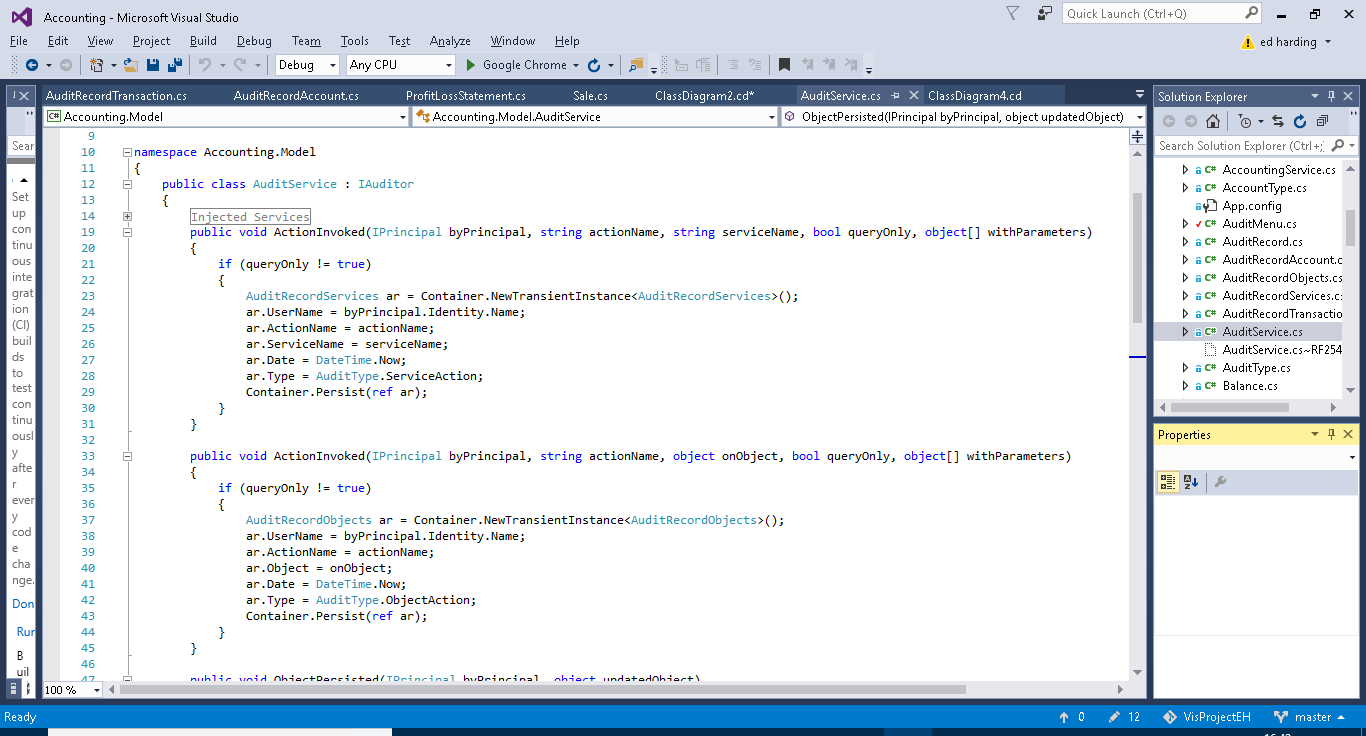
Line 123 is a SetUp method used for creating an example from the click of a button, it can have nothing or lots inside it, before release of the program it should be hidden, as it should only be used by the program developer.

#### Auditing

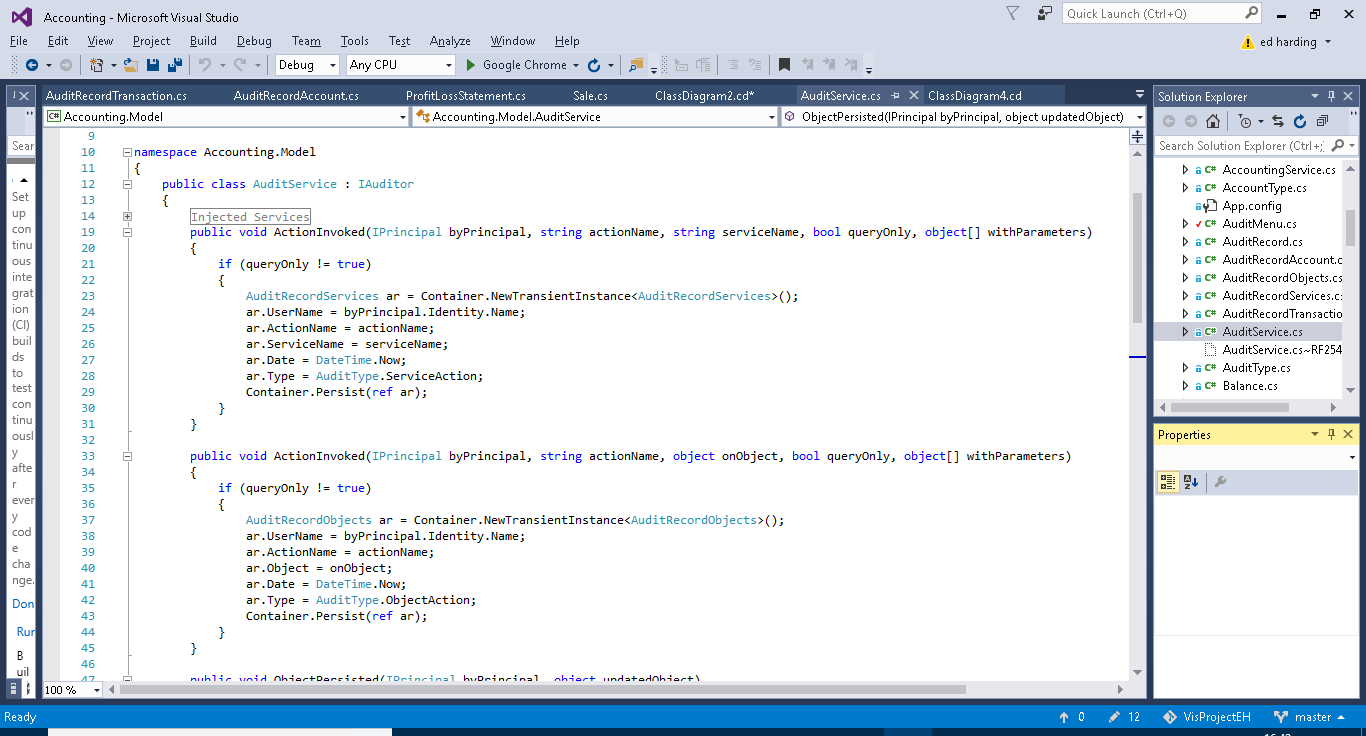
Auditing is one of the more complex capabilities of my program. My auditing code can be split into 3 sections (I will go into more detail of each below). The AuditService which detects when actions have been perfomed by a user, the AuditRecord which is used to save information about the action and user and finally the AuditMenu which is the UI end of the audit capability, it is used to display the audit records to a user.

##### Audit Service

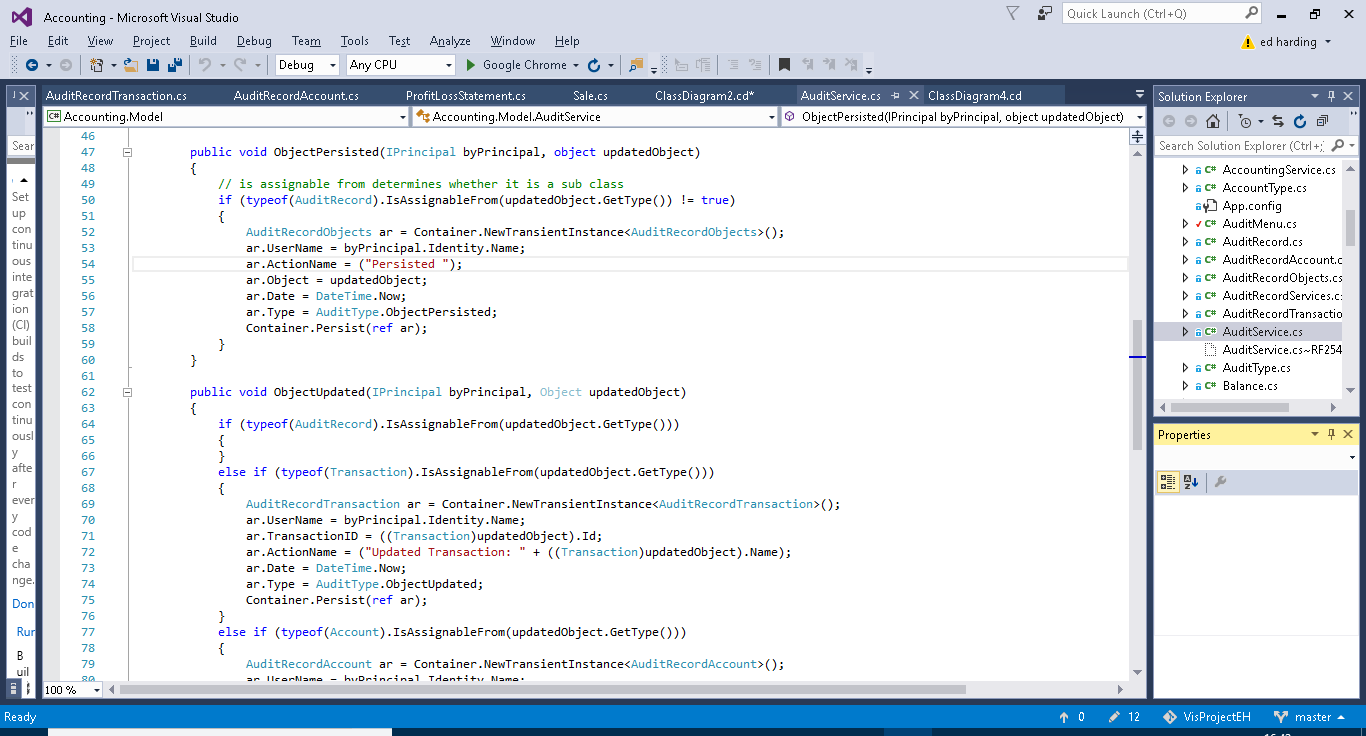
The Audit service constists of 4 methods each will be called under different circumstances.



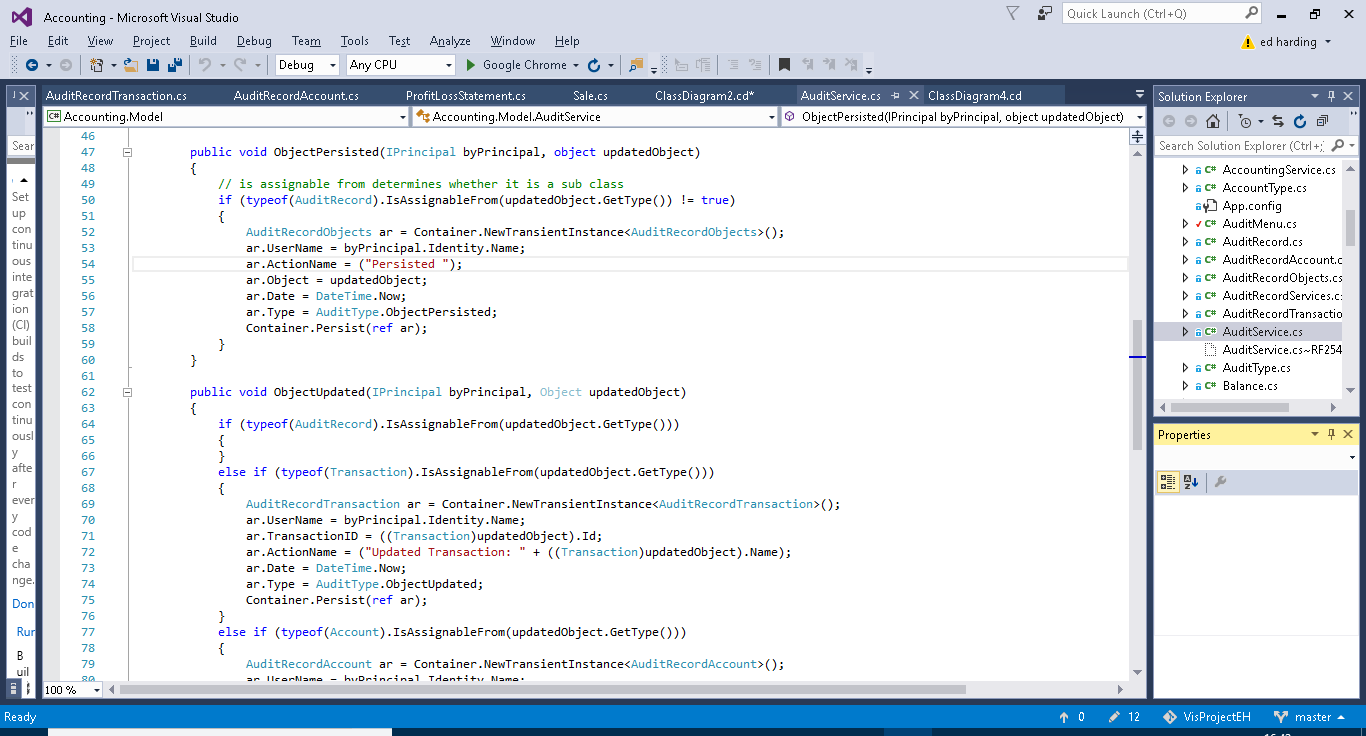
The first method (above) is called when a service action (e.g.) is executed by the user. The if statement checks to see if the action was a query only action, if it is nothing will happen as queries don’t necessarily need to be recorded as nothing critical is happening to the data. If the action isn’t a query action the method will create a new instance of AuditRecordService (AuditRecords are explained further on in this section). It then fills out the fields from AuditRecordService and persists the new instance. The username is obtained from the byPrincipal parameter. The username comes from the Auth0 log in system. AuditType is an enum I created for querying purposes you will be able to see this in use later on.

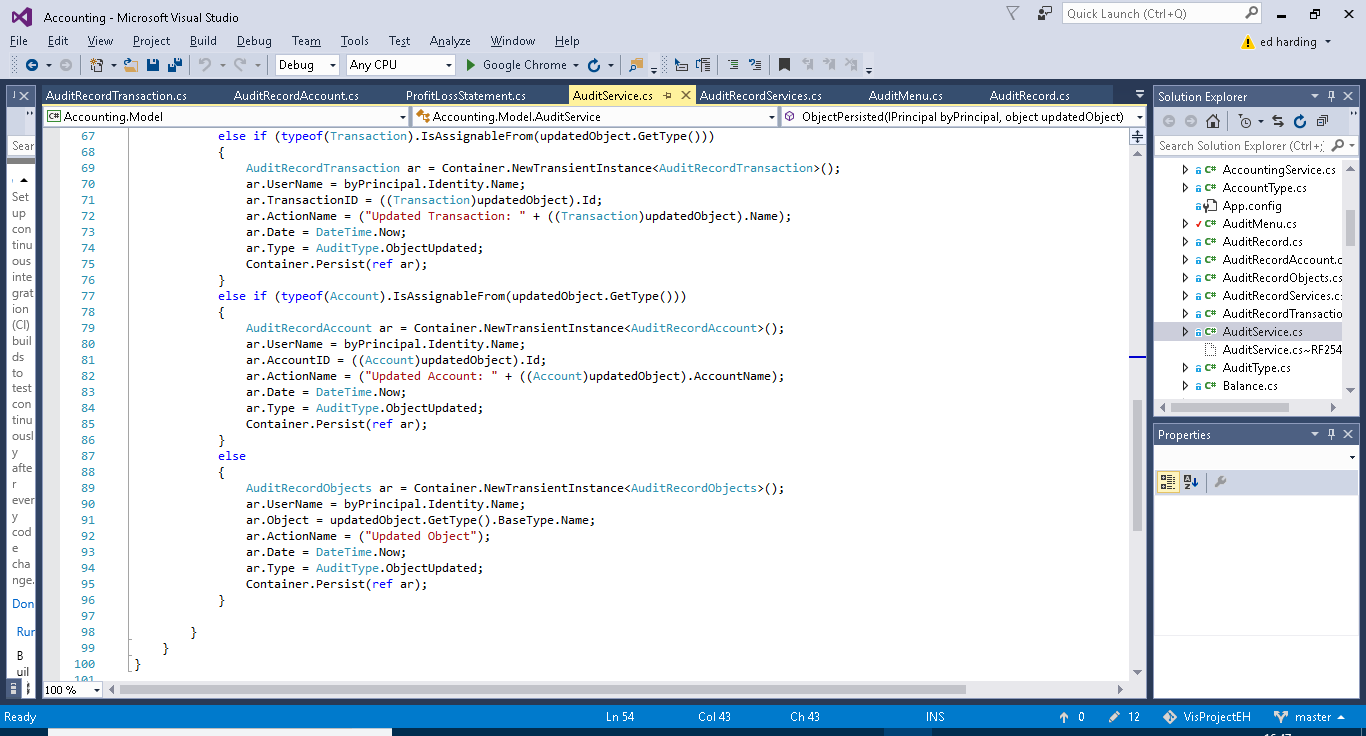


The second method is very similar to the first however it acts upon actions performed on objects (e.g.). Like the first method it checks whether the action is query only or not, it then creates a new instance of AuditRecordObjects, fills out the fields and persists the new instance.



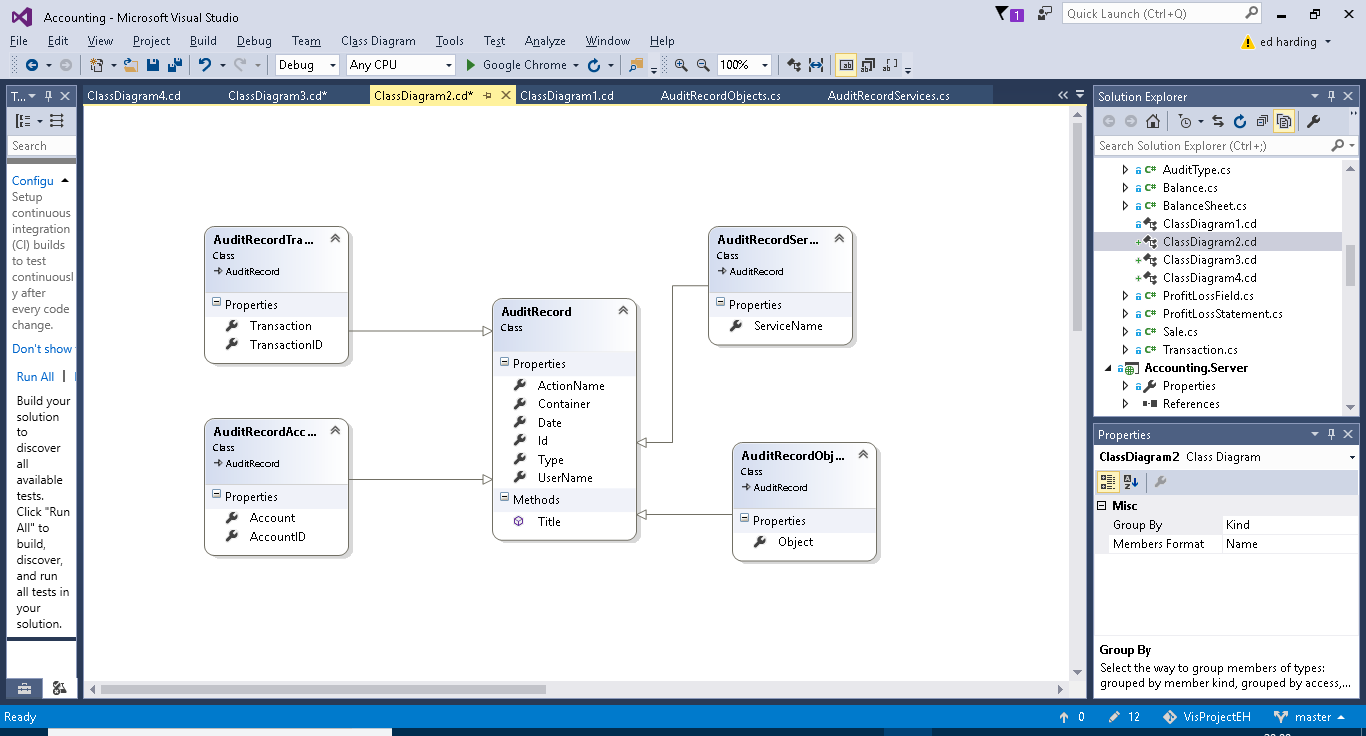
The third method is for auditing persisted objects. The first if statement is very important, its purpose is to assert whether or not the object which has been persisted is a sub class of AuditRecord (e.g. AuditRecordService), if it is then the method does nothing. This is important as without this if statement this method will be called whenever any AuditRecord sub class is peristed such as in the previous methods. As this method itself persists a subclass of AuditRecord it then will get stuck into a loop and a stack overflow error would occur, this is in fact what happened the first time I tested my audit methods, as I did not have the if statements to catch the problem I ended up with my program halting everytime I did anything to an object. If the persisted object is not a sub class of Audit Record then the method will create, fill out and persist a new instance of AuditRecordObjects. The action name is not passed in through a parameter as in previous methods and is therefore declared as “persisted” by the method.



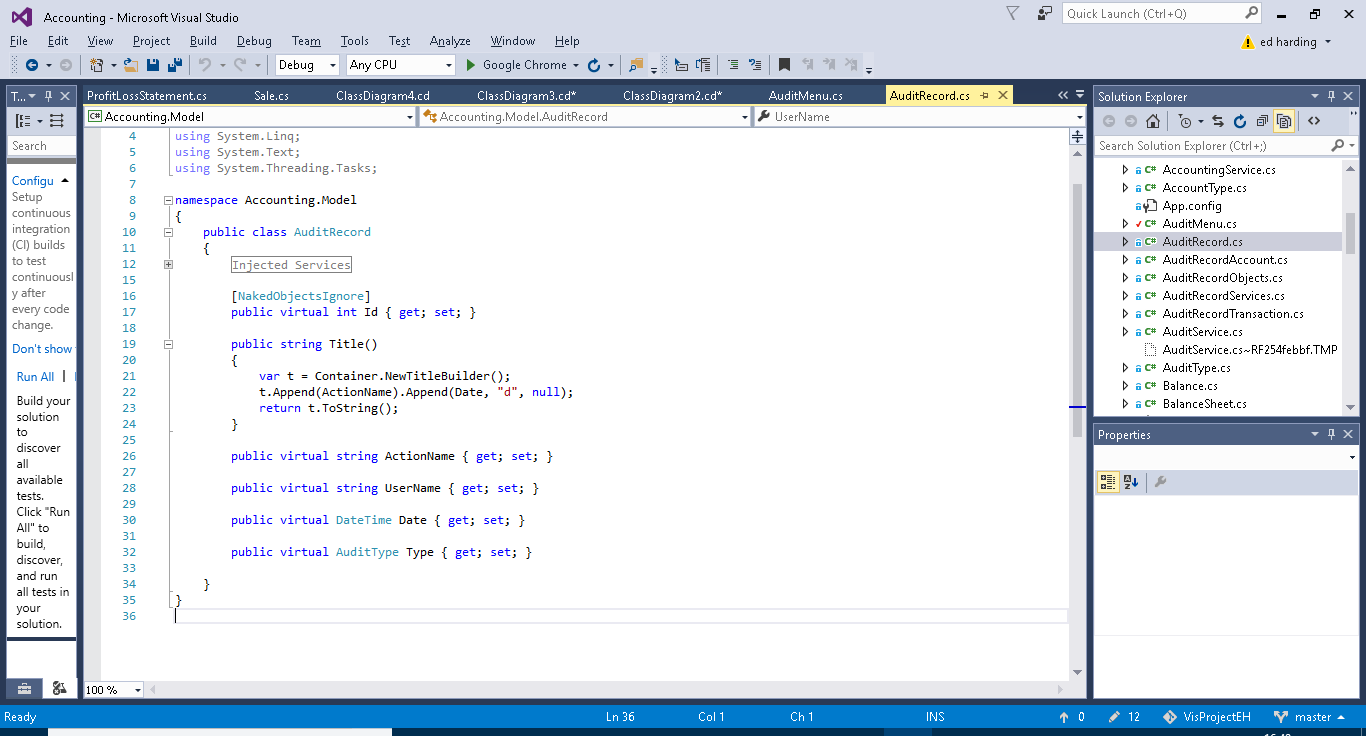


Similiarly to the previous method the first if case checks that the updated object isn’t a sub class of Audit Record, again this is to aoid stack overflow errors as described before. The next three if cases are to establish if the updated object is a transaction, account or other object. This is because transaction and account have their own subclasses of AuditRecord. As before a new instance of the required AuditRecord subclass will be created the fields initialised and then the instance will be persisted. The advantage of having Account and transaction records separate from one another and other objects is so that the name of the account or transaction can be added to the action name for userbility purposes and for querying advantages.

##### Audit Record

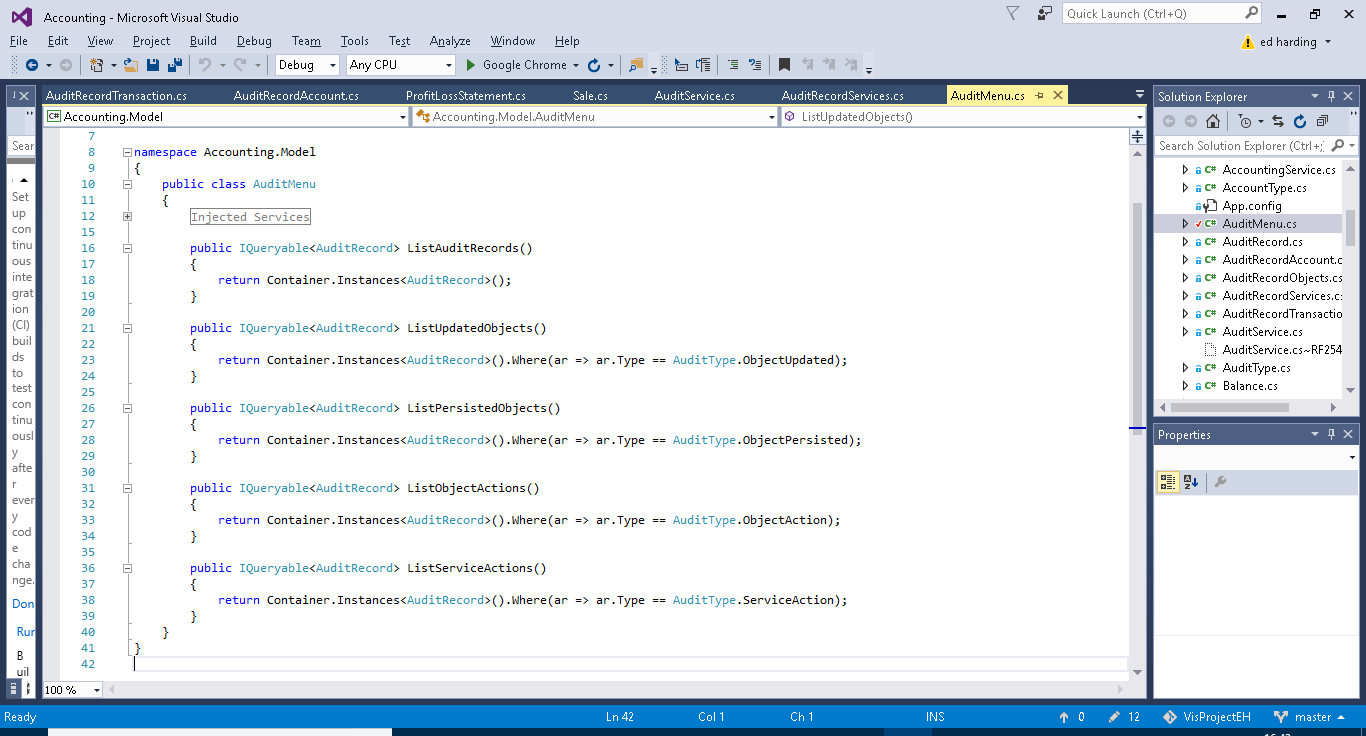


Above is a class diagram showing the four subclasses of account record. These were used in the methods in Audit Record (previous section).



This is the super class Audit Record. It contains the Username ActionName and Date fields, very typical and useful information for someone auditing the system. The AuditType field is for querying purposes. And the Title Builder is for usability purposes i.e. making the name of each instance of AuditRecord more presentable and convenient for the user to find what they need as it combines the date and the action name. The subclasses each have one or two additional fields specific to themselves.

##### Audit Menu



The audit menus sole purpose is to provide pre-built query methods on the entirety of saved AuditRecord. These methods just make it easier for the user to find what they want. The first method just outputs an entire list of all AuditRecord (this being all the instances of the subclasses of AuditRecord). The second method Lists all the instances of AuditRecord which contain information on Updated Objects. The third method Lists all the instances of AuditRecord which contain information on Persisted Objects. The fourth method lists all the instances of AuditRecord which contain information on executed object actions. The fourth method lists all the instances of AuditRecord which contain information on executed service actions.

#### Use of an IViewModel

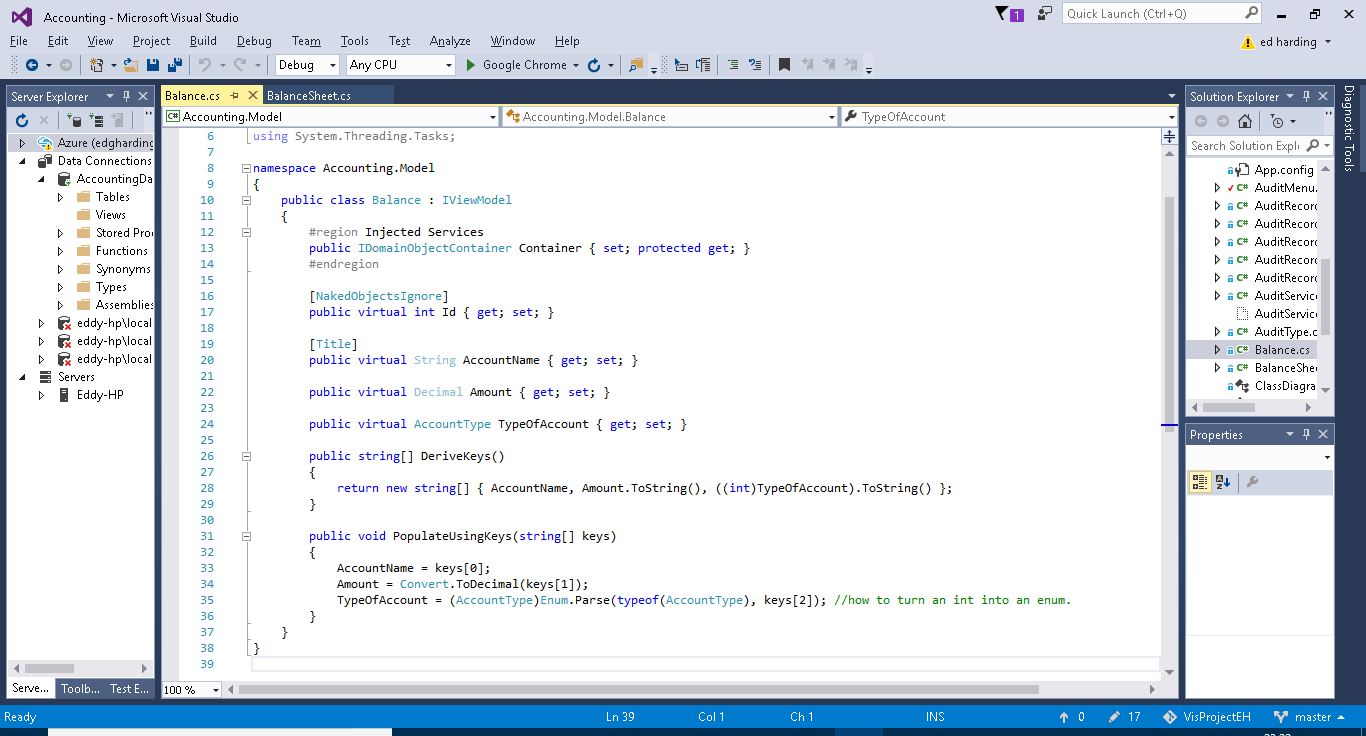
For creating the balance sheet, I needed to get the balance for each account as of a specified date. The balance of an account depends on the different credit and debit transactions applied to it. I could simply have accessed the accounts balance directly from the balance sheet class, however I saw 2 main advantages to using a view model class instead.

First, I will explain what a view model is and then I will move on to the 2 reasons I decided to use one. A view model is a class which appears to the user through the GUI as if it was a persisted object but is not.

1. The fact that a view model is not persisted means that it must instead get the information from somewhere else, in my case this is from the Accounts class, the advantage of this is that the balance sheet is created from all of the previous transactions up to a specific date, therefore the balances need to be based on the balance of the accounts at the specified date. However, it is of course possible to add transactions (which effect an accounts balance) with past dates. Therefore if a balance sheet is created and then a transaction is added which has a date before the date of the balance sheet, this would cause a problem if the balances are persisted objects as the balance sheet will not update and will therefore be inaccurate. Therefore, using a view model causes the balances to be created live every time the balance sheet is opened (how this works will be explained later on).
2. The other advantage of using a view model is that it allows for an increased level of customisation of how the balance sheet is displayed to the user.

These advantages are demonstrated in the code break down which follows.

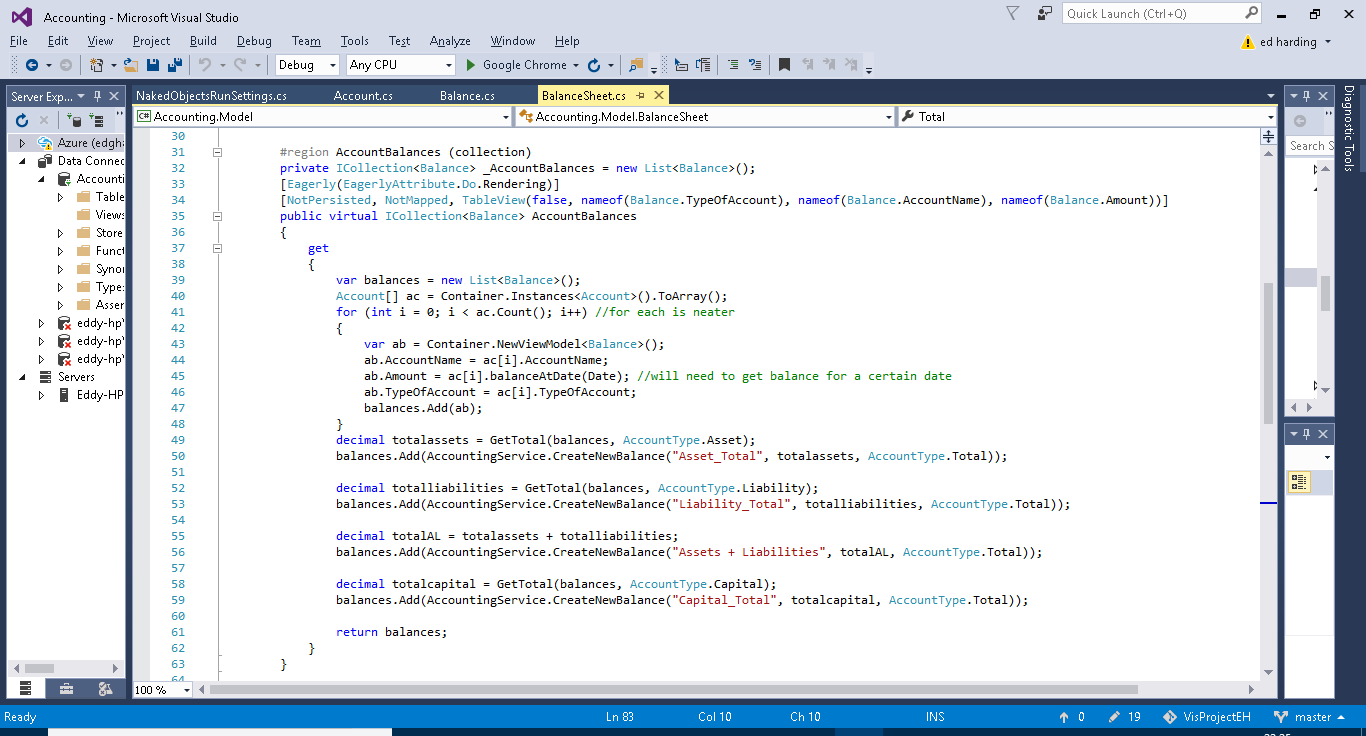
Below is the code for the Balance View model, as you can see it ‘copies’ some of the data held in the accounts class. It achieves this by using two method DeriveKeys and PopulateKeys, I will go into more detail of what they do below.



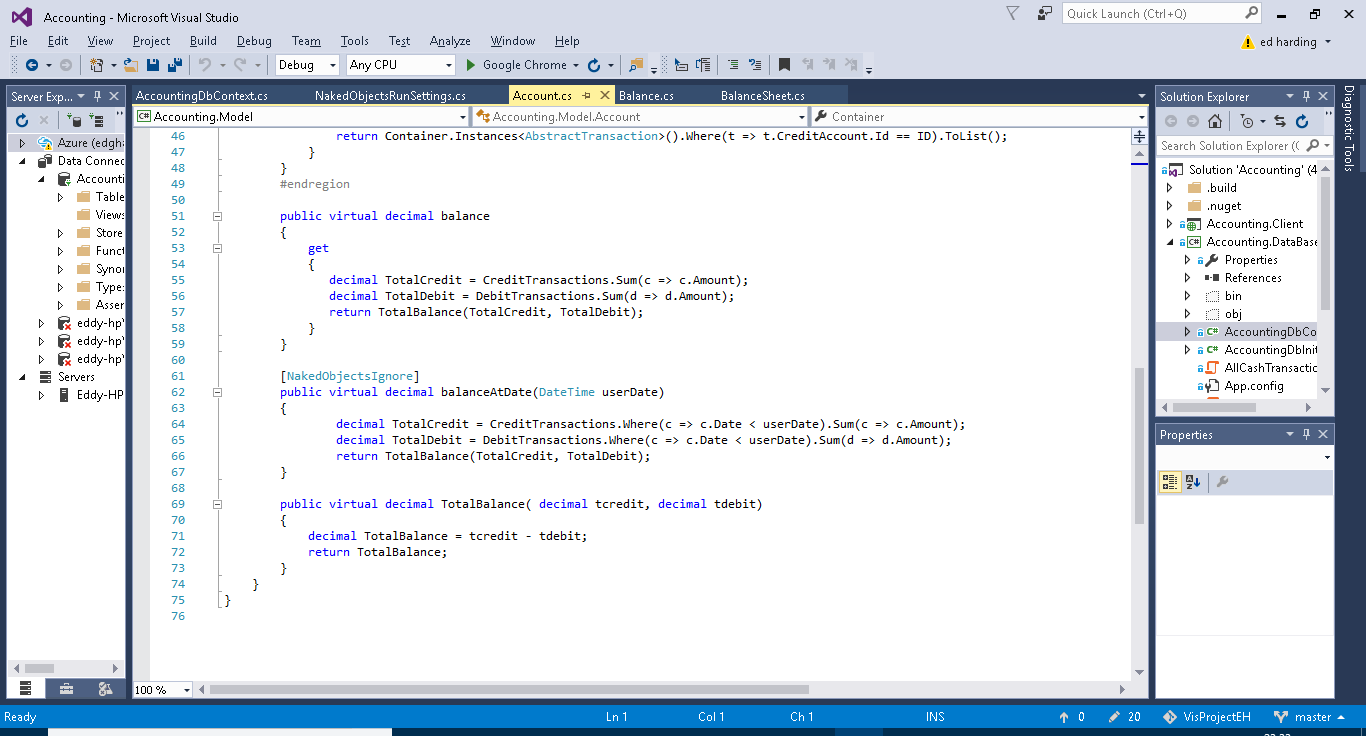
DeriveKeys: This method returns a string array containing the keys derived from the object which the view model represents, so in this case it contains three of the properties from the Account class; AccountName, Amount and TypeOfAccount.

Populate Keys: This method populates the properties of the View Model with the keys which are retrieved by the DeriveKeys method. As you can see the Balance AccountName is being set to keys[0] which in the string of keys returned by DeriveKeys is the AccountName gotten from the Account Object. This is how the View Model is able to be created live every time it is called upon.

The below screenshots are of the balance sheet class and show the use of the balances view model inside the balance sheet class.

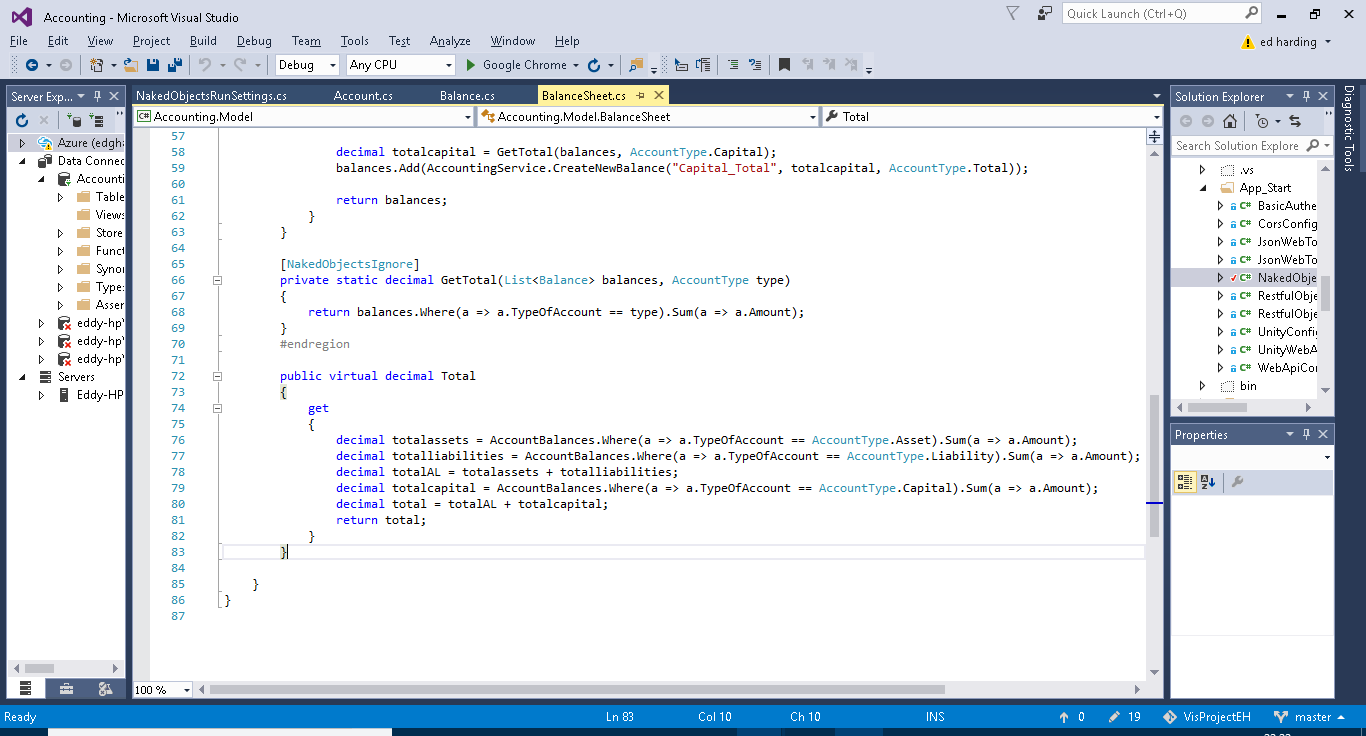


Here you can see a collection of balnces being created, I chose to use a collection as it would be able to create a nice ordered table of the accounts (as the balance class represents the accounts class), this provides and easily readable and follows the typical format of a balance sheet. As you can see above (line 39 to 48) a new list of balances is being created and the balance of each individual account is being assigned to a new instance of balance every time until this has been done for all of the accounts. The balnceAtDate method comes from Accounts and allows for the balance of the account to be calculated at a specific date, the method is shown below, this like a lot of my methods is done using LINQ queries to find the data required.(back to the above screenshot) I am then creating 4 custom balances and adding them to the list of balnces. These custom balances are each totals which are used for calculating the overall balance of the balance sheet (Assets +liabilites – Capital), this is done using the GetTotal method which is showna dne xlained below. Again thanks to the View model and that overall the whole thing is a collection, these will be included in the table of accounts (balances view model).



Below (lines 65 to 69) is a method for getting the total balances of all of the accounts of a certain type, it is pretty straight forward and uses a LINQ query to do this.

Below (lines 72 to 83) is the final part of the balance sheet the total. If a double entry bookkeeping system has been programmed and used correctly the total of a balance sheet (which uses the equation total assets + total liabalities + total capital) should always = 0. Below is simply the method I created which calculates this from the totals balances which I mentioned earlier.



## Server

As previously stated the solution follows the client-server relationship. Eventually the idea behind this architecture is that the server project would be deployed via a webserver such as azure so that it could be accessed anywhere. The project uses REST which helps communication between the server and client. It does this via a RESTful API, this allows for the client (the UI) to talk to the server (the object modelled database) to request and receive data. It does this using the http commands GET, PUT, POST and DELETE. REST also means that the database can be traversed through from one start point down to the lowest level, I have provided some screenshots below to show the visual representation of the RESTful Server.

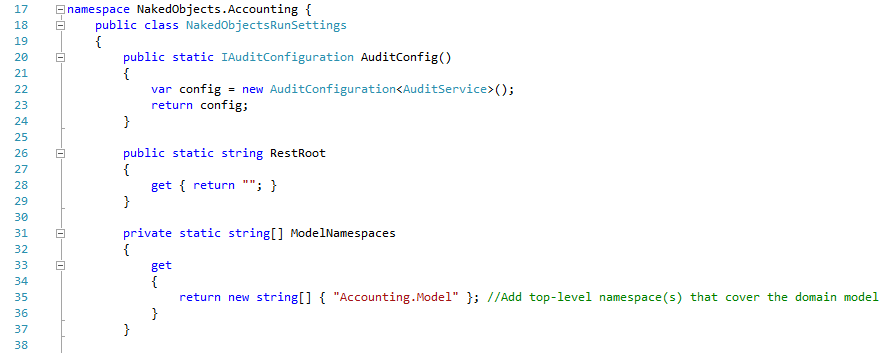


Above is the top layer and below is an account specifically the share capital account, as you can see I was able to navigate my way there fluently in the same way which the client server model does. (the below screenshot only shows part of the full account).

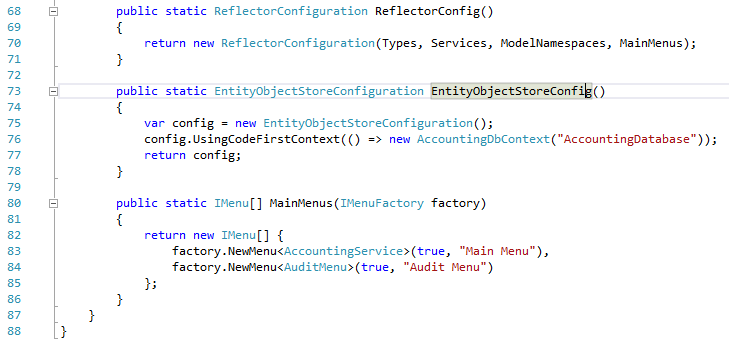


I will point out that during all my testing and use of this project the Server was completely run locally on my machine. There was not much in this project which I actually needed to edit other than configuring the naked object run settings.

Needs to be explained





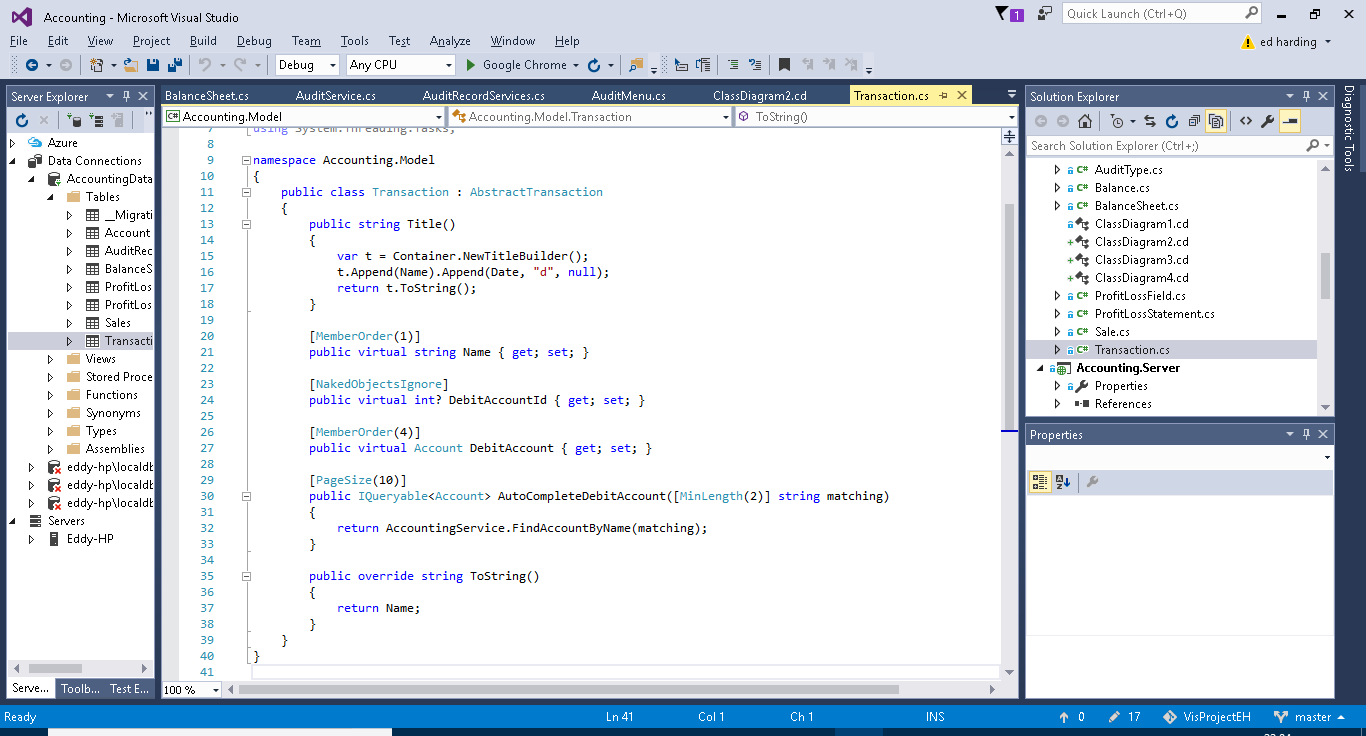


## Database

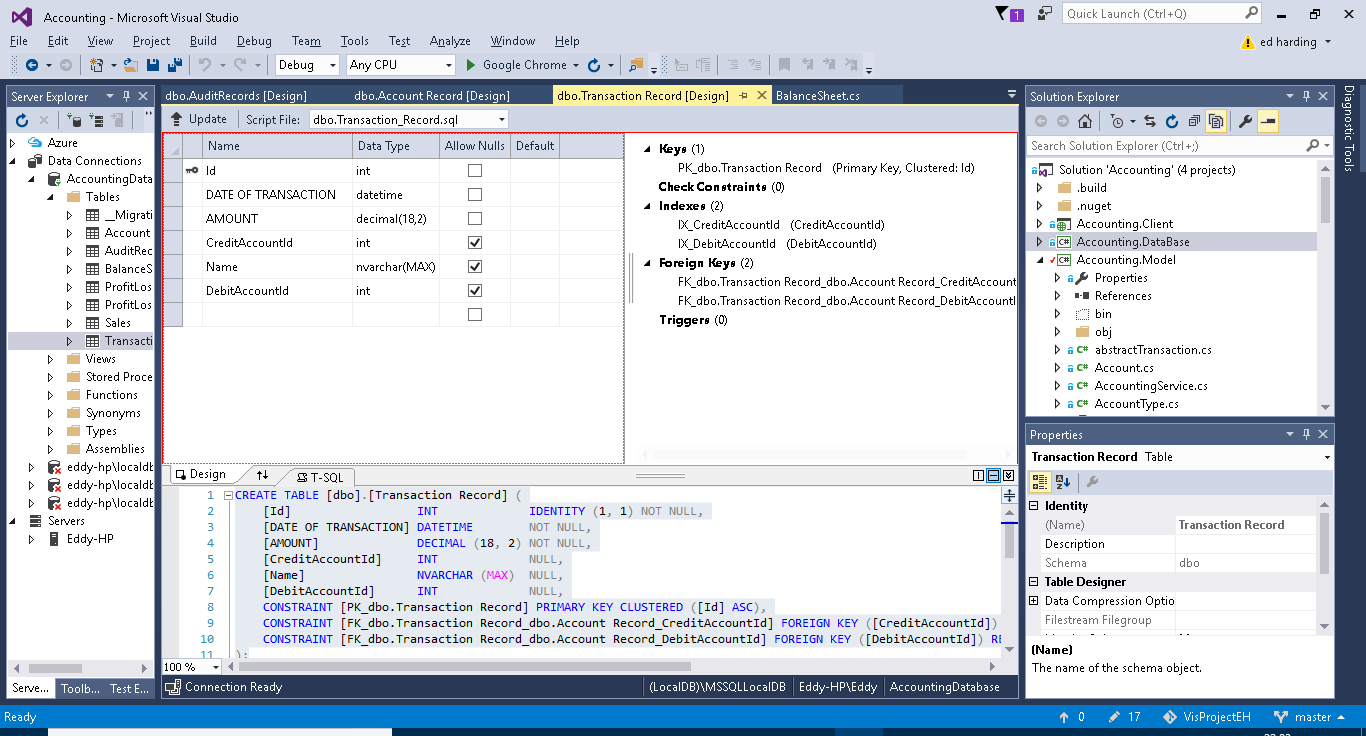
### Entity Framework and SQL Servers

Entity Framework is an integral part of the project as it acts almost like a translator between the business-based objects I have coded and the SQL server database. Entity framework has the ability to format and map user made objects to a database by creating an Entity Data Model from the objects it does this by using a technique called object relational mapping. ORM is a method for converting data of incompatible types into a virtual object database. Entity Framework is a Microsoft product therefore combined with Microsoft Visual Studio there should be little bugs in this process. This therefore reduces a whole work load for the developer. It also allows the developer to perform LINQ queries in C# on the database. This is extremely useful when creating searching methods and such.

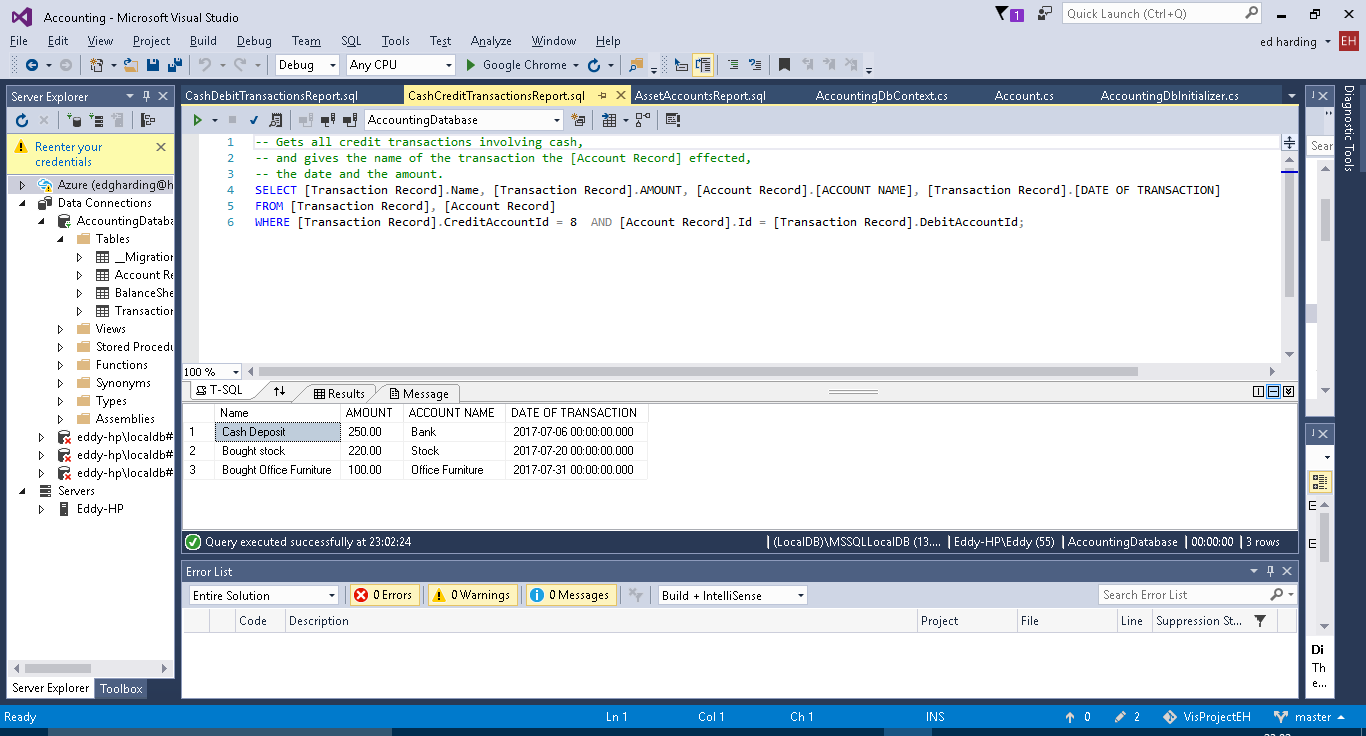
SQL Server is a relational database management system, which as mentioned earlier is used in tandem with Entity framework to model and query a database built from an object orientated project.



Above is the Transaction Object from my solution and below is the same object but it a tabulated form viewed via the database explorer.



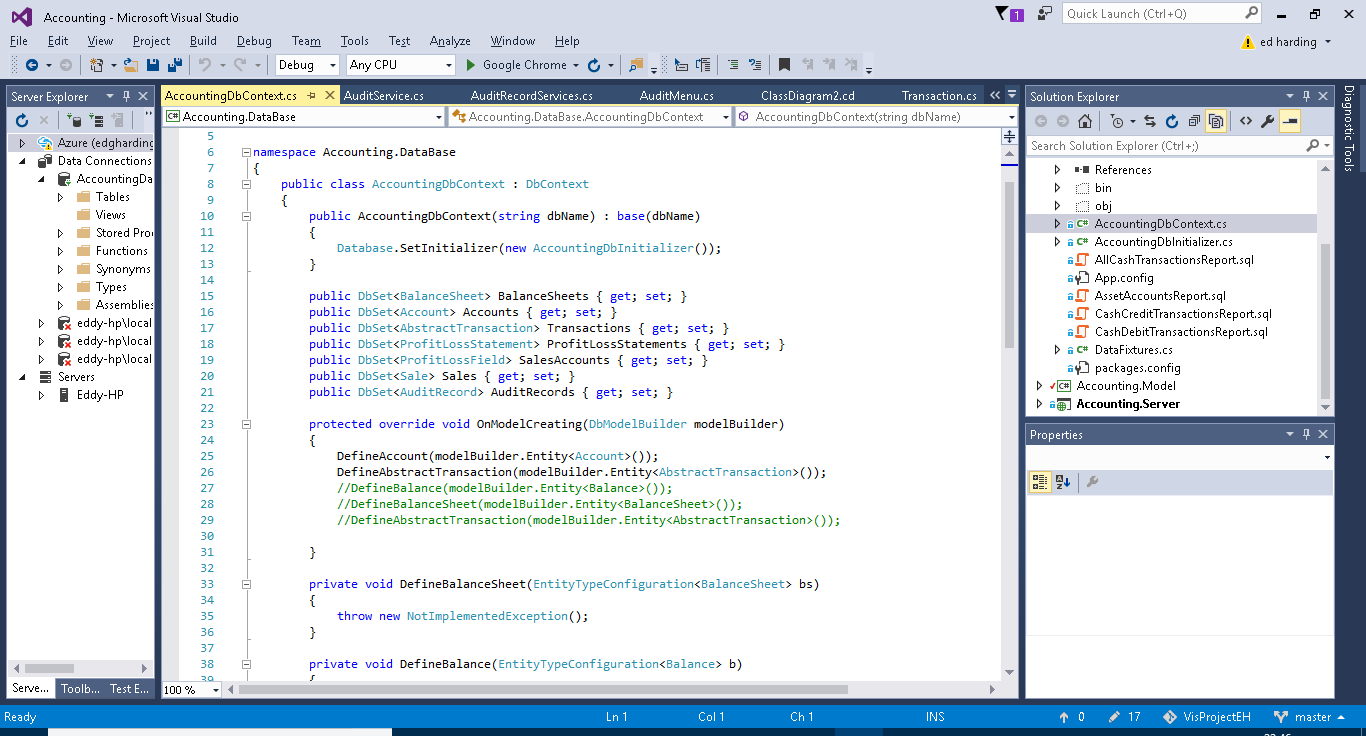
Below is a SQL query I wrote (what it does is explained in green in the screenshot). Below the SQL is the return message.



### DB context

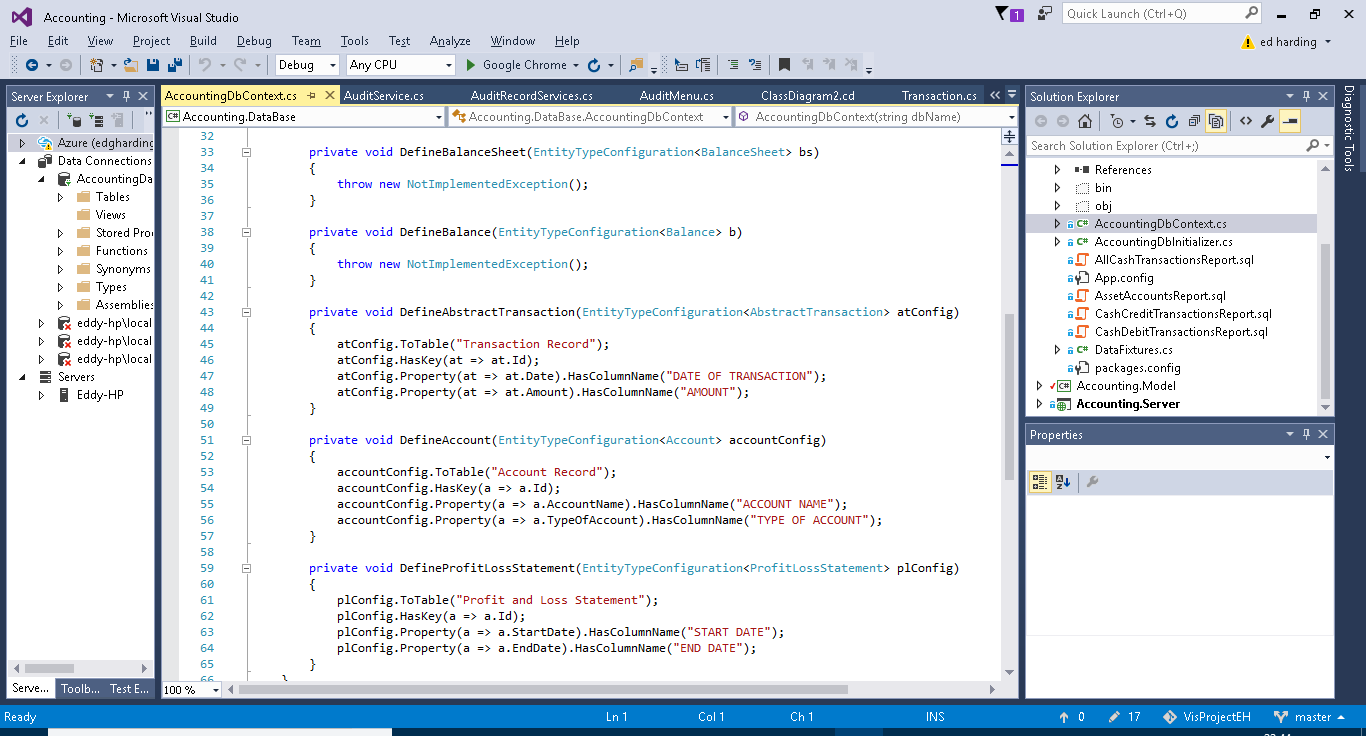
The DB Context class is used for specifying which classes need to be persisted and it is used for adding methods which allow for the customisation of the tables created by entity framework.

On lines 10 to 13 below there is a method which just tells the server which Db initializer (explained further on) is being used, this is because you could theoreitcally write lots of different ones and then call the one you want by changing these lines of code.



Above from line 15 to 21 are a list of the different classes which need to be persisted inside the database, this basically just points out the classes to entity framework so it knows which ones to model. It also allows for instances of these classes to be created from inside the DB initializer.

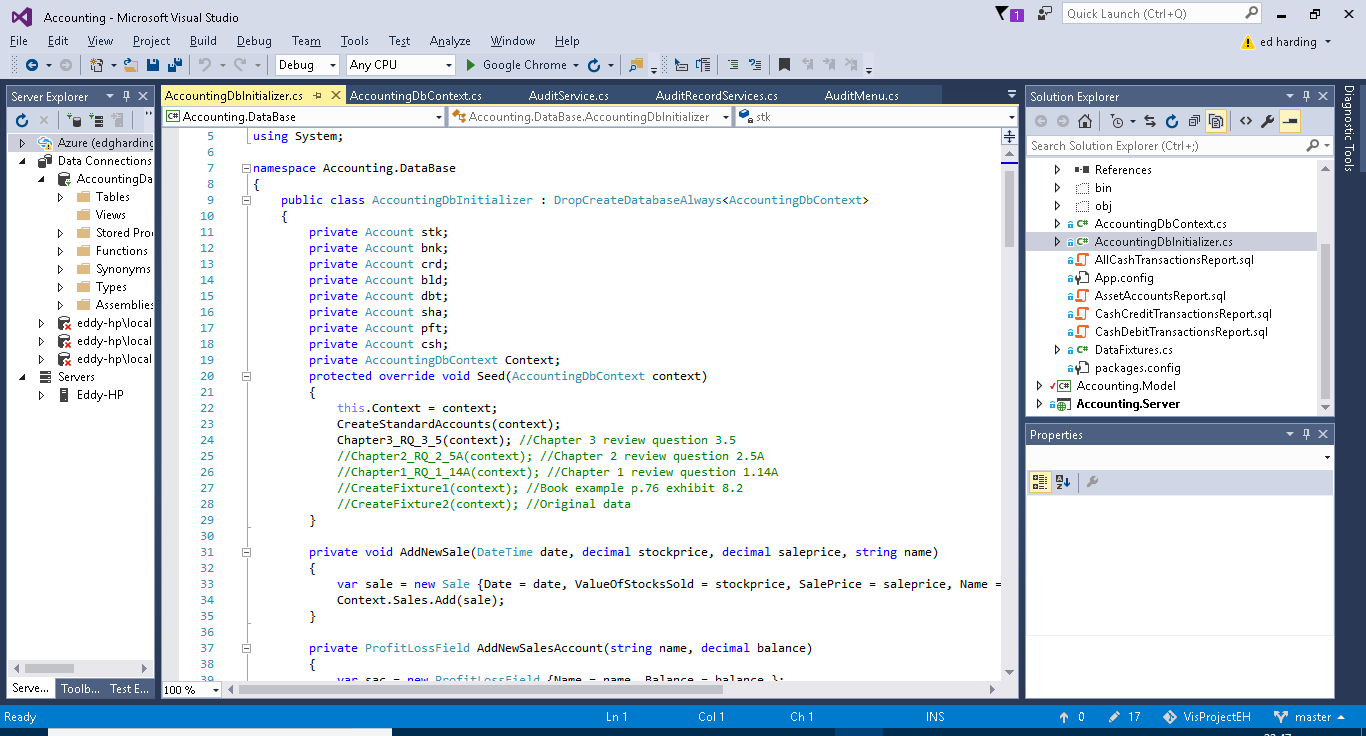
Above from line 23 to 31 is the method which overides the entity framework tables with the methods listed inside it, these methods are shown and explained below. The commented out lines are methods show below on lines 33 to 41,as you can see I did not impliment anything here and therefore have commented them out in the overide method.



Lines 43 to 65 contains 3 methods, each do the same thing just for different objects. As mentioned previously these methods are for customization the tables which entity framework creates. These methods simply help to format the tables created, as you can see the key for the tables is being set to the ID of the class, and then the statements following that are just giving the columns more appropriate names. I mainly did this for the use of the SQL statements, it means that when a table is returned from a SQL report it will have more user-friendly column names etc.

### DB Initializer

The DB Initializers purpose is for seeding data for the project, this can be fore testing or providing a default start up for the user. Below are some screenshots of the DB initializer which I will explain the aspects of.



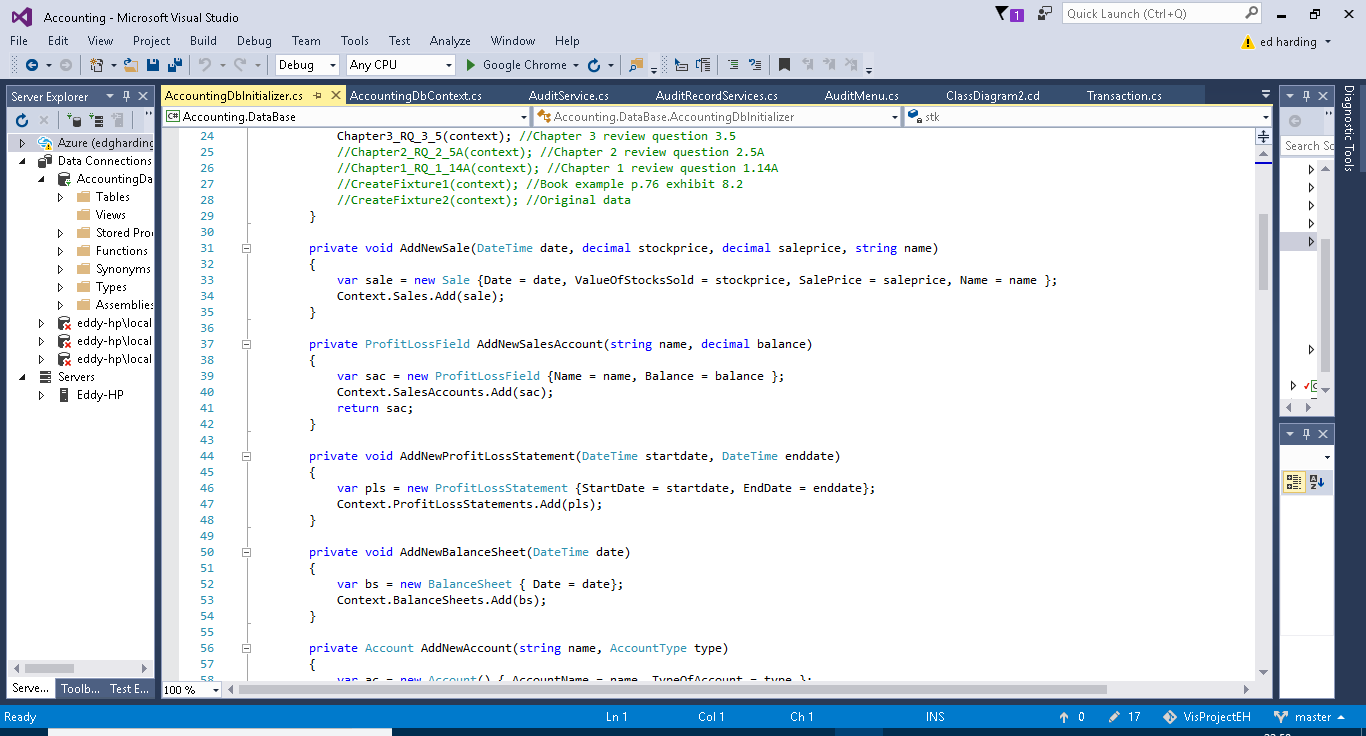
Above:

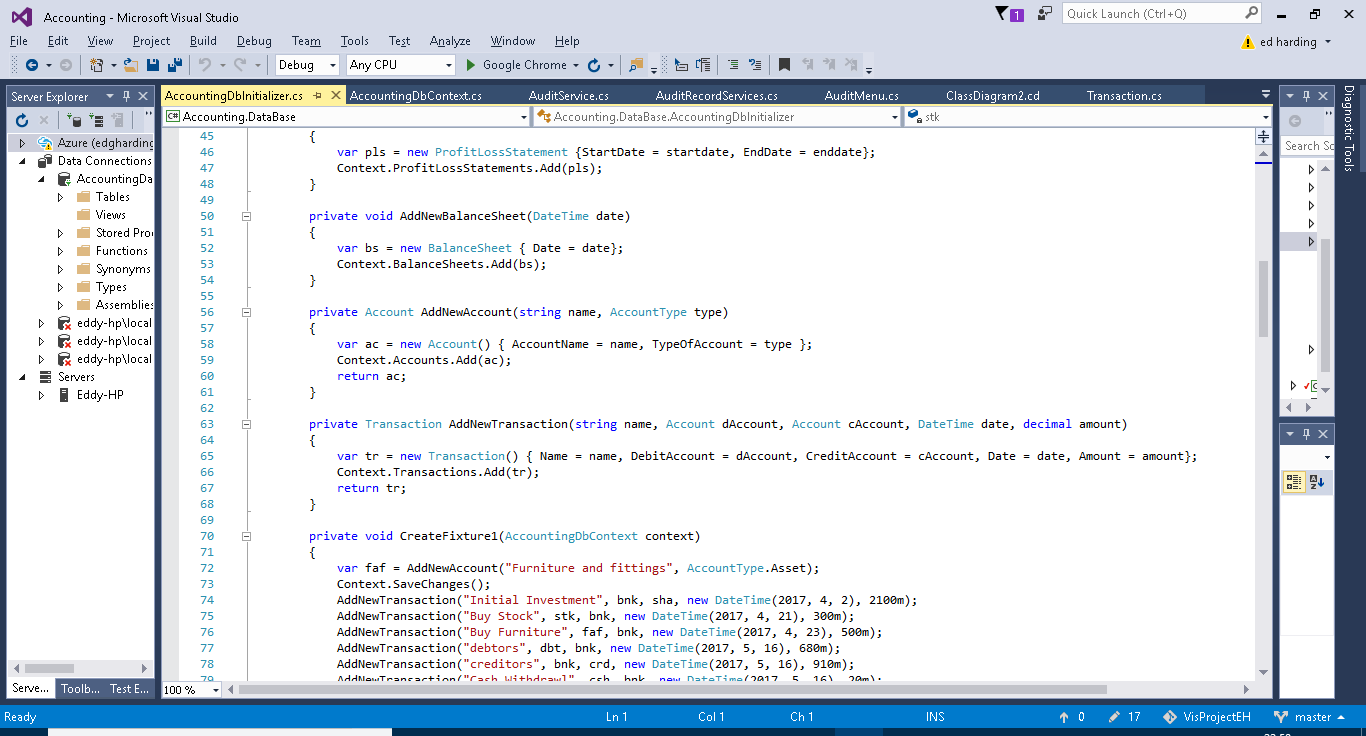
The first thing to talk about is the DropCreateDatabaseAlways on line 9. This line simply means that each time the solution Is run the old data from the previous session will be discarded and the seeded data held in the DB initializer will be added again. This means that the state of the solution is not saved. This is currently set this way for testing purposes. It is also possible to change this to DropCreateDatabaseIfModelChanges, this means that the data will only be reseeded if the Model project has been altered.

From lines 11 to 18 I have simply declared some accounts which are used regularly throughout my examples such as sha (shares) and bnk (Bank). Below this is where the data is being seeded. The Data in CreateStandardAccounts is added followed by one of the few options of data I have provided below that. The ones I am not using are commented out in green and the comments to the right of them just explain where the data used in them has come from. This system allows me to switch between different test and example data very easily.

Below:

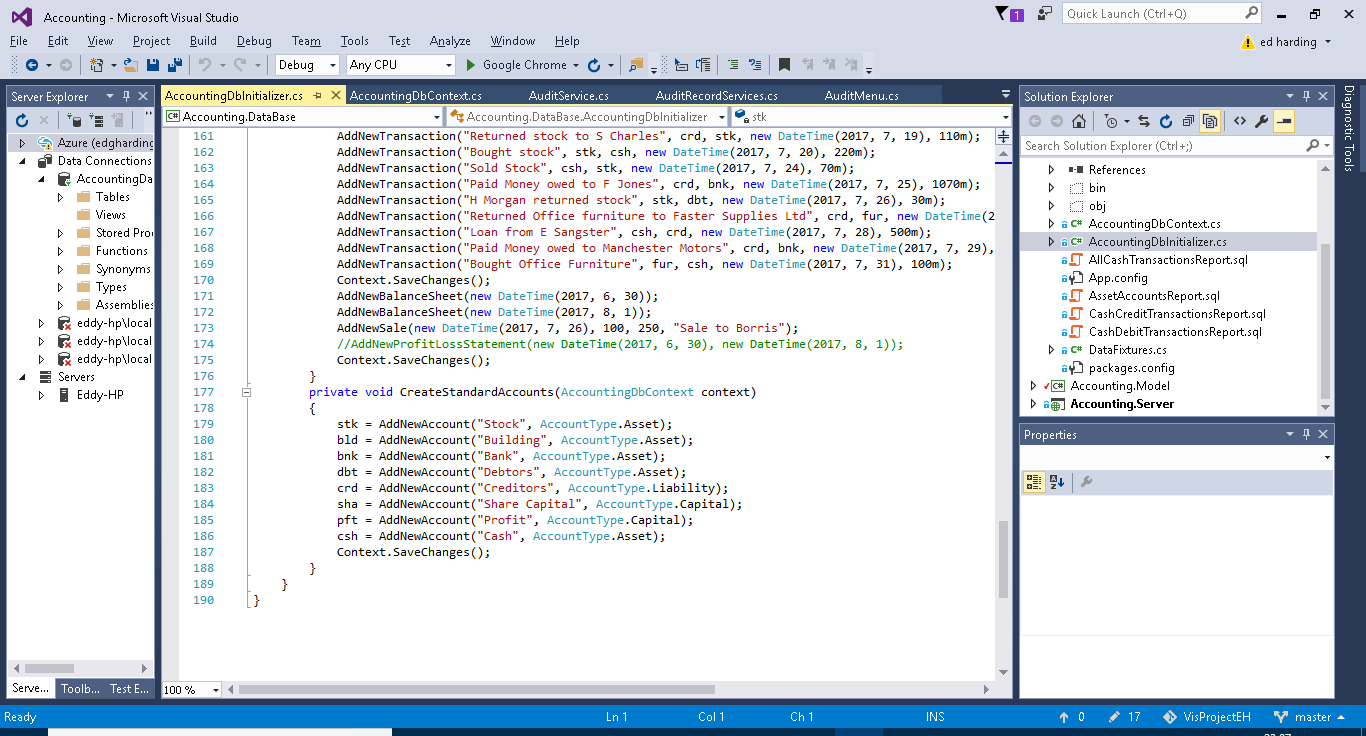
The following two screenshots just so some methods I needed to define for creating instances of the objects.





Below:

Here is the data used for the CreateStandardAccounts which I mentioned earlier, this uses the pre-declared variables from the top of the class file, it was necessary to declare the accounts outside of creating these data banks as the standard accounts are used in the test data, and therefore it had to be declared before use.



Below:

This is one of the multiple different examples of test data I added Here you can see the addition of new accounts, the addition of mutliple transactions and finally the creation of a couple of profit and loss statements dated before and after the transactions, to show the effect of the transactions.