## To do:

DB context, Auditing edits, Use of an I view Model, server restful api.

Also do the Analysis to get feedback.

## There are a few parts of the technical solution already marked which I haven’t included. This is everything but the functionality section of the Model Project (I know it’s the biggest and most important and will be by next step)

Did Client, server and database.

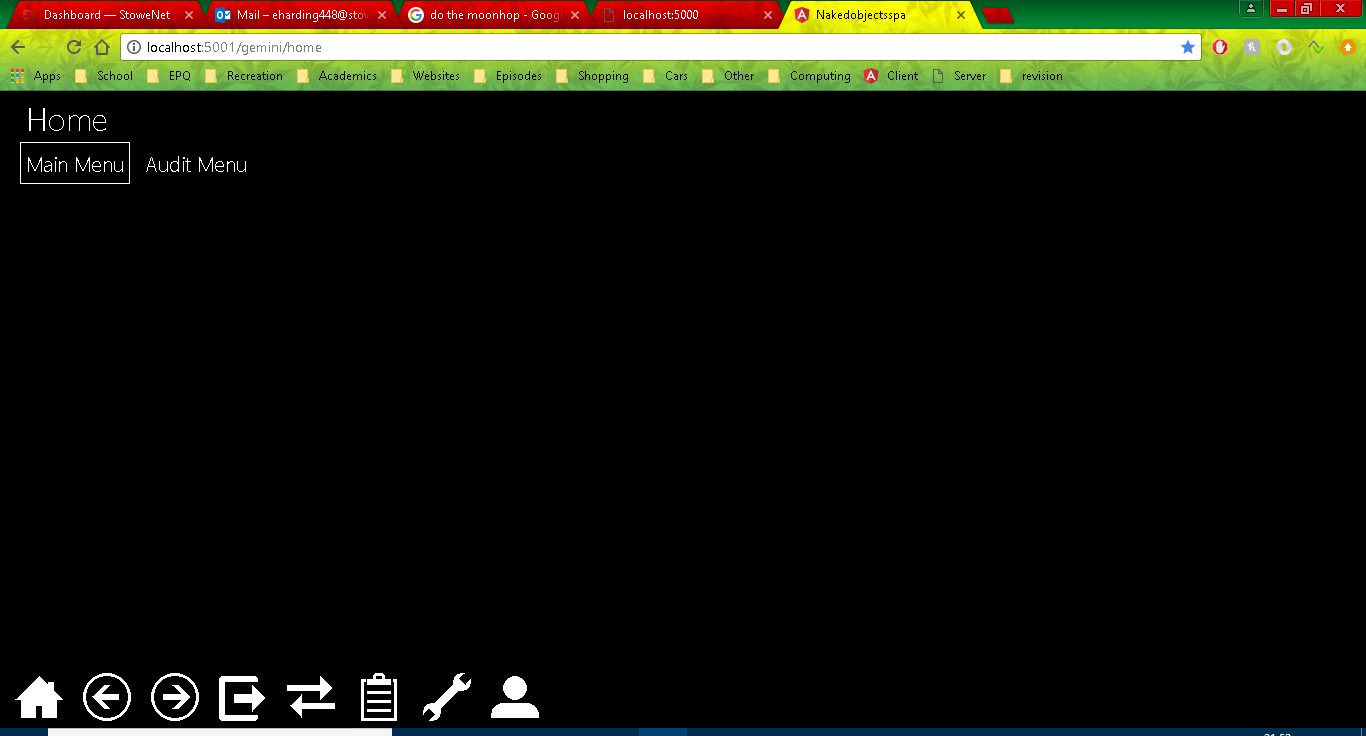
Roughly 1400 words added.

## 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

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.

Specifics (IViewModel) (Put Auditing section in here).

#### Functionality

##### Objects

##### Usability

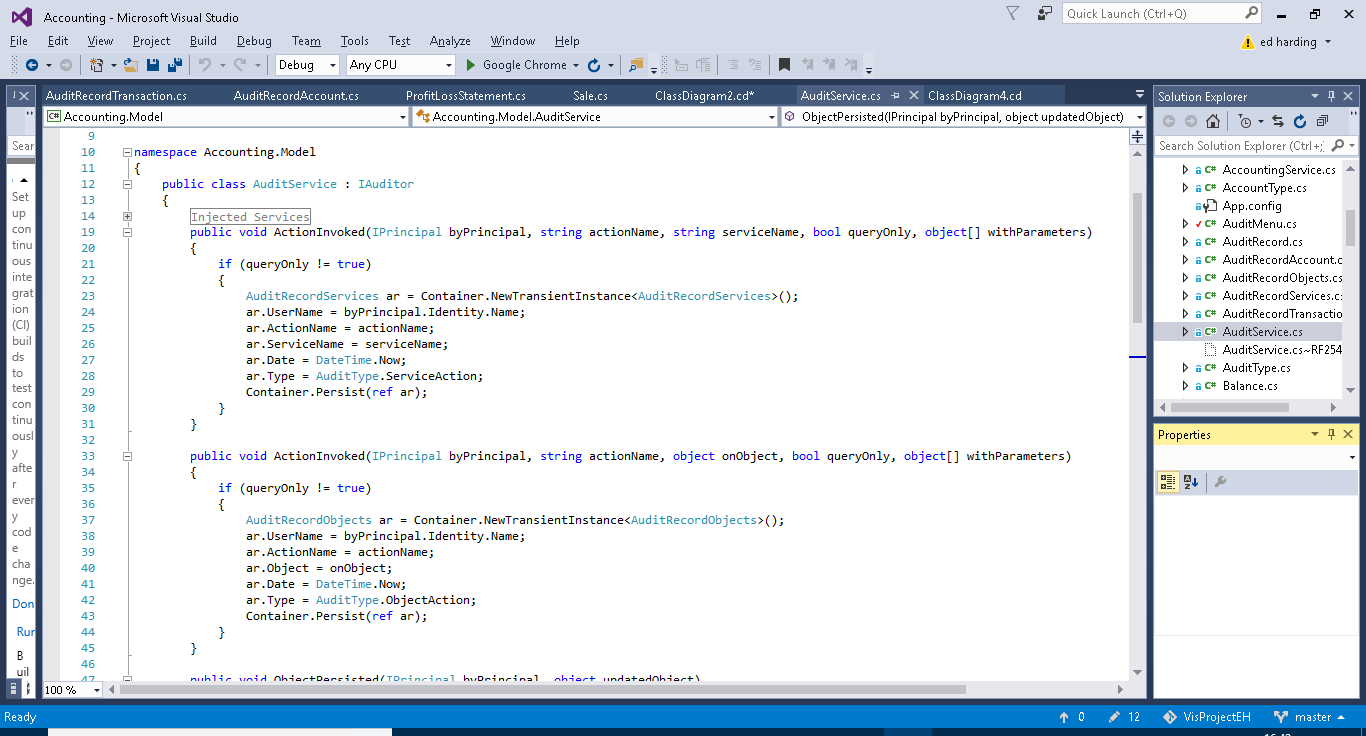
##### Robustness

#### Auditing

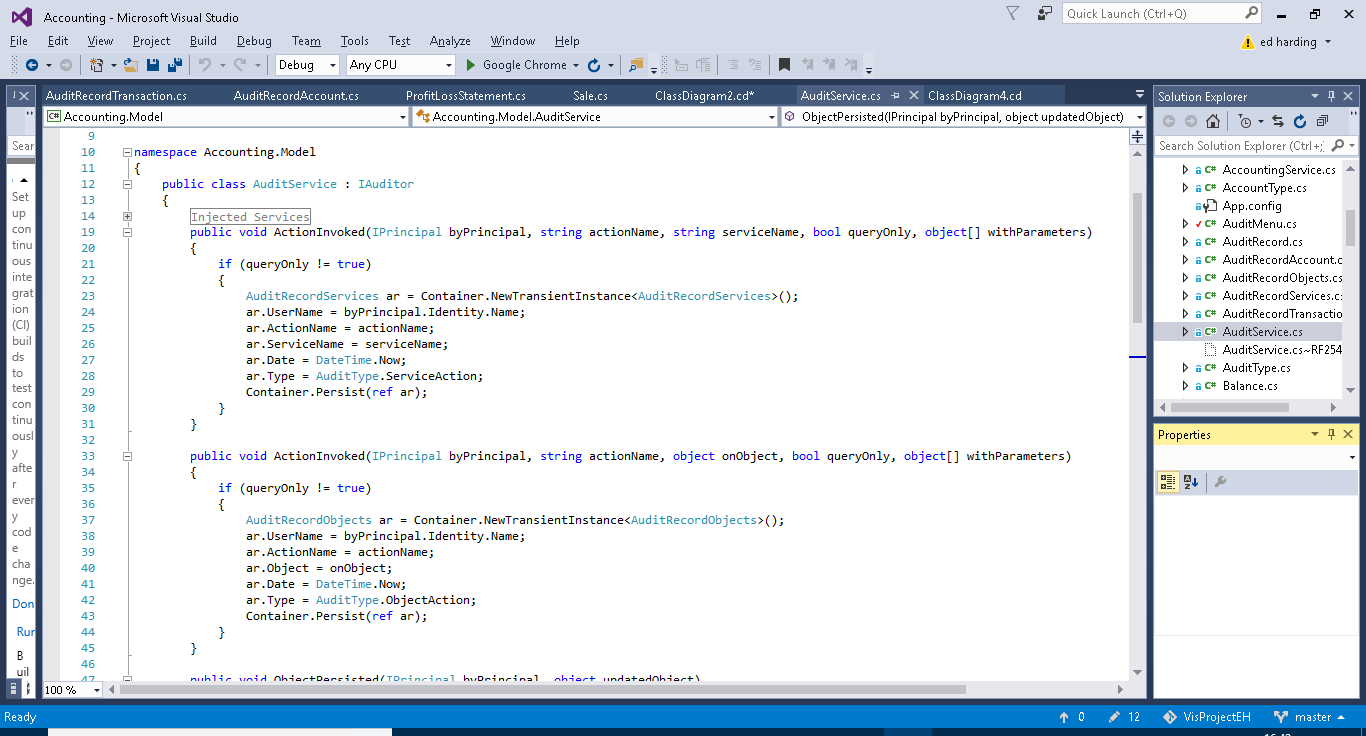
Auditing is one of the more complex capabilities of my program. My auditing code can be split into 3 sections (I will go into more detail of each below). The AuditService which detects when actions have been perfomed by a user, the AuditRecord which is used to save information about the action and user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

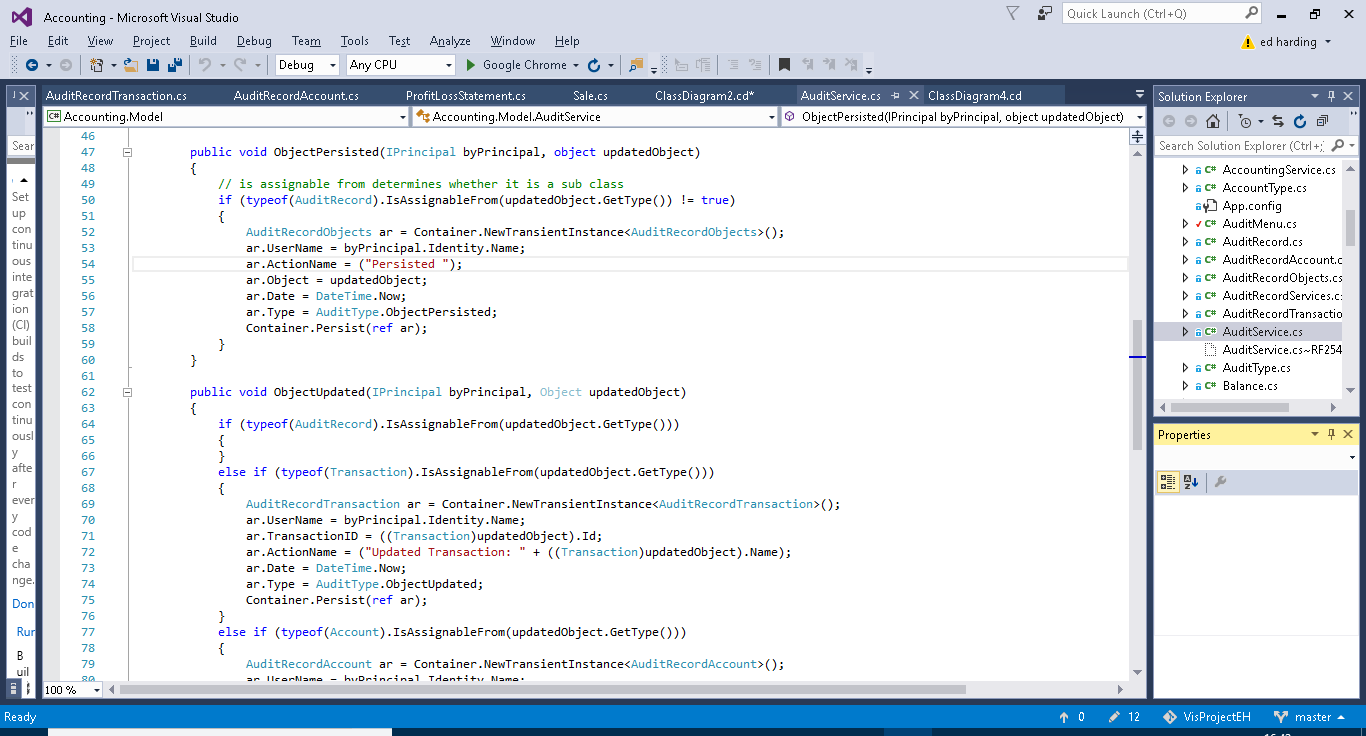
The Audit service constists of 4 methods each will be called under different circumstances.



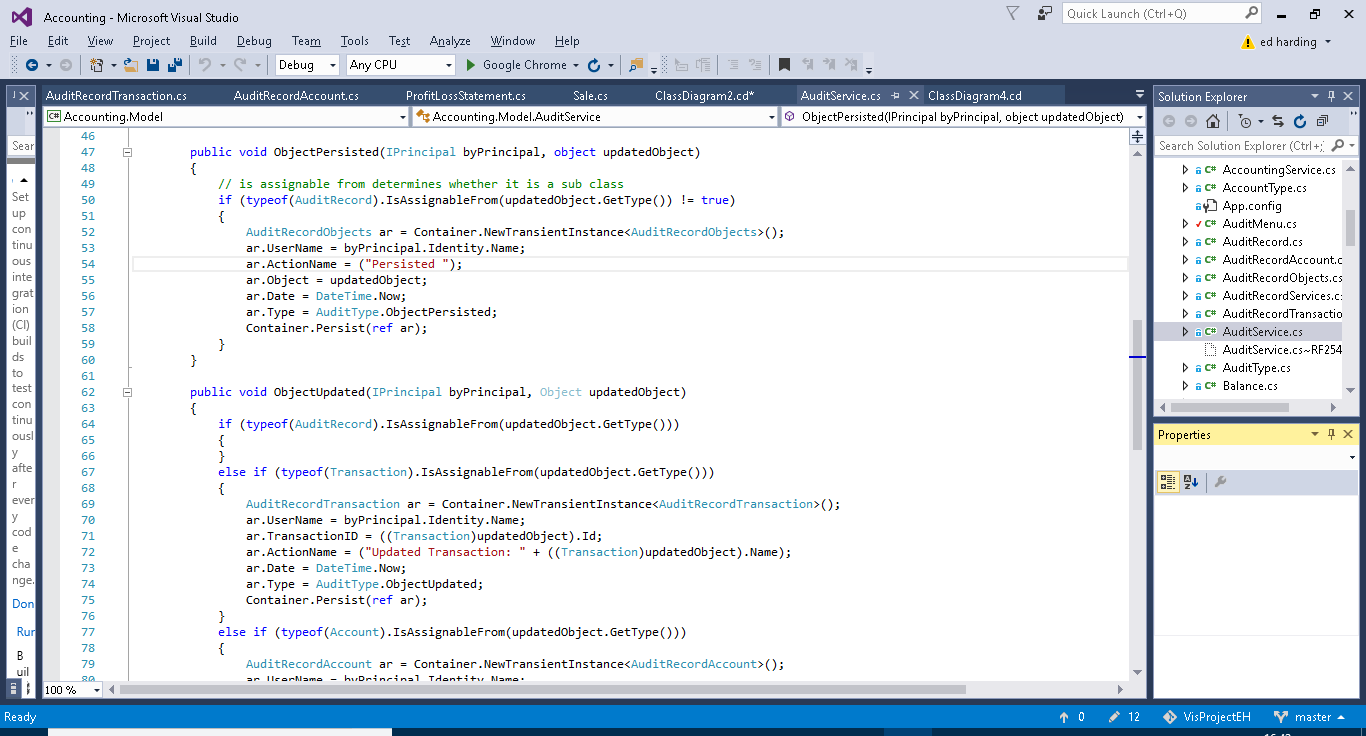
The first method (above) is called when a service action (e.g.) is executed by the user. The if statement checks to see if the action was a query only action, if it is nothing will happen as queries don’t necessarily need to be recorded as nothing critical is happening to the data. If the action isn’t 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.

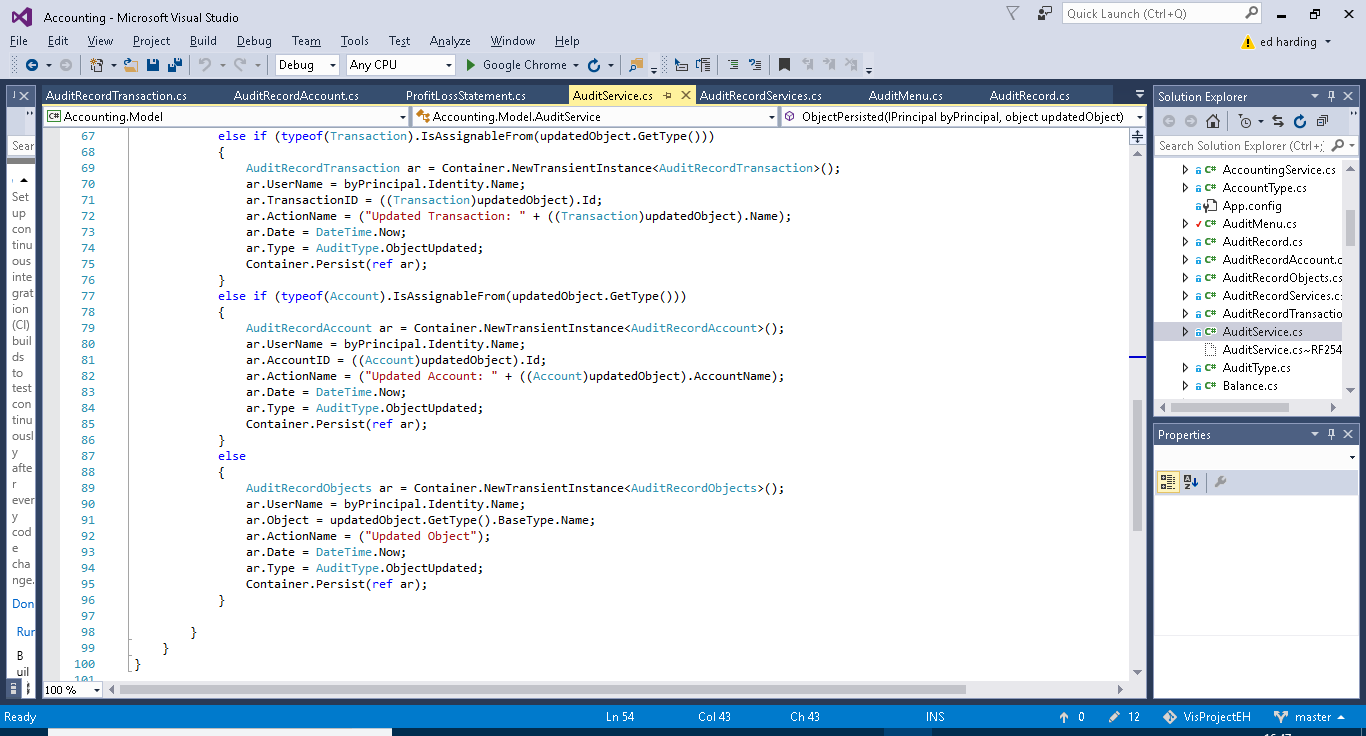


The second method is very similar to the first however it acts upon actions performed on objects (e.g.). Like the first method it checks whether the action is query only or not, it then creates a new instance of AuditRecordObjects, fills out the fields and persists the new instance.



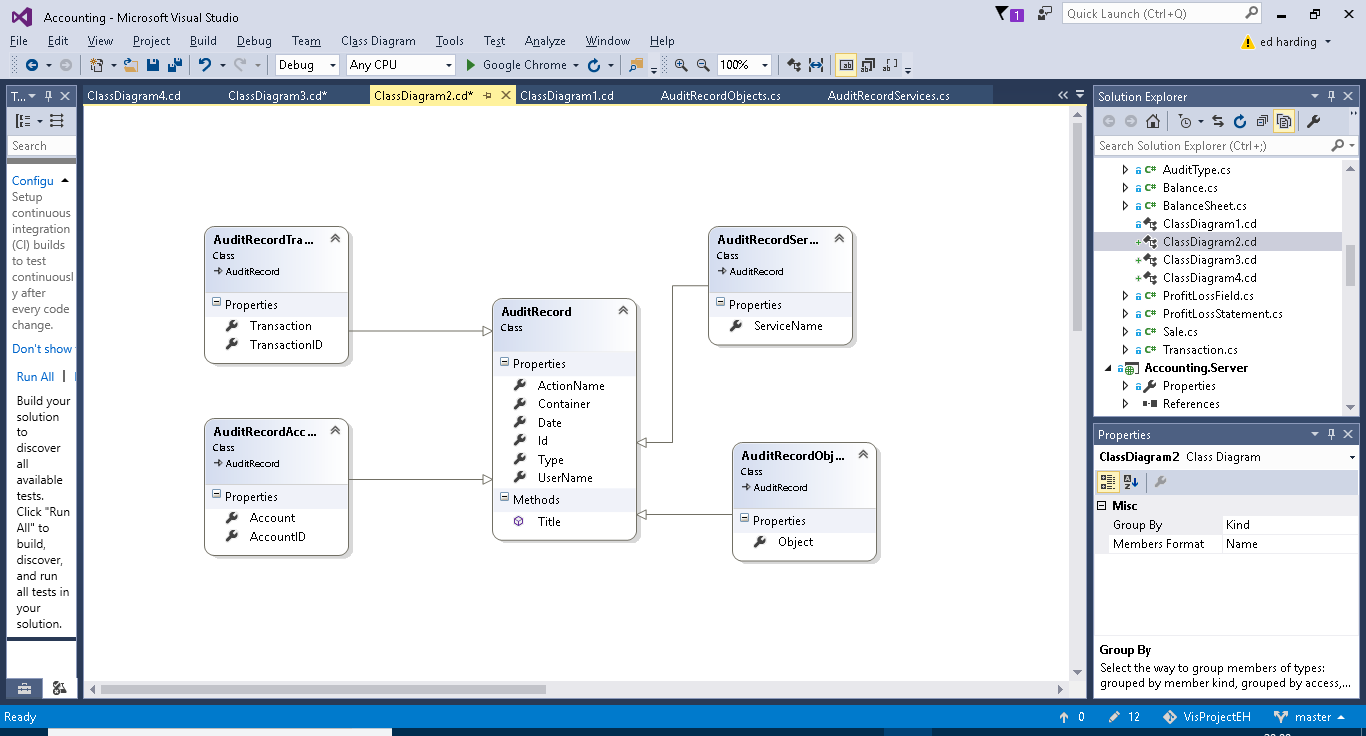
The third method is for auditing persisted objects. The first if statement is very important, its purpose is to assert whether or not the object which has been persisted is a sub class of AuditRecord (e.g. AuditRecordService), if it is then the method does nothing. This is important as without this if 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.



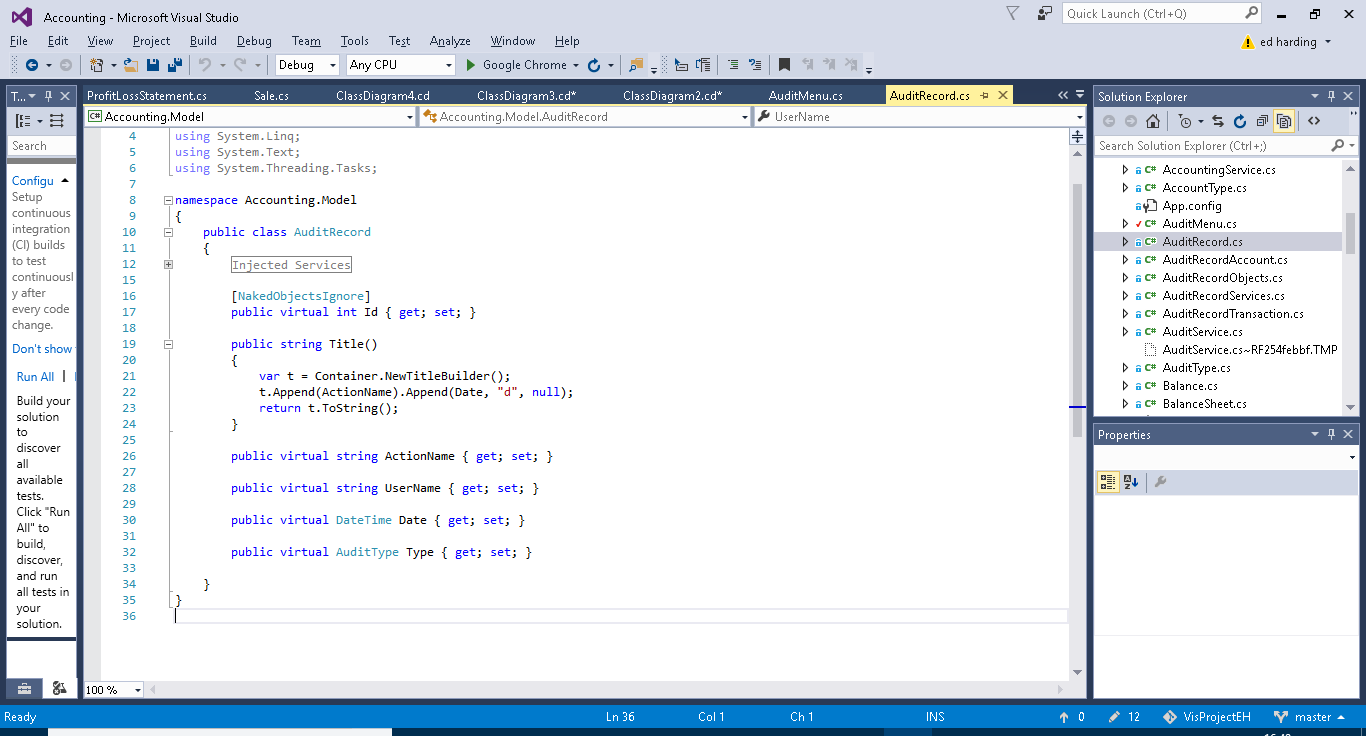


Similiarly to the previous method the first if case checks that the updated object isn’t a sub class of Audit Record, again this is to aoid stack overflow errors as described before. The next three if cases are to establish if the updated object is a transaction, account or other object. This is because 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

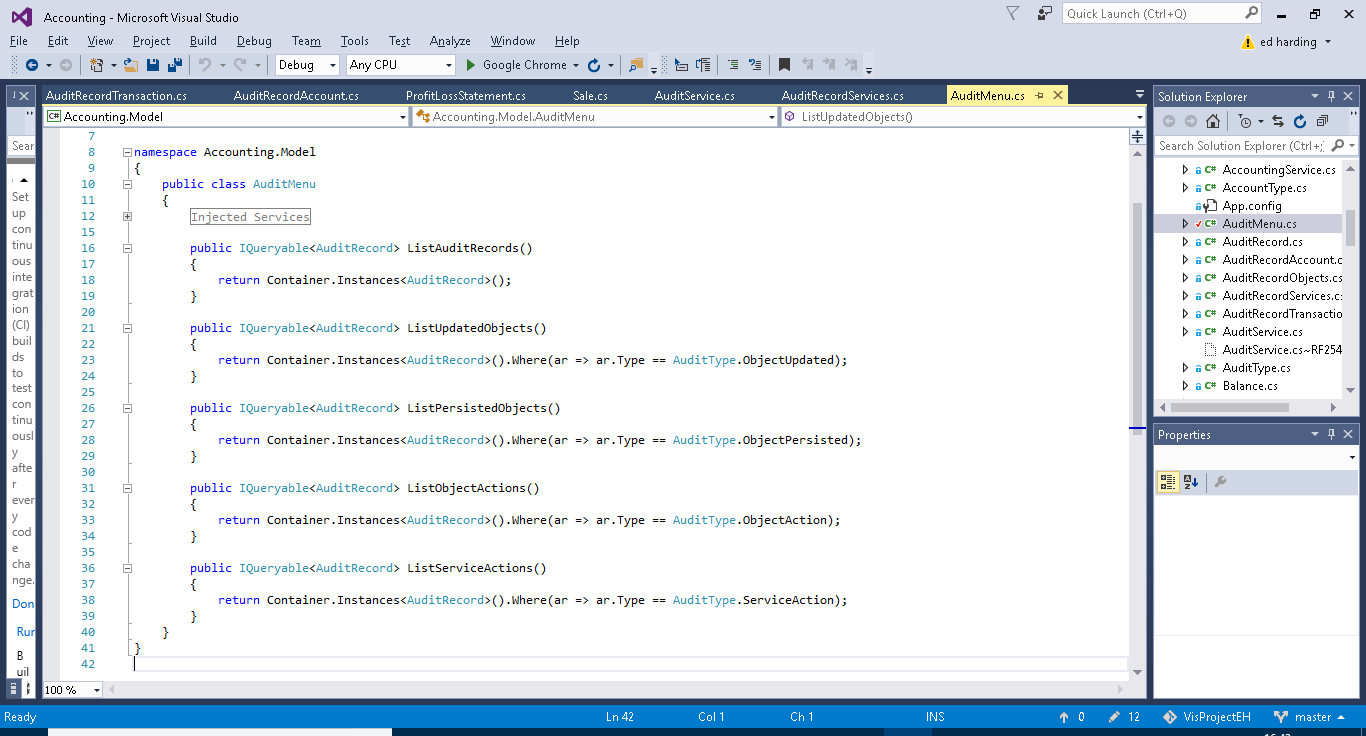


Above is a class diagram showing the four subclasses of account record. These were used in the methods in Audit Record (previous section).



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



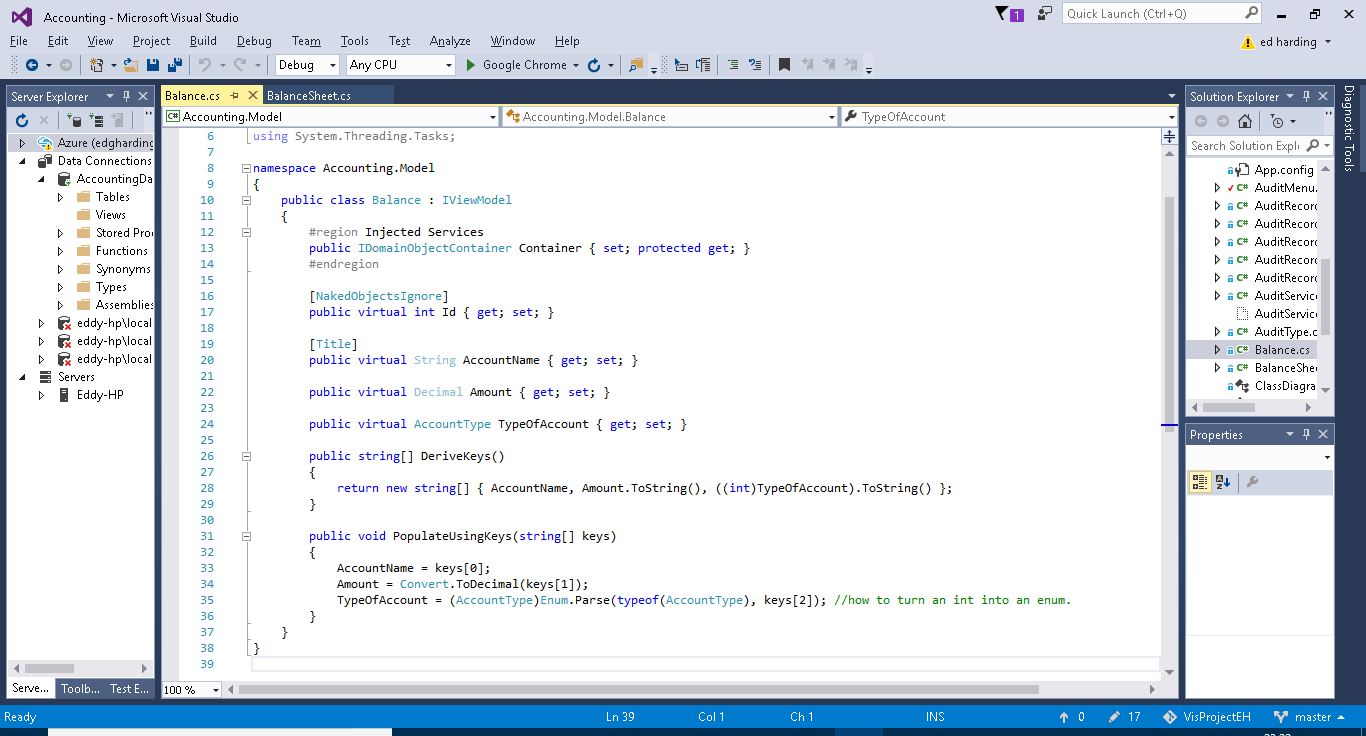
The audit menus sole purpose is to provide pre-built query methods on the entirety of saved AuditRecord. These methods just make it easier for the user to find what they want. The first method just outputs an entire list of all AuditRecord (this being all the instances of the subclasses of AuditRecord). 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. The balance of an account depends on the different credit and debit transactions performed on it.

Use view model so that doesn’t have to persist every time accounts balance changes, just changes with it

Created live, not persisted, used in balance sheet class for getting balance at a date. Server is stateless needs to be able to recreate exactly the same object. Show balance sheet class.



### Server

RESTFUL API

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. 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.

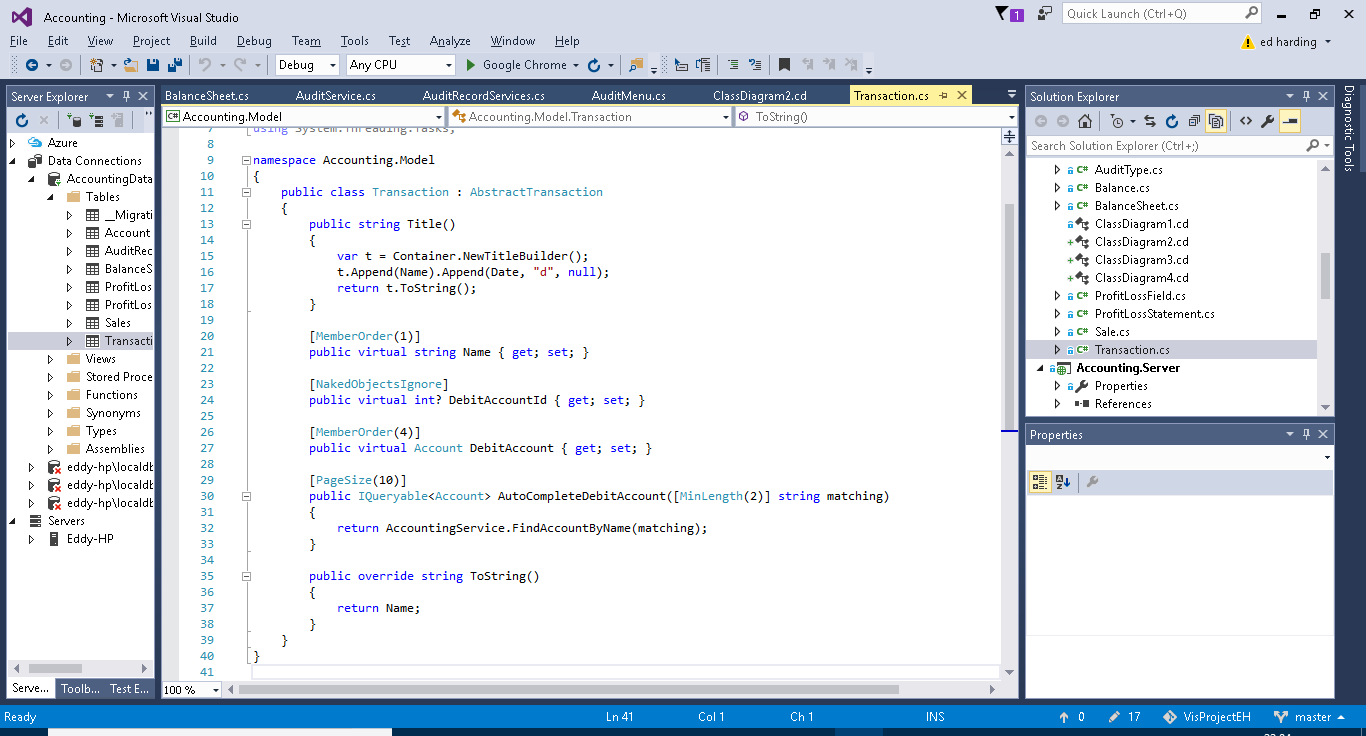
Add screenshots of run settings explain the configurations, add menu’s to run settings as stated in design.

### Database

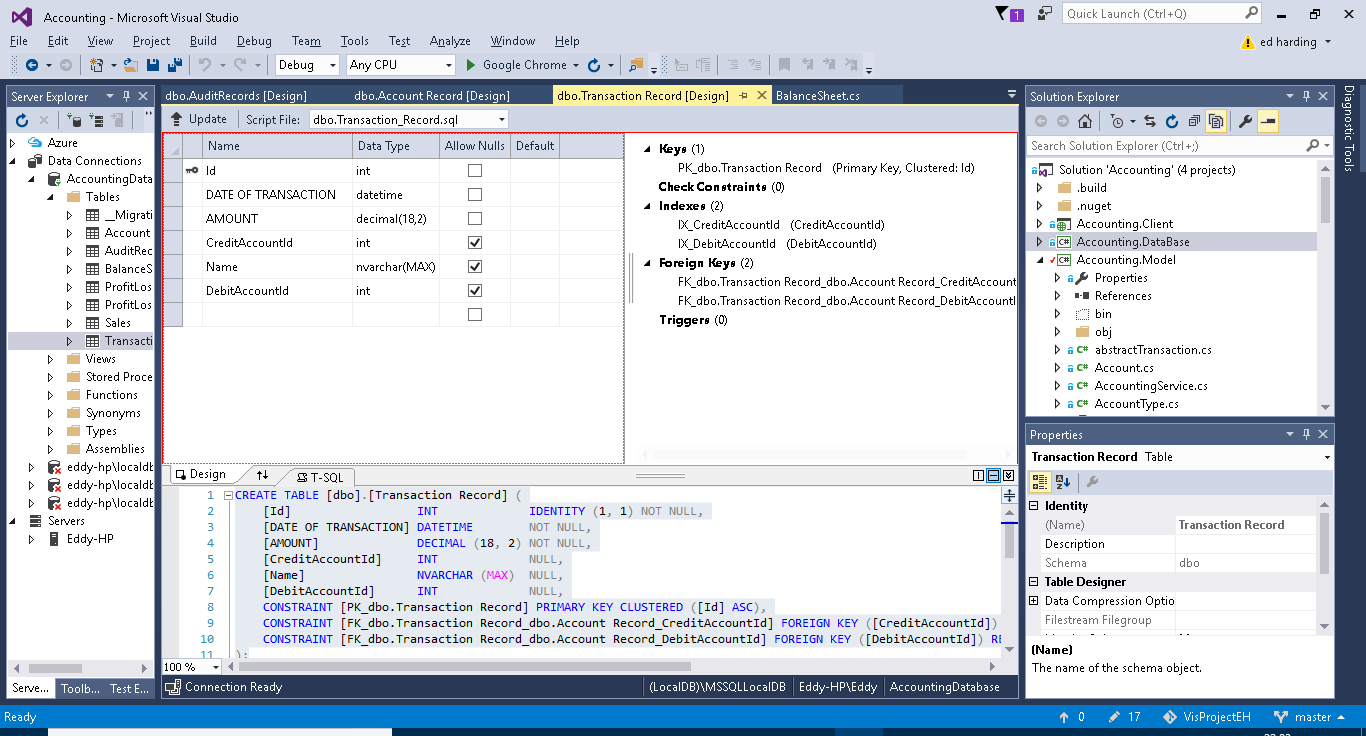
#### Entity Framework and SQL Servers

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.

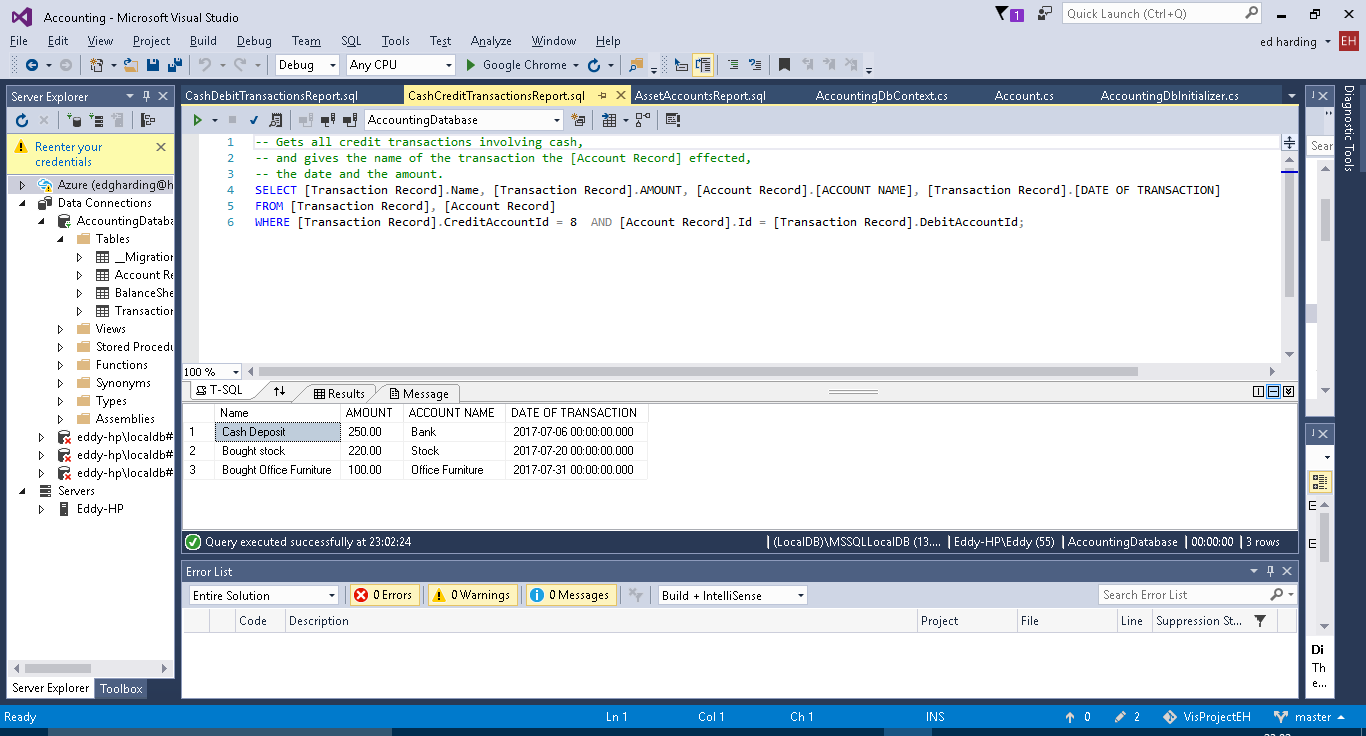
SQL Server is a relational database management system, which as mentioned earlier is used in tandem with Entity framework to model and query a database built from an object orientated project.



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

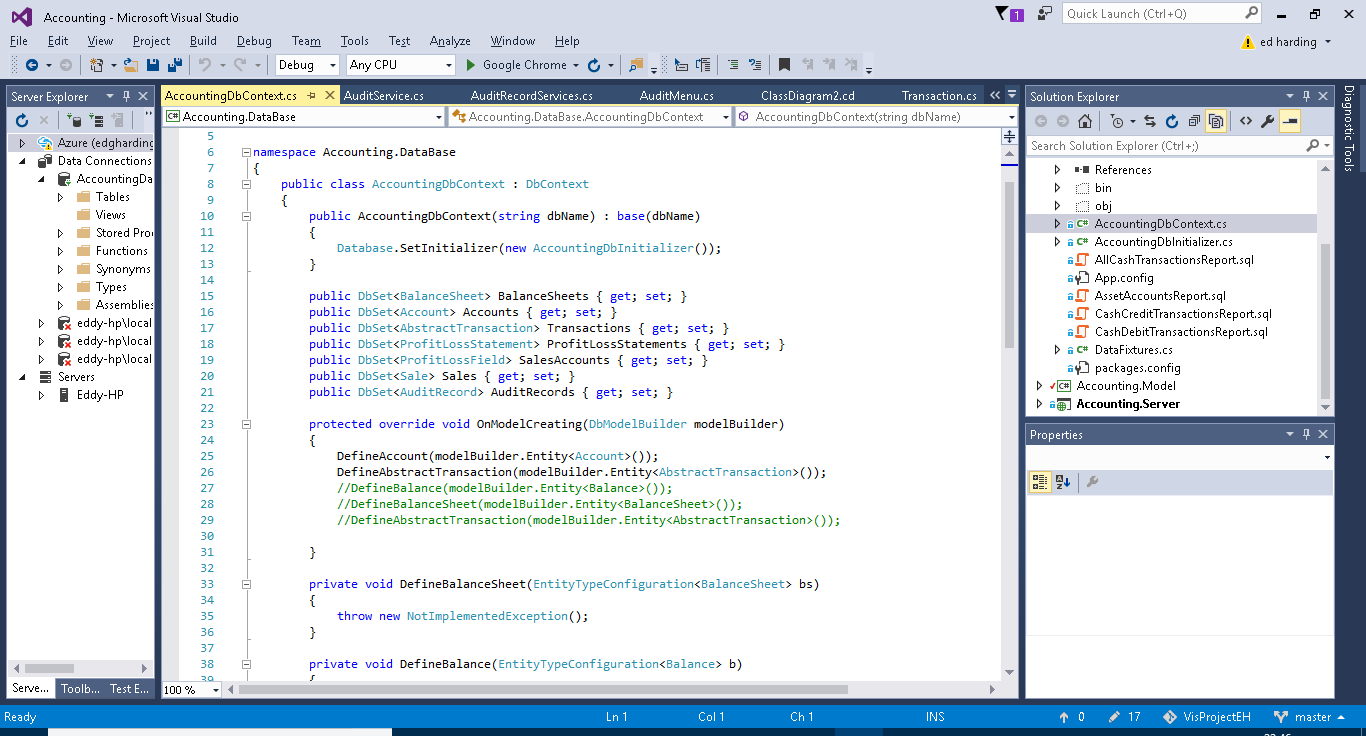


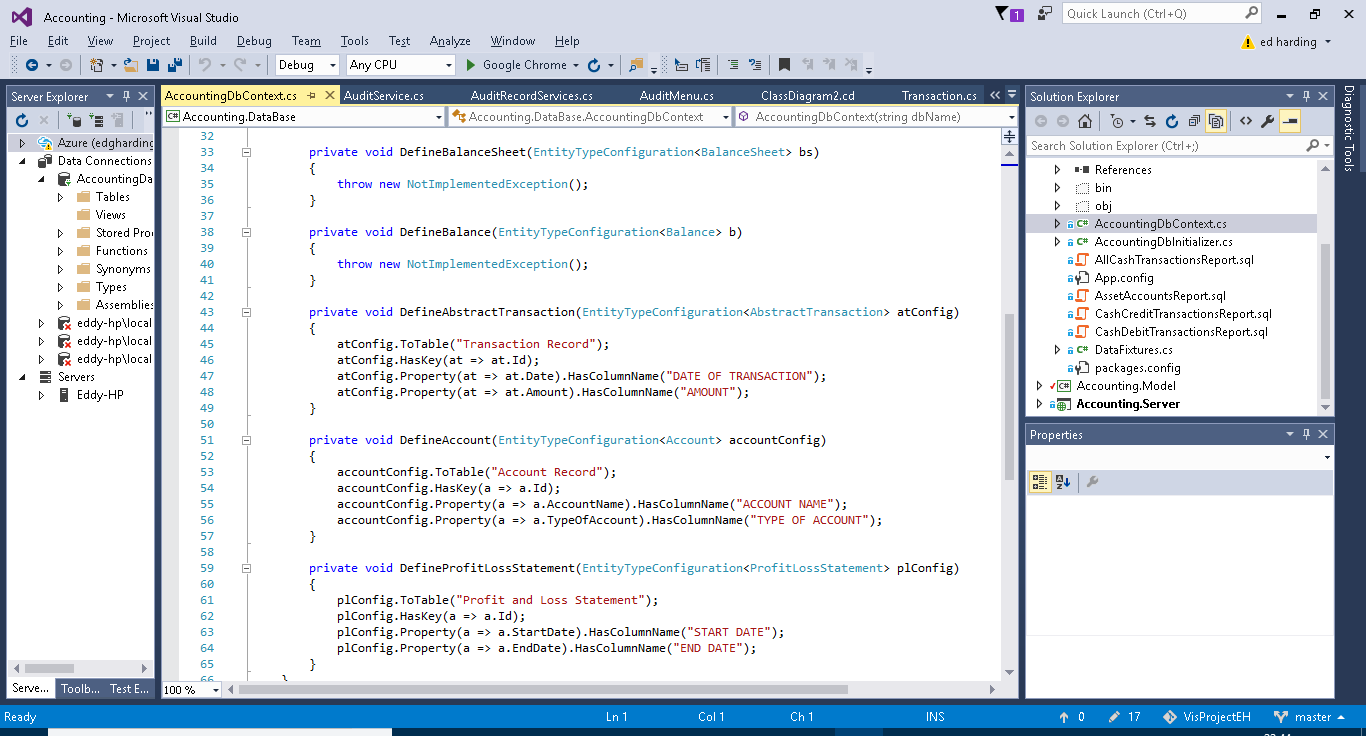
Below is a SQL query I wrote (what it does is explained in green in the screenshot). Below the SQL is the return message.



#### DB context

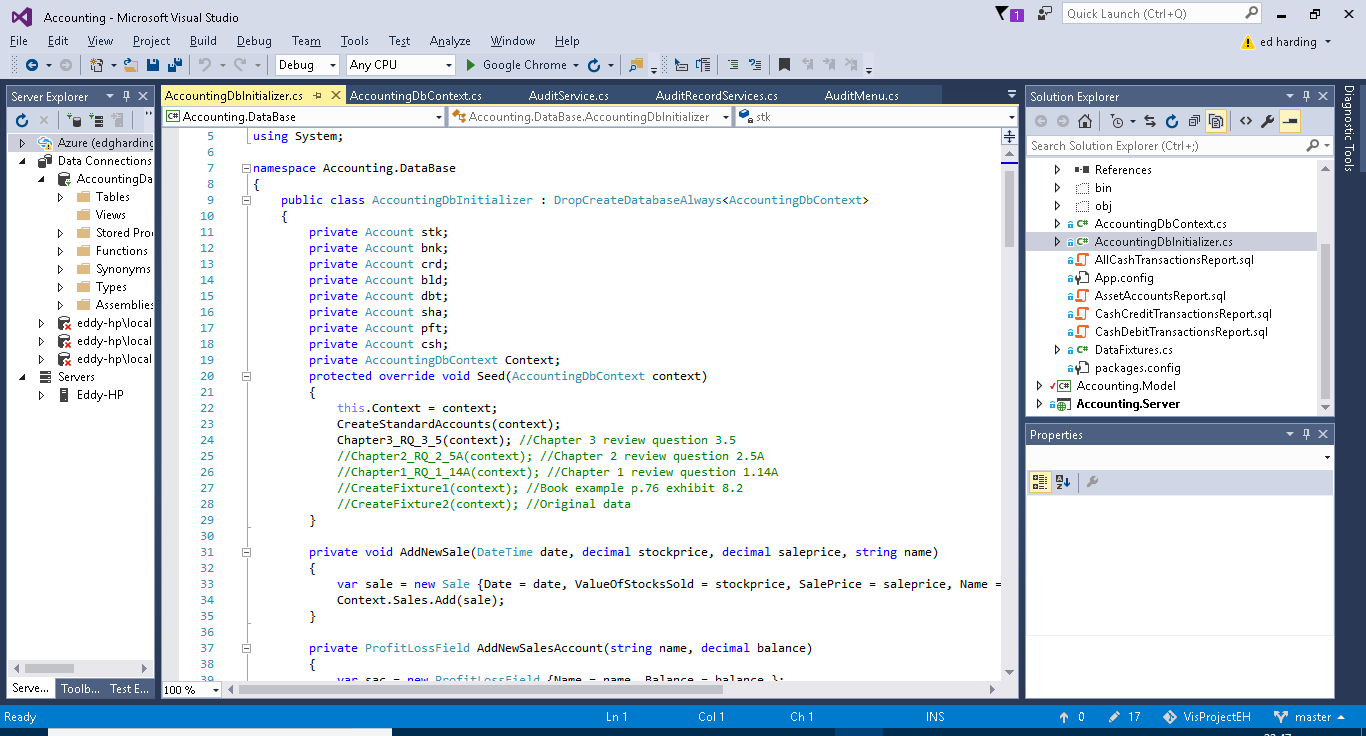
The DB Context class is used for persisting objects and … Need help here





#### 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.



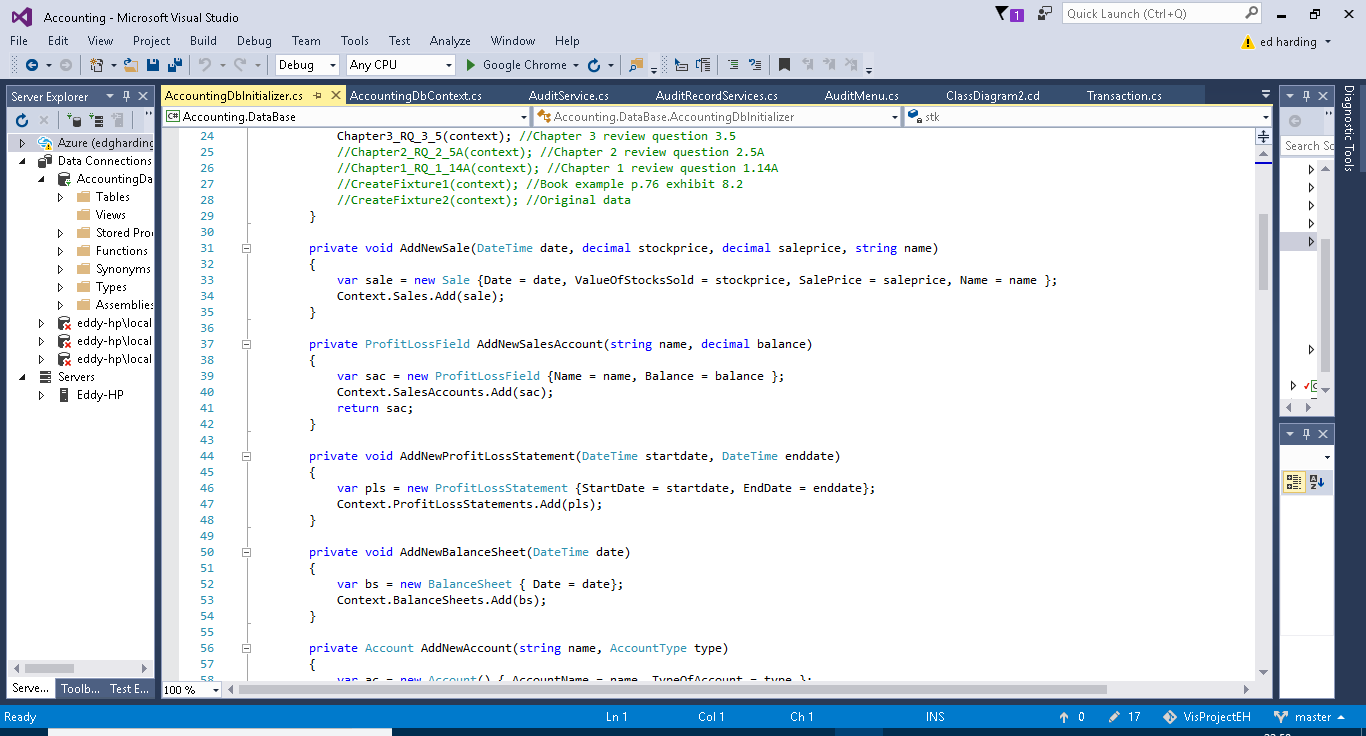
Above:

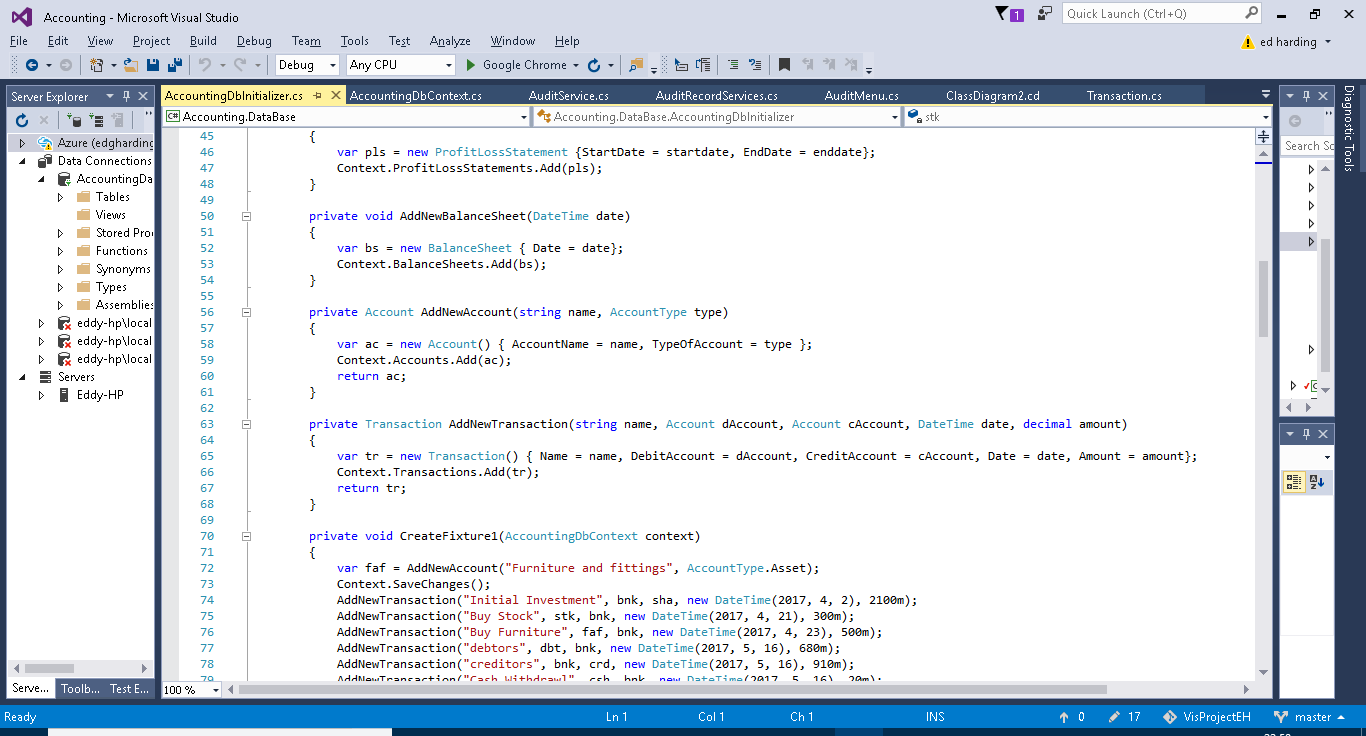
The first thing to talk about is the DropCreateDatabaseAlways on line 9. This line simply means that each time the solution Is run the old data from the previous session will be discarded and the seeded data held in the DB initializer will be added again. This means that the state of the solution is 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:

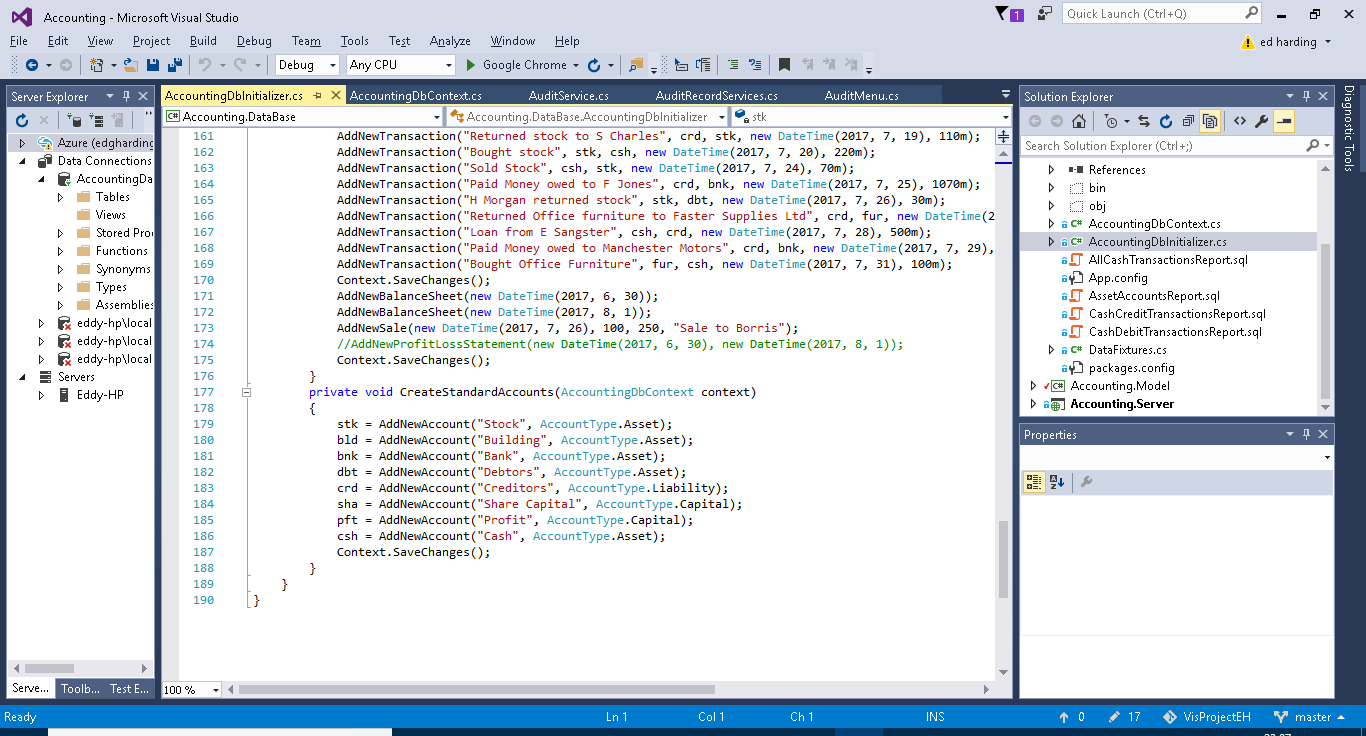
The following two screenshots just so some methods I needed to define for creating instances of the objects.





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.