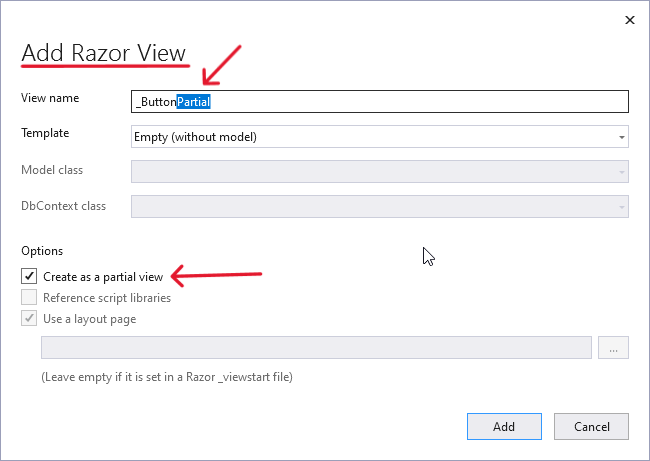
**Partial View:**

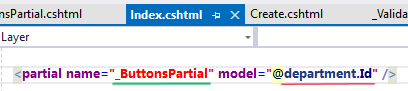
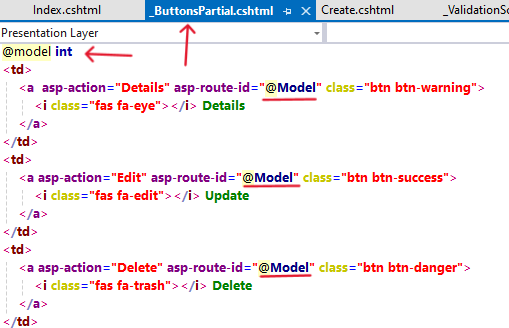
We use partial view to save repeated code

We render it in the views that has the common repeated code

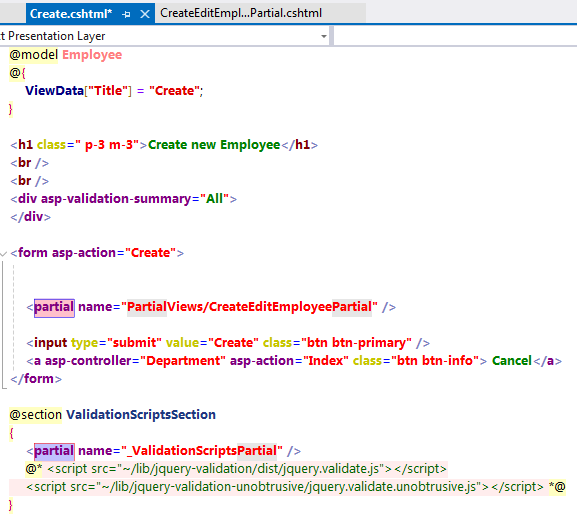
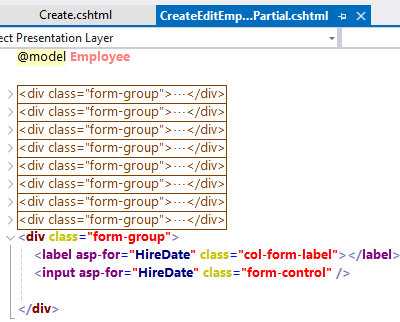
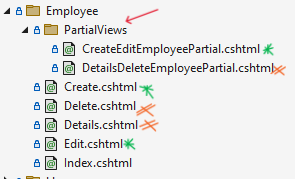
We used to name it \_XPartial 🡺 end its name with Partial word



Remove repeated buttons from Index cshtml files [Department & Employee] and add them in the Partial view of Buttons



Repeated code of Employee views 🡺



**View VS Partial-View Vs Layout VS Section Vs View Start VS View Import** 🡺 Interview Question

View: html page that action [action inside MVC controller] return it as a response

Partial View: repeated part of code written in separate file and rendered in the views containing that repeated code

Layout: structure of the page, views having the same structure use the same layout

View start: we write in it the code that we need to be added in the start of each view [like using a default layout for any view]

View import: the imports we use in more than one view so that we don’t have to repeat the imports using in each view

**Binding:**

Action [index] works as HTTP-Get, send piece of info from action to the view related to this action [the view that the action will return as response]

So binding is sending info from action to view

note the action that has verb get [the action sends info to view] while the action that has verb post [the action receives info from view]

ex: the action Index [http-get] sends model [Employees] to view [the model that will be displayed in the view], the action creates [http-post] receives info from the view [the model that will be created] when click submit button

the direction from view to action or action to view is one way [can’t send info from view to action then receive info from action to view]

the model is the main info send between view ⬄ action

the main info is the model … what if we need to send extra data? We can store the extra info in the storage of the view [Dictionary] as each view has dictionary[storage]

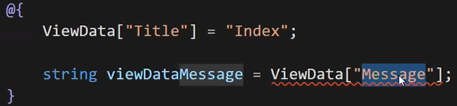
set the data we need to use in the view inside the action [in the action method in the controllers] then use it in the view [set the data in the view storage and use it in the View]

we can access the View Dictionary using 2 ways:

* ViewData[old]: more safe and faster than ViewBag, uses Dictionary [key value pairs]

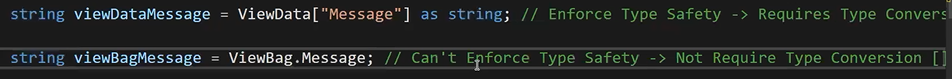
Compiler enforces type safety … requires type casting/conversion

Faster as its type is detected in compilation time not the runtime



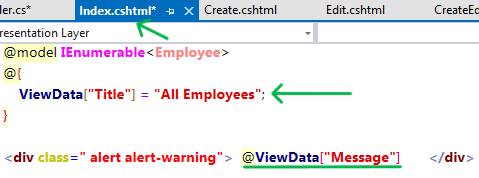
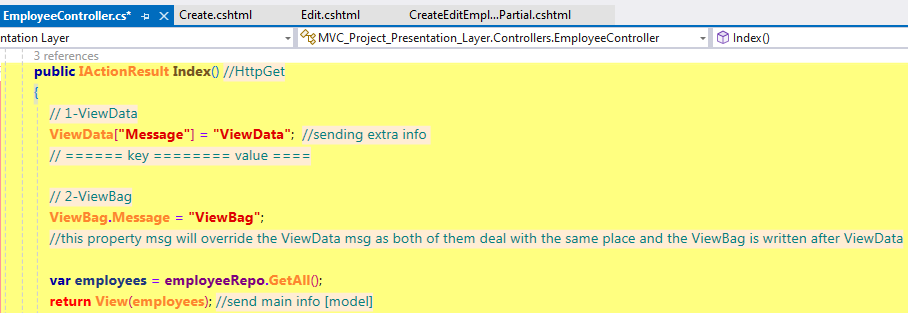
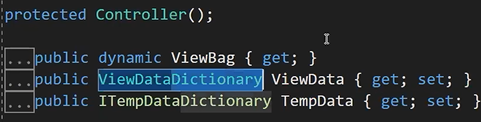
* ViewBag[new]: based on dynamic keyword [dynamic property]🡺 which means it will detect the type of the object in the runtime that’s why ViewData is better and faster, we use it to access the view dictionary in a different way

Don’t need to worry about the casting just use the dynamic ViewBag property and CLR will determine its type



Each View has its own dictionary, so if we used the key of index view in the create view we’ll find out that it will be null [if page is redirected to another page it will look for the key value in the dictionary of the redirected view]

The controller class inherit those 2 properties from Class Controller [not ControllerBase as ControllerBase doesn’t deal with views]



Note: ViewData & ViewBag store in the same place

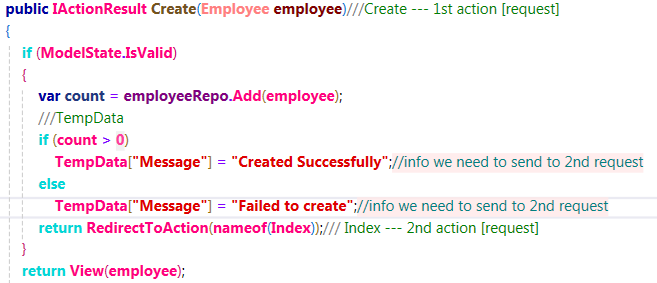
* TempData: is used to send data between 2 consecutive requests [from current request to next request]

Inside the Create action we have another action [redirect to another view] Index action

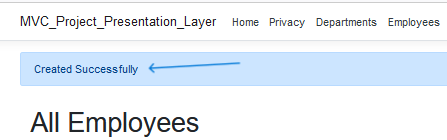
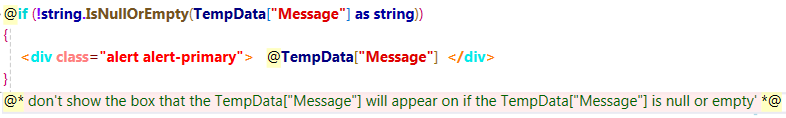
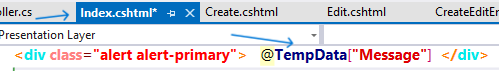
We need to send info from the Create action to the Index action… send a piece of information to display it in the next action [after the Create action send me info about the object if it is created or not]

TempData is similar to ViewData in structure both of them are dictionary but with different usage

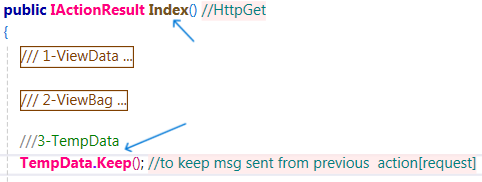
TempData & ViewData have different dictionaries



display the message in the Index View



If we need to keep the TempData sent to the index to send it to another action [another request] go to the Index action and Keep it 🡺

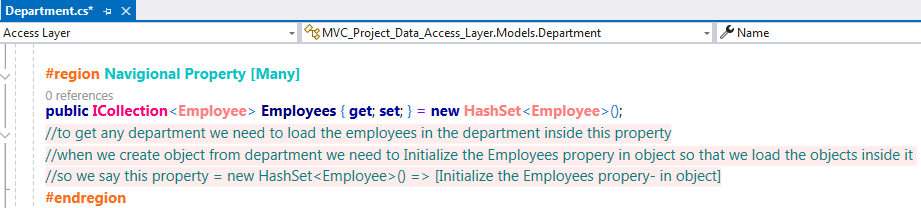


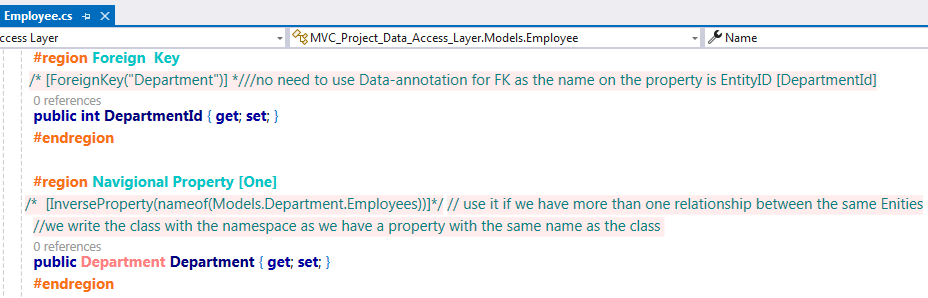
<https://www.c-sharpcorner.com/blogs/viewdata-vs-viewbag-vs-tempdata-in-mvc1>

**Employee Department Relationship [one Department🡺 many Employee]**

PK of 1 🡪 many [add it as a FK in Employee table]

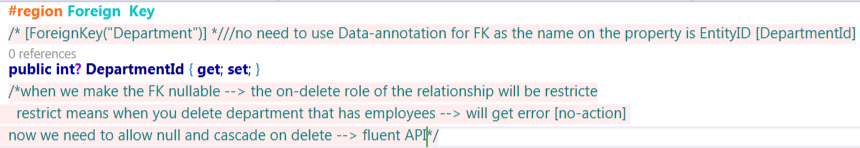
As we are working Code 1st so we’ll do that using navigational property

Go to the DAL project 🡺 Employee Model [Employee only works in one Department] 

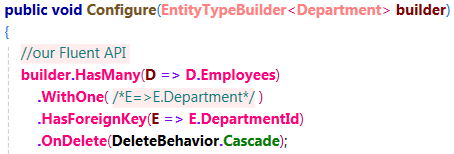


If The Fk doesn’t allow null 🡺 on-delete [when we delete department that has employees] action will be cascade[delete all employees in that department] while if at allows null the action will be no-action

To make it allow null and on-delete cascade 🡺do it fluent API



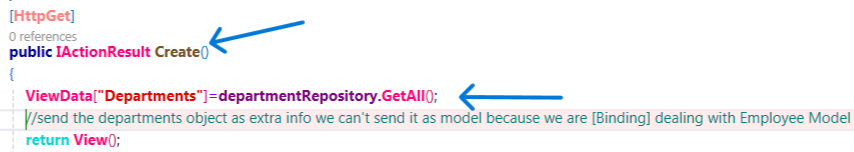
We have navigational property in both sides so we can make configures [fluent API] in any configuration class of them [remember if we make navigational property in one side (Department) we go and continue our configuration in the same side (DepartmentConfigurations)]



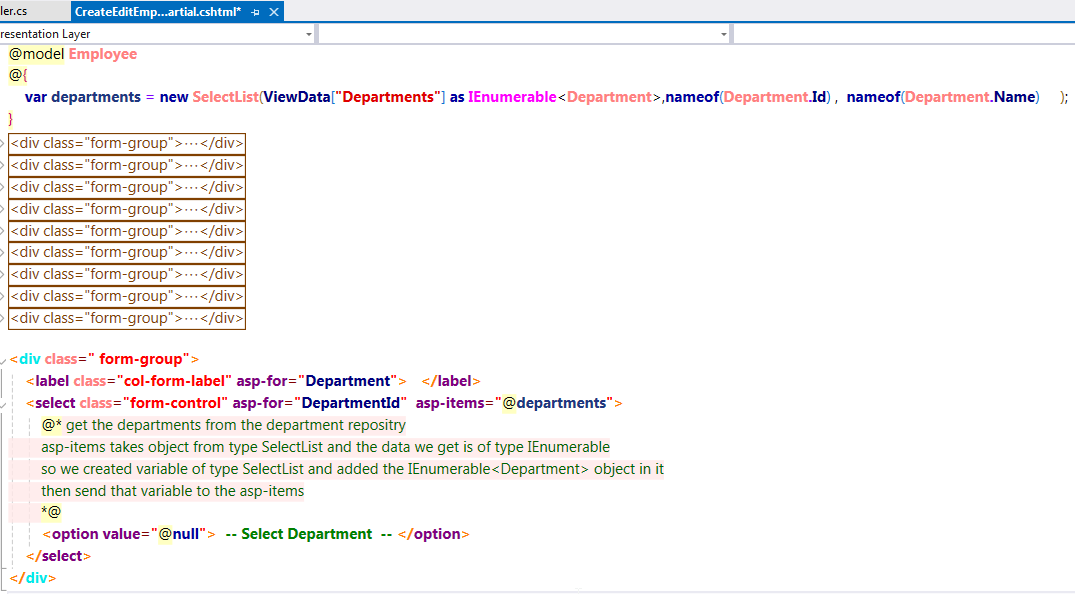
**Select**

We need to add input for the Employee create and update pages so that user can choose the Department from that input [dropdown list]

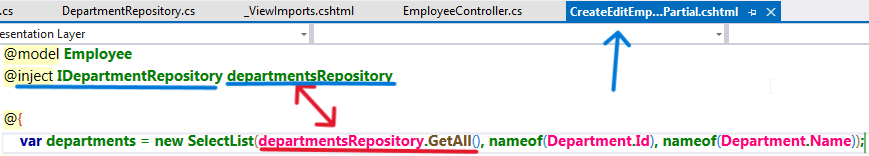
* Allow DI for DepartmentRepository in the EmployeeController Constructor and make a read-only attribute for that object of type IDepartmentRepository so that we can save the objects we get from the constructor in it and use it in the create method… the department repository is responsible for getting departments data from database
* Send the data we get from the DepartmentRepository as extra info using ViewData so that we can display them in the Employee Create/update view



Using the ViewData in the partial view🡺

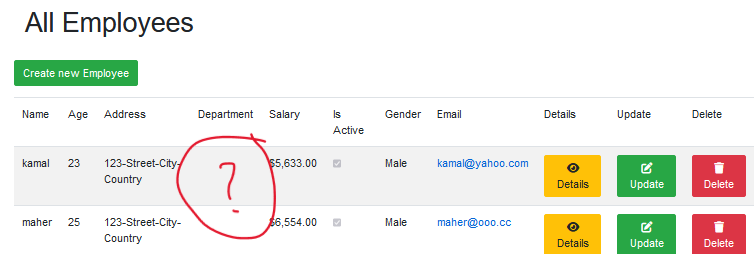


* This Code [EmployeeController] has performance issue🡺 when we call the Index action method we call it from object as actions are non-static object member methods🡺 creating object from EmployeeController depends on creating object from class implements IDepartmentRepository which is un-needed object in the Index action so in case a user just want to retrieve all Employee data the CLR will also create un-needed object from DepartmentRepository class
* So to solve this performance issue we’ll inject the object we need in the view/partial view that will use it not the whole controller class that may have actions don’t need it
* After Injecting the object directly in the view, we won’t need to send extra info in ViewBag/ViewData
* Remove DepartmentRepository injected object from EmployeeController and add it in the View 🡺



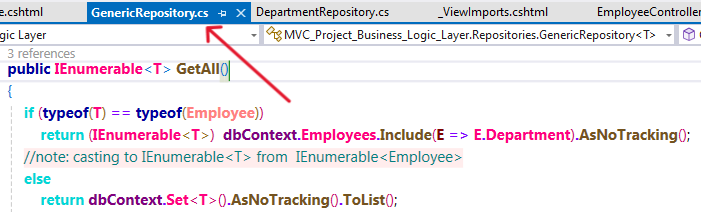
Why Departments are not shown although the employees are assigned to department?

* The department to the employee is a navigational property[related data] 🡺 it won’t be loaded by default , it has to be loaded by one of the 3 loading ways [explicit/lazy/eager]



**To Remember** 🡺

* lazy loading is similar to explicit loading but it loads implicitly
* so now we have to decide between the 2 loading ways [eager/lazy]
* before deciding we have to enable the lazy loading mode by installing the package [Microsoft.entityframeworkcore.proxies] which has extension method [uselazyloadingproxies] 🡺 call this method before calling the method [usesqlserver]
* then make all the domain models public and all navigational properties virtual [so lazyloadingproxies can override the navigational properties to work in the way of lazy loading]
* then decide to use lazy loading or eager loading
* lazy loading, we just enabled it no need to do anything else🡺 it loads the main data in a request and if we need the related data it loads it in another request
* if we decide to use eager loading use include method 🡺 eager loading loads the data and related data in the same request
* now decide [eager loading or lazy loading] 🡺the navigational property ONE we use eager loading
* while the navigational property MANY 🡺 it depends if the relation between the related data and the main data is composition [required]we use eager loading , while if the relation is aggregation [not required as part of the main data ]use lazy loading
* now we need to load the departments in the Index view [Index Action] of employee
* the index page gets its data by executing the GetAll method that is inside the EmployeeRepository which is inherited from the GenericRepository
* go to the GenericRepository class and inside the GetAll method [related to index method] add the Include method which enables the eager loading
* now we have problem in the design 🡺 we can’t use the include method inside the GenericRepository as it is generic not EmployeeRepository … it takes type T that type T we put a constrain to be from type ModelBase ..that ModelBase class only have one property which is the Id so if we used the include method we can only include the Id … solving this problem requires **Specification Design Pattern**
* we’ll solve this issue temporarily till discussing **Specification Design Pattern** in the web API 🡺
* we can make the method virtual in the GenericRepository class and override it in the EmployeeRepository and use the include method there
* or another solution in the GenericRepository class by using if statement and check if the T of type EmployeeRepository use the DbSet of employee and include department else use the generic normal method



**Service Lifetime:**

The lifetime of the service that we use in dependency injection, when to say that the service that we register in the container to allow dependency injection will be transient/scoped/singleton

* Transient: per operation🡺 will create object per الطلب inside the request, in the same request if we asked 2 times for object, it will create 2 objects

The mapping[model ⬄ view-model] service is usually transient as this operation only happen one time [mapping the model to view model ”GET” and mapping the view model to model “CREATE/UPDATE/DELETE” is only happen one time per request so we need only one object that’s why we make it transient]

* Scoped: per request 🡺 example: the IEntityRepository is usually scoped lifetime which means once the request is ended the object will be removed from heap [not really removed from heap but will be unreachable for us[its reference will be removed from stack]🡺 for the object to be removed from heap the garbage collector must remove it automatically or you force it to work and remove unneeded objects from heap]

Repositories appropriate lifetime is usually scoped

The DbContext usually scoped lifetime 🡺 we don’t make it singleton as we may have another database. Example: the departments and employees are in different databases; the user may request the employees then make another request for departments if the life time is singleton the CLR will use the same DbContext object and won’t create another DbContext object related to the other database … so we make the DbContext object lifetime pert request “Scoped”

* Singleton: per application per user 🡺 only one object per the application for each user [note that the singleton is levels we may have a singleton that only creates one object per application for all users/per server]

Serveries that is usually created with singleton lifetime:

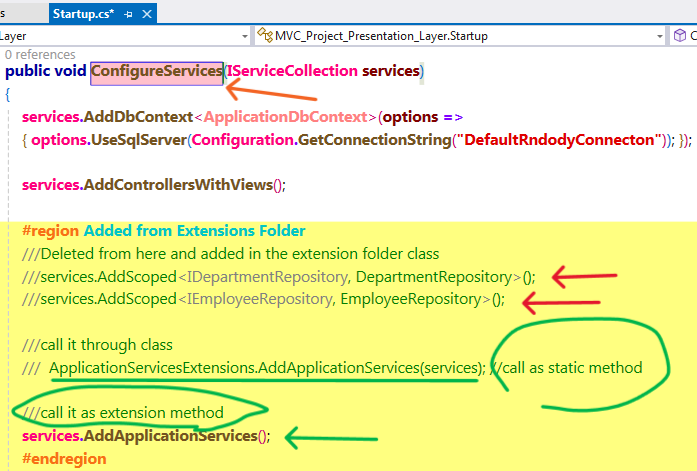
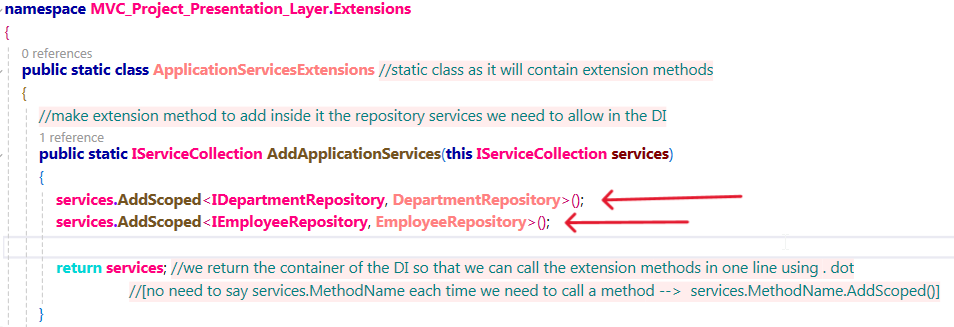
Caching 🡺 is class that has 2 functions [Get-cached-response & cache-response] 🡺example: user send request to see all the users, CLR created us object from class caching services use this object and look for the response of my request in the cache [the cache is the memory of the server] if the found the response is cached, no need to execute LINQ queries to get the data as it will get the data from the cached memory.in case the data we need is not in cached memory so the caching service will get the data and cache it 🡺 so that’s why we need the caching service to be with user till the application terminated

Also logging service [that log exceptions] has singleton lifetime as we need it to be available to log any exception in any request as long as the application is not terminated

* Extension Folder:

We need to group the custom-user-defined services that we need to allow in the dependency injection container in the ConfigureServices method in the Startup class

Create extension folder in the PL project layer that will contain all classes that will contain extension methods

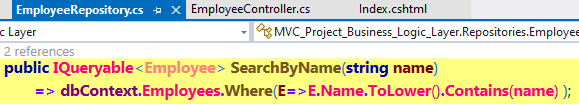


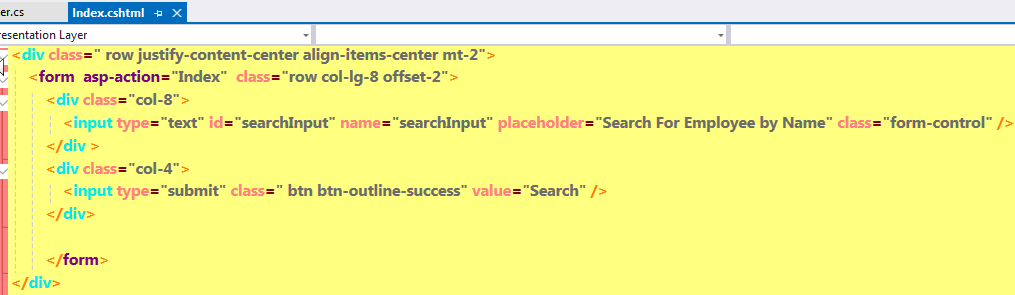
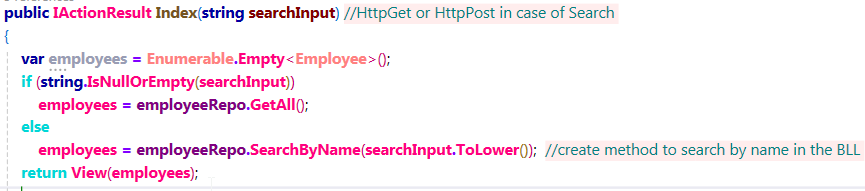
**Search:**

In the view Index in the EmployeeController we need to make search inputs [text and submit]

In the Index action we’ll check if the input value is not null or empty get the employee name

We don’t have a search by name method🡺 create one in the BLL in the EmployeeRepository [remember to add signature for that method in the I EmployeeRepository interface]





**Mapping:**

Mapping is to map from Model [the class represents the data in database] to View-Model [the class represents the data that will be rendered in the View for end-user] and vice-versa

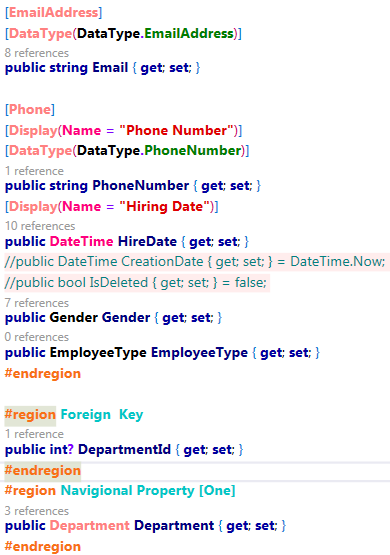
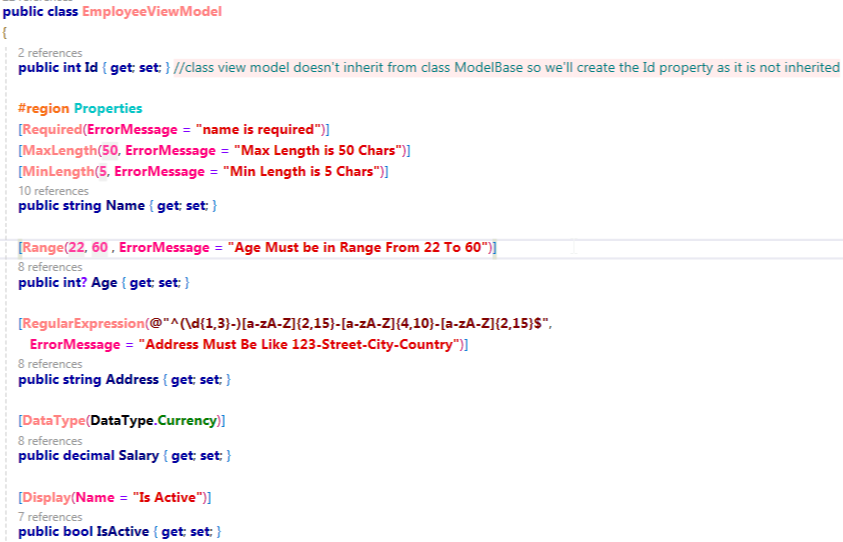
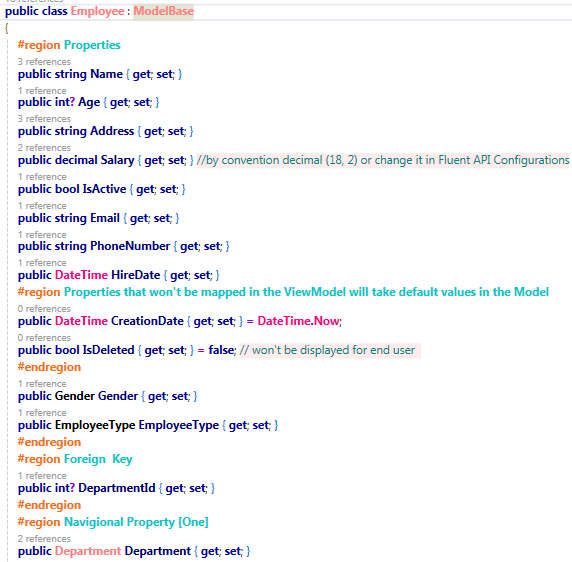
In case of get [index/details] we map from model to view model

In case of post [create/ update/ delete] we map from view model to model

Now in the views we bind on ViewModel Class not the Model so we go and make all views bind on the ViewModel class 🡺

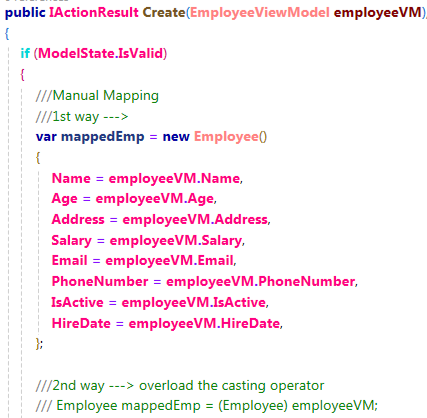


Model class Vs ViewModel class 🡺



Now the classes and views are ready … go and implement the mapping in the actions so that we get the data from the database Model and view the data in the ViewModel for the user, get the data from the user in the ViewModel and send it to database in the Model

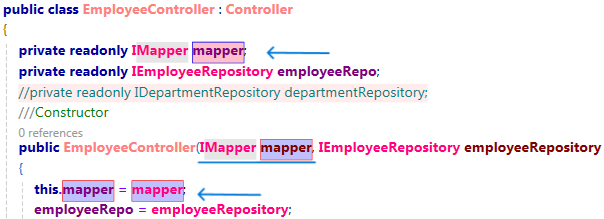
We have 2 manual mapping ways 🡺



Or use the auto mapper package instead on manual mapping🡺 install the package in the PL



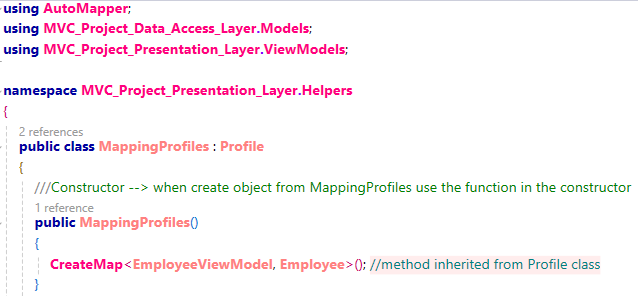
In the constructor of the Employee controller inject object from class implements interface IMapper and ctrl +. Create and assign field [remember using namespace ]



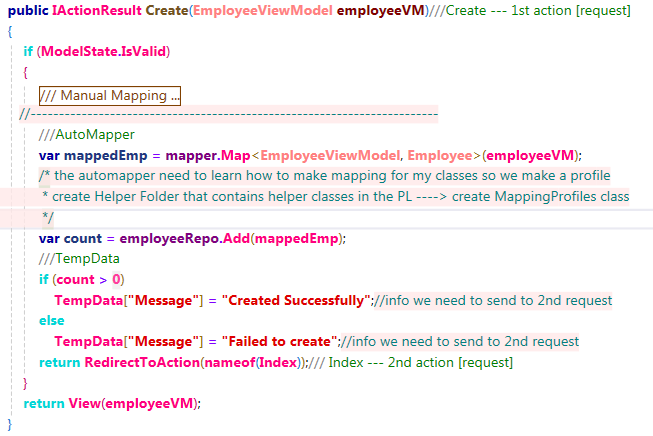
Allow dependency injection in the Startup class 🡺



The MappingProfiles Class in the Helper Folder🡺



Create Action in the EmployeeController🡺



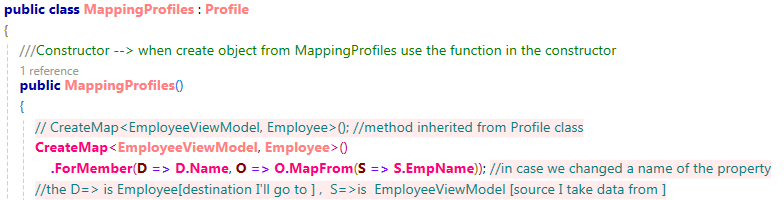
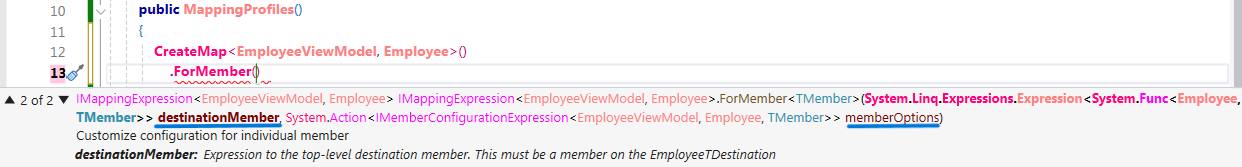
* Summary for the auto-mapper lifecycle:

Creating object from class EmployeeController depends on creating object from class implements IMapper 🡺CLR will create object from class implements IMapper [allow the DI using the AddAutoMapper method ] 🡺 when creating object from class implements IMapper it will add profile for this mapper this profile is our class MappingProfiles, add out profile by adding object from it 🡺 creating object from MappingProfiles will execute the method inside the constructor that creates map between EmployeeViewModel and Employee [this method do basic mapping, it maps each property in the class with a property in the other class having the same name] 🡺 after creating object from MappingProfiles it will be add to our Mapper and inject it in the EmployeeController Constructor 🡺 then we use it in the Create action method and send the objects we need to map using the Map method



Note: The AddAutoMapper lifetime is transient as it only needs to do mapping only one time per request

In case we have a property in the 2 classes we need to map don’t have the same name, we have to configure it 🡺

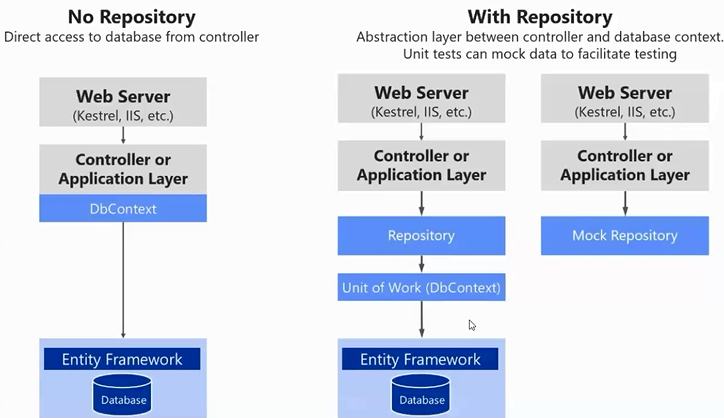


Unit of Work Design Pattern:

Imagine: If we are in the EmployeeController and after creating new employee we needed to update department the delete project 🡺 we’ll face problem of creating objects from Department and Project Models in the EmployeeController Constructor [remember we faced this problem before and solved it by injecting the DepartmentRepository in the View uses it not the Constructor] but now we need those objects in the Create action itself so the problem is to inject objects in the constructor we won’t use them in all actions

Even if we worked that way and created objects that won’t be used in all actions, we’ll face another problem 🡺 we’ll save changes [send request to database] more than one time [3 times] 🡺 the add & update & delete actions use the SaveChanges method which violates the idea of the DbContext class that we save changes one time

We may solve the other problem [SaveChanges more than one time] by removing it from the 3 methods in the GenericRepository class and save changes once in the Create action 🡺 save changes will need an object from DbContext 🡺 in the EmployeeController class we don’t have that object as we are working with the GenericRepository 🡺 deal with DbContext through repository



So now we have to problems🡺 if we need specific service in an action we have to inject it in the constructor 🡺 creating un needed objects in case of executing the other actions. The other problem even if we inject that un-needed object we’ll save changes many times

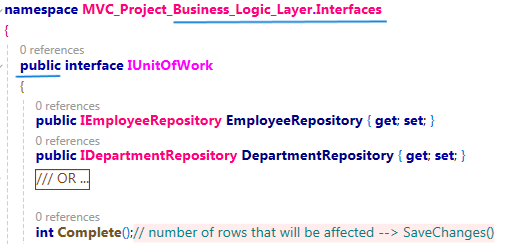
To solve those problems, we are going to use UnitOfWork design pattern

The UnitOfWork is a class 🡺 we call it unit as it is the unit of all our business work with database [will be a re-representation for the Repository 🡪 see diagram above]

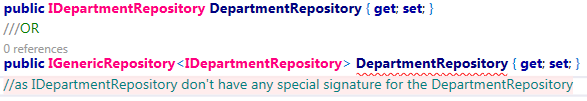
The DbContext have property of type DbSet for each and every table 🡺 the UnitOfWork will have a property of type repository for each and every table

The DbContext have SaveChanges 🡺 UnitOfWork have Complete

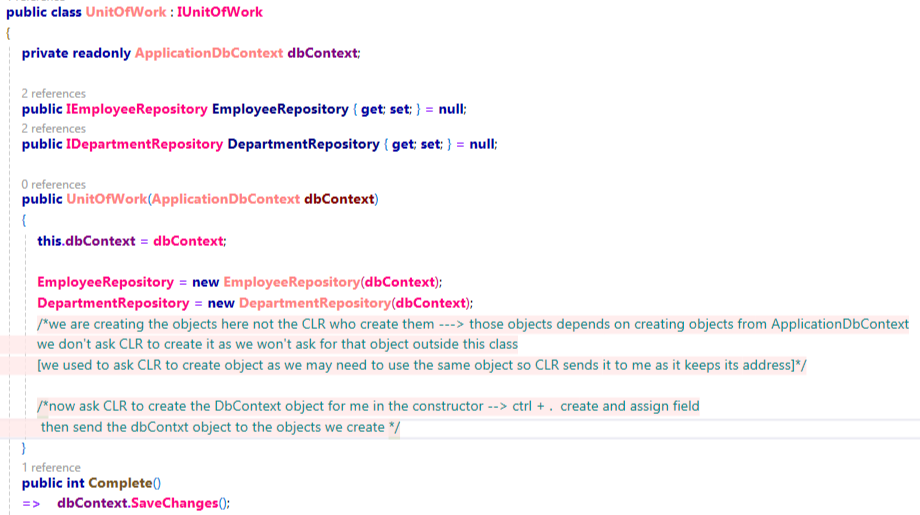
Implementing UnitOfWork in the BBL 🡺 1st make interface as we may have more than one UnitOfWork class in case we have more than one DbContext 🡺 in case we have more than one database





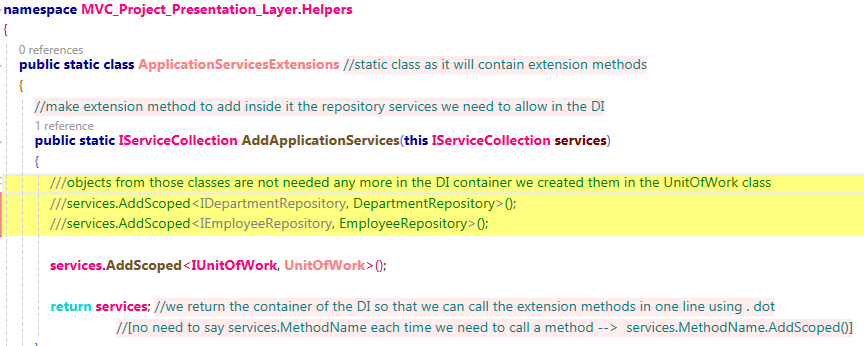
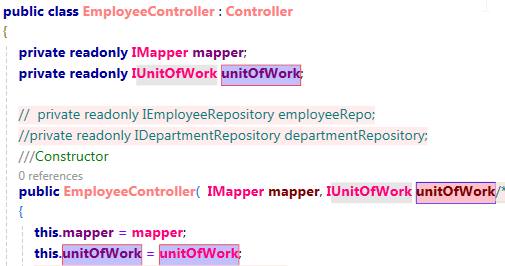


Create class UnitOfWork in the BLL directly or in the folder of the Repositories



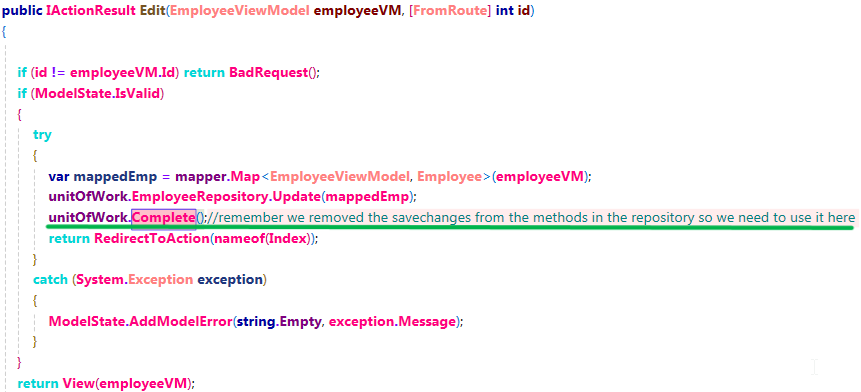
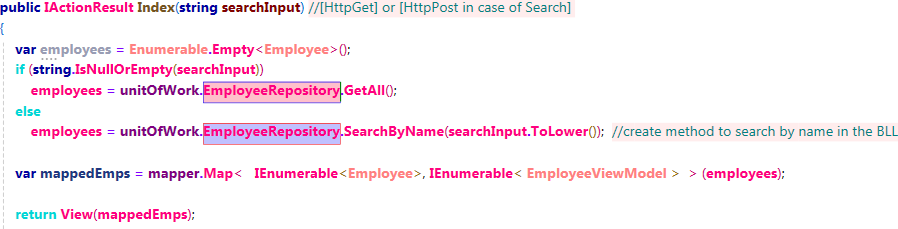
In the EmployeeController class we won’t need to inject objects from IEmployeeRepository, or IDepartmentRepository

🡺 now we need to allow the dependency injection for the classes implements IUnitOfWork 🡺 we’ll add it in the Extension class we created before [ApplicationServicesExtensions]

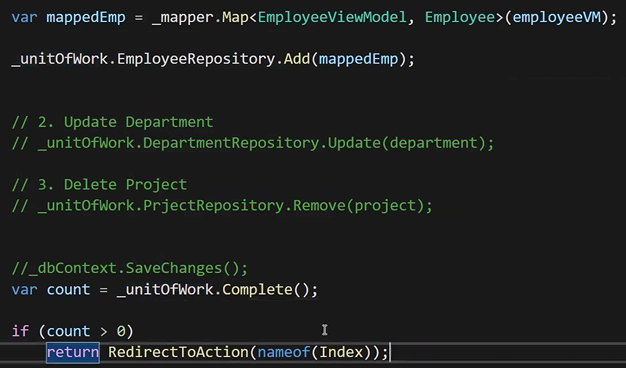


Now use the UnitOfWork object in the EmployeeController

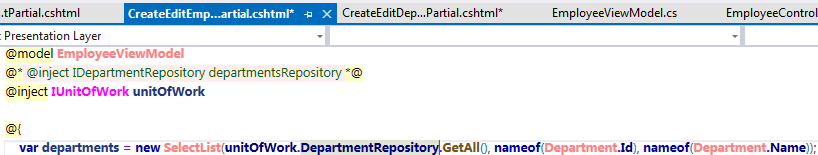
🡺 do that for all other actions



🡸 our draft example of the department and projects we needed an object from them to use in the create action



* Remember in the Employee Views[except for index] we injected the DepartmentRepository, now we don’t need instead we’ll inject object from class implements IUnitOfWork
* At that time we asked 2 times for object from class implements IUnitOfWork one in the constructor and the other in the View needs object from class DepartmentRepository but remember its life time is scoped which means the CLR will keep the object address as long as we are in the same request so now it won’t create new object but will use the object previously created



* we never closed the connection with the database

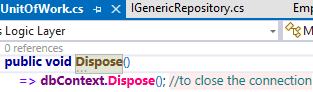
We need to dispose the connection as the connection won’t be closed until the one who opens it disposes it .. even when the lifetime of the DbContext object ends the connection still open [at that time the object is still in the heap, we only lost its address in the stack]

The class UnitOfWork asked for creating object from class ApplicationDbContext so UnitOfWork class must close that connection

If we need to close the connection the class must have a method called Dispose [add a NEW behavior for the class]… how to make sure that the class has this method[behavior]? By signing the contract [implementing interface]

Any class implements the interface IDisposable grantee that it contains the method closes the connection … UnitOfWork must implement that interface to close the connection

So we’re going to make the IUnitOfWork interface inherits the IDisposable interface 🡺 class UnitOfWork implements the IUnitOfWork that have Dispose method that it inherited from IDisposable interface



* Our implementation for the UnitOfWork design pattern is not ideal …it needs some enhancements 🡺

The constructor of the UnitOfWork class doesn’t create any repository object … we’ll create objects if requested as if we create in the constructor, any time I need object from class UnitOfWork it will create objects from all repositories in the constructor 🡺 make construct for DbContext object only

Create method that will create us the object needed on-demand 🡺

