#### **What is Repository Pattern**

Repository Pattern is an abstraction of the Data Access Layer. It hides the details of how exactly the data is saved or retrieved from the underlying data source.

وملخص الفكرة هنا ان هذا النمط هو نمط تجريدي لطبقة الوصول الى البيانات .. أي انه يطبق احدى مبادئ opp والبرمجة عموما والذي يقتضي باخفاء التفاصيل غير المهمة للمستخدم بمعنى ان جوانب مثل كيف تتم جلب البيانات او كيف يتم حفظها هذه أمور توضح ضمن ريبوستوريس مخفية ونتسخدم بعد ذلك دالة مجرد تنفذ هذه المهمة .

وعمليا فإن انشاء ريبو لاي كيان مثل العملاء سيتضمن ملفين ملف عبارة عن انترفيس عبارة عن عقد توجيه عام يضم التعريف للعمليات التي يمكن ان تنفذ في هذ الريبو .. أي يجيب عن السؤال ماذا يمكن ان يتم وليس كيف , لان الكيفية والتفاصيل هي مهمة الكلاس الذي يطبق او ينفذ هذه الانترفيس .  
الانترفيس تضم التعريف العام فقط أي ماهي العمليات المدعومة في الريبو .. إضافة الى تحديد البرميترات التي تستقبلها أي دوال بدون جسم للدالة وسنفهم الفكرة بالتفصيل بالامثلة العملية .

وهذا مثال عام لعمل انترفيس لريبو واسع بالموظفين واخر بوظايف اقل للاقسام

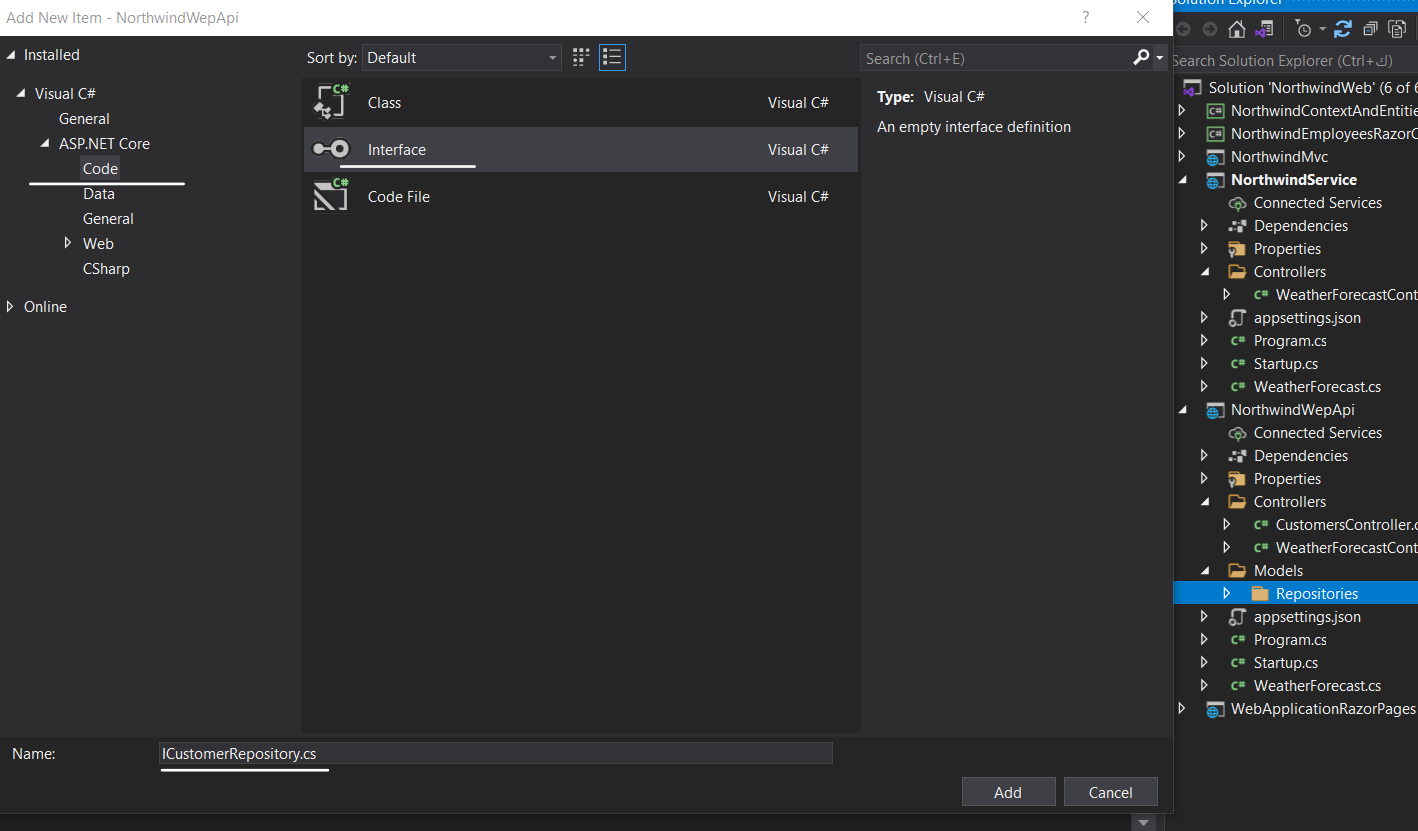
|  |
| --- |
| public interface IEmployeeRepository  {  Task<IEnumerable<Employee>> GetEmployees();  Task<Employee> GetEmployee(int employeeId);  Task<Employee> AddEmployee(Employee employee);  Task<Employee> UpdateEmployee(Employee employee);  Task DeleteEmployee(int employeeId);  Task<IEnumerable<Employee>> Search(string name, Gender? gender);  Task<Employee> GetEmployeeByEmail(string email);  } |
| public interface IDepartmentRepository  {  Task<IEnumerable<Department>> GetDepartments();  Task<Department> GetDepartment(int departmentId);  } |

وهنا لو ركزنا فهو مستوى عام يحدد المهام ويخبرنا بماذا وليس كيف ونحدد كذلك البرميترات التي ممكن ان تستقبلها الدالة  
علما ان العمليات الأساسية عادة تكون هي CRUD

• C for Create  
• R for Retrieve (or Read)  
• U for Update  
• D for Delete

لكن ممكن ان نضمن فيه كذلك عمليات أخرى سنحتاج لاجرائها كالبحث او جلب عنصر بواسطة حق معين او عملية البجنيشن " الصفحات " وما الى ذلك من الأمور التي سوف نعرفها بالاخير نحدد ما نريد سواء اكثر او اقل .

وبالعودة الى مشروعنا الخاص ب Northwind database وقبل التطبيق العملي للكود سنعمل أولا في مجلد اسمه Models ثم مجلد داخله واسمه Repositories وداخله ننشى أولا ملف انترفيس ونسميه ICustomerRepository.cs على النحو التالي



ثم داخل ICustomerRepository.cs نكتب الكود ( وملاحظة انه ولاننا نعمل على تطبيقات حديثة فسنتخدم ال task من اجل لاحقا استخدام التنفيذ غير المتزامن من اجل تحسين الأداء )

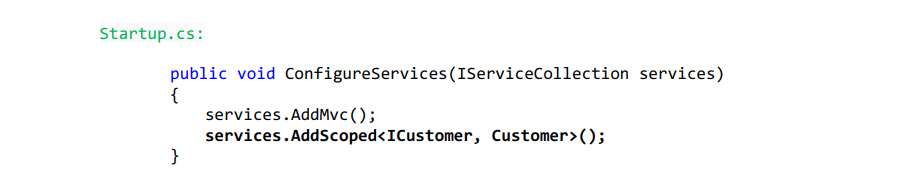
|  |
| --- |
| public interface ICustomerRepository { Task<IEnumerable<Customer>> RetrieveAllAsync(); Task<Customer> RetrieveAsync(string id);  Task<Customer> CreateAsync(Customer c); Task<Customer> UpdateAsync(string id, Customer c); Task<bool?> DeleteAsync(string id); } |

تسمية الدوال والقيم المرتجعة و البرميترات المستقبلة نسميها على كيفنا   
الان الخطوة الثانية هي انشاء كلاس ونسميه مثلا CustomerRepository ينفذ هذه الانترفيس أي نعمل له امبلمنتيشن بهذه الدوال ومحتواها وهناك سنكتب الbody حق هذه الدوال

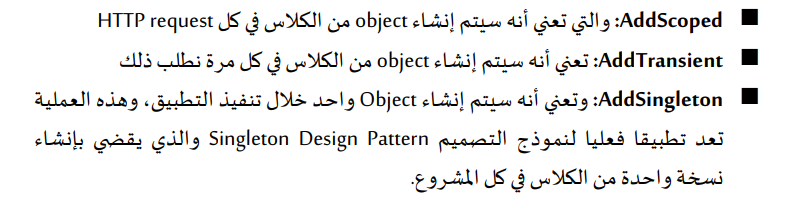
|  |
| --- |
| public class CustomerRepository : ICustomerRepository {} |

الان في كلاس الريبو CustomerRepository نبدا نعلم تنفيذ دوال الانترفيس ICustomerRepository   
أي نعمل لها الامبلمنتشن كما تعلمنا في EF وكل دالة بما يناسبها من كود حيث نعرف في كل ريبو مثيل مثل من الداتا كونتكست context أي نعمل لها انجكشن حقن ونستخدمها في الوصول الى الكيان الخاص بال Customer و نعمل فيها الاكشن اللي نريده .

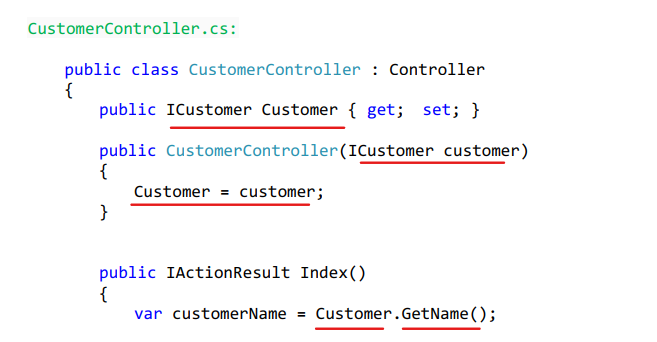
الان علشان نقدر نستخدم هذا الريبو ودواله لازم نعمل له تسجيل في ملف startUp



طبعا هنا ثلاث دوال لتسجيل الريبو :



الان بعد تسجيل الريبو في كسيرفس في كلاس startUp وعلشان نستخدم دواله في أي كونترولر نقوم أولا بعمل انجكيشن له في الكونترولر .

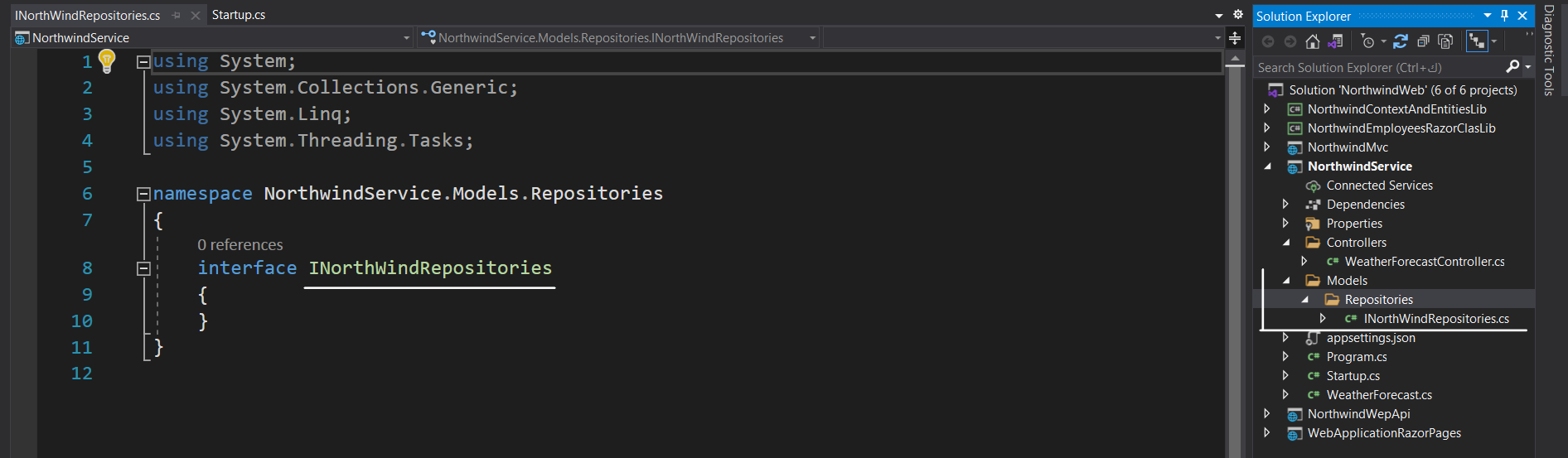


مشكلة  
 نتخيل ان لدينا عشرة كيانات ... عملاء وموظفين واقسام ومنتجات وموردين وجميعها تحتاج لهذه العمليات المدونة في الانترفيس ICustomerRepository .

هذا يعني انه يجب ان تكون لدينا عشرة انترفيسات وكل انترفيس يتم تنفيذها من الكلاس الذي يناسبها بالكيان فمثلا يكون لدينا انترفيس باسم IEmployeeRepository وينفذها الكلاس EmployeeRepositoryوهكذا .  
وهذه المشكلة سنحلها ب generic بحيث نكتب انترفيس واحدة وتكون مرنه تورثها اكثر من كلاس " ريبو "

لذلك مع استخدام ال generic فبامكاننا ان نكتفي بكتابة انترفيس واحدة وتكون مرنه بحسب كل كلاس يعمل لها تنفيذه حيث تأخذ النوع بواسطة القوسين <> .  
وبالتالي فنحن بحاجة الى اسم انترفيس عامة ك BaseRepositories.cs ولكن يفضل تكون بحسب اسم النظام ككل فمثلا بنظامنا هذا نسميها INorthWindRepositories.cs .

لذلك بمجلد ال Repositories الذي داخل Models نحذف الانترفيس ICustomerRepository ونظيف أخرى باسم INorthWindRepositories .



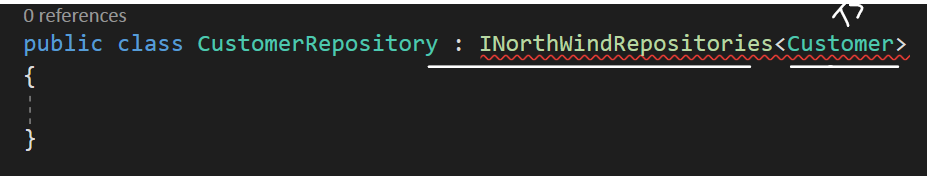
وبما انه سنتخدم إمكانيات الجينريك فيصبح كود الكلاس على النحو التالي  
حيث ببسساطة اضفنا <T> بعد اسم الكلاس وازلنا اسم الكيان المحدد لنوع البيانات وخليناه جينرك T .

|  |
| --- |
| interface INorthWindRepositories<T>  {  Task<IEnumerable<T>> RetrieveAllAsync();  Task<T> RetrieveAsync(string id);  Task<T> CreateAsync(T c);  Task<T> UpdateAsync(string id, T c);  Task<bool?> DeleteAsync(string id);  } |

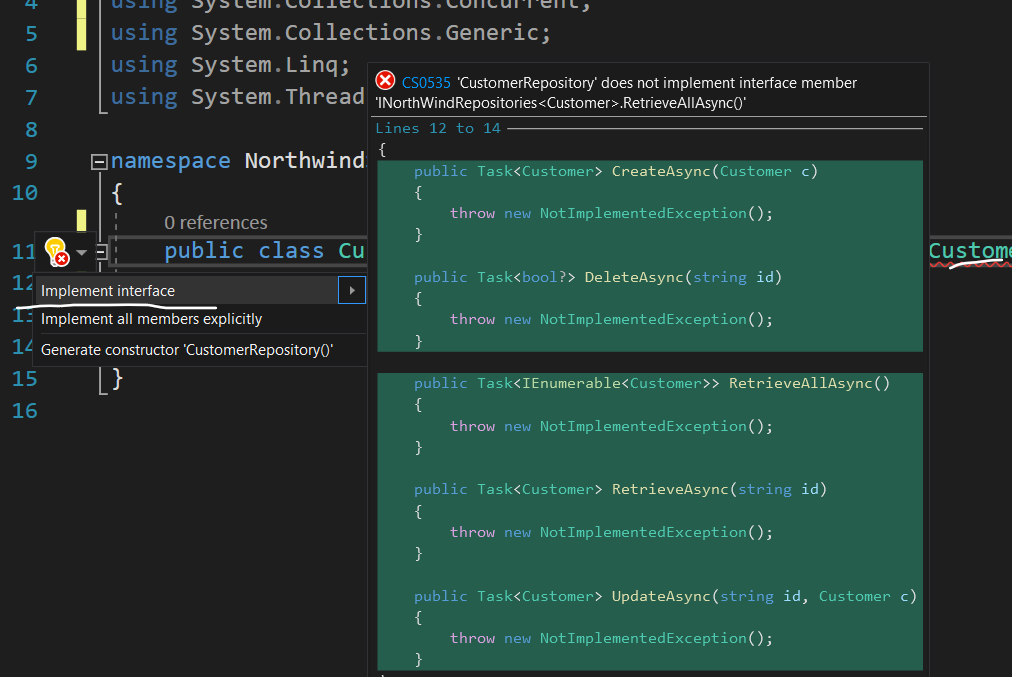
الان هذه الانترفيس نقدر نعمل لها امبلمنتشن من كلاس أي كيان حيث نمرر بدل T اسم الكيان مثل Customer او Employee. كما سنتعلم بعد قليل .

الان جاء دور الكلاس الذي سننفذ فيه هذه العمليات والذي سيكون خصيصا للكيان Customer .

ننشى كلاس عادي ويكون اسمه CustomerRepository.cs . حيث سنجعله يورث من الانترفيس أي يعمل لها امبلمنتشين وبالتالي كتابة الدوال فيها كامله مع body .

هنا عملنا وراضة ومررنا اسم الكيان Customer بين القوسين <T> ليحل الاسم مكان كل T في الانترفيس .

الان لو نلاحظ انه لدينا خط الخطا الأحمر وهذا يعني انه لم ننفذ الدوال التي في الانتفريس بعد فلو نقرنا باليمين على اسم الانترفيس واخترنا



فسنجد كود بهيكل الدور تم ادراجه مباشره



هنا تم إضافة الدوال كامل الموجود تعريفهن في الانترفيس وبنفس الهيكل من حيث نوعية القيم المرتجعة و من ناحية البرميترات المستقبلة. مع ملاجظة انه بالنسبة لما بين قوسي body موجود عبارة throw new NotImplementedException وهذا يعني انه لم يتم عمل كود لها وهذا مفيد في حالة انه لدينا انترفيس جينرك وتحتوي على تعريف وبنفس الوقت عملنا لها امبلمنت مع كيان لا يحتاج لتلك الدالة فهنا نخليها بنفس المحتوى المدرج تلقائيا لانه ما ينفع ما نعملها خالص سيظهر خطا فالانترفيس بمثابة عقد يجب ان ننفذ مافيه كله .

ثم كل دالة من الدوال في الريبو نعمل لها الكود الخاص والمناسب كما تعلمنا في EF   
وقبل ان نستخدم هذا الريبو ودوله في الكونترولر الخاص بالعملاء فانه يجب ان نعمل الخطوتين اللي نعرفها والأولى هي عمل تسجيل لهذا الكلاس والانترفيس في دالة السيرفس ب كلاس startUp حقه

|  |
| --- |
| services.AddScoped < INorthWindRepositories<Customer>, CustomerRepository>(); |

والخطوة الثانية هي حقن لسيرفس الريبو في الكونترولر علشان نقدر نستخدم دواله فيه

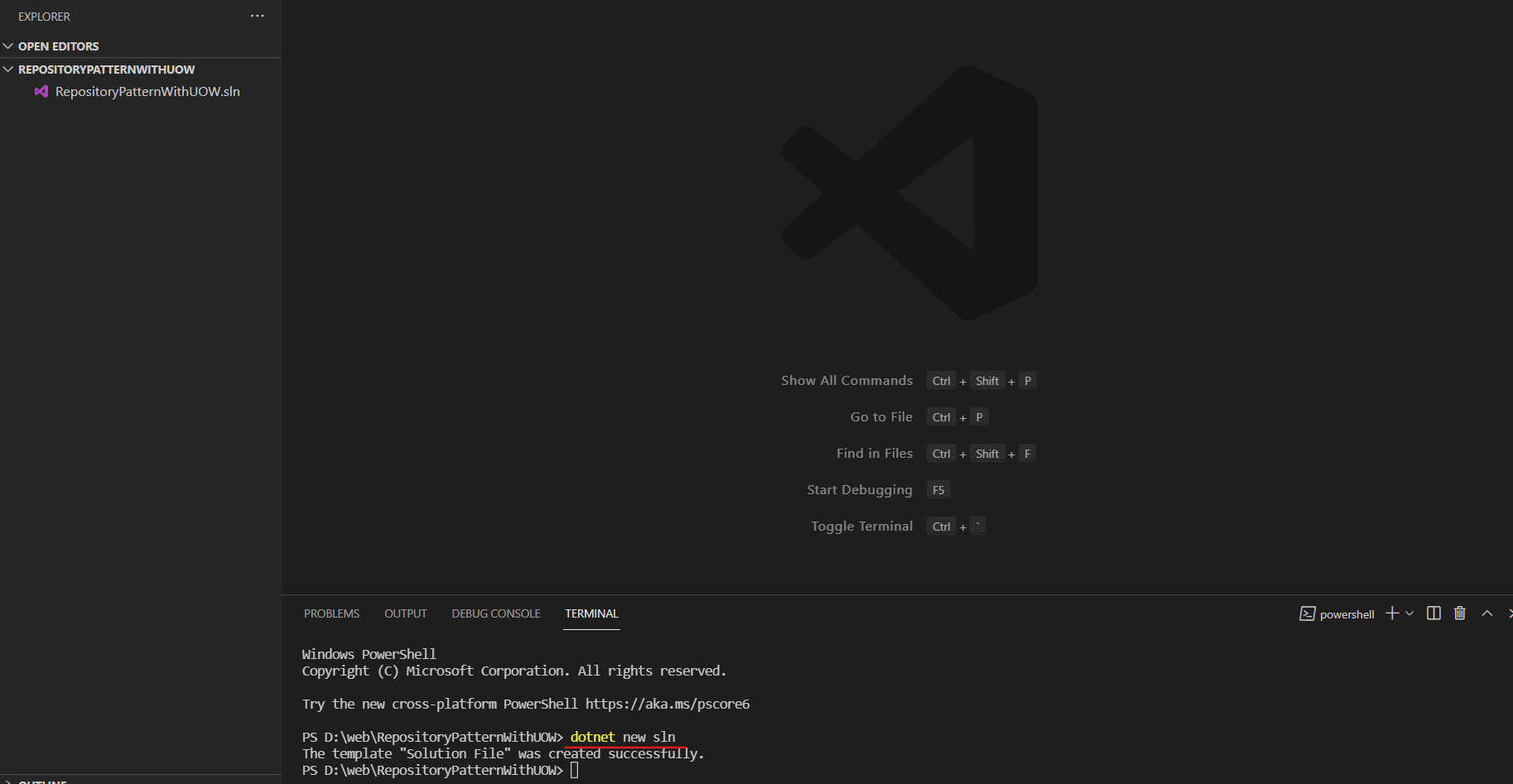
|  |
| --- |
| private readonly INorthWindRepositories<Customer> repo;  // constructor injects repository registered in Startup  public CustomersController(INorthWindRepositories<Customer> repo)  {  this.repo = repo;  } |

نلاحظ هنا انه حددا سواء في عملية التسجيل او الحقن نوع الجنرك تايب بين القوسين <> .

الى هنا وعلى هذا الأساس نكون قد اختصرنا كثير من الكلاسات وخلينا كل الريبوستورس تورث من انترفيس واحده.

ولكن ماذا لو اردنا ان نحسن الكود ونختصر الامر اكثر ونجعل الانترفيس والريبو اللي يورثها كلاهما generic ونخلي التحديد للكيانات فقط داخل الكونترولرز التي تستخدم دوال الريبستوريس .

تجهيز المشروع  
ننشى مجلد باسم RepositoryPatternWithUOW ثم نفتحه بالفيجوال ستيديو كود  
بالترمنل حقه ننشى أولا C# solution بالامر الموضح بالصورة



الان مطلوب ننشى مشروع من نوع webApi داخل مجلد Api ونضمنه داخل السلويشن المنشئ كمان .

نقدر ننفذ هذا بامرين   
الامر الأولى ننشى مشروع من نوع webapi ثم نضيفه لمجلد نسميه API .

dotnet new webapi -o API

The -o parameter lets you specify the output directory

ثم بالامر الثاني نضيف المشروع الى السوليشن بالامر

dotnet sln add API

الان نحن سنكتفي بهذا المشروع وفيه سنضمن كل الكلاسات بالمشروع  
ولكن لو حبينا من باب التنظيم فممكن نضيف اثنين كلاس ليبرري أي مكتبتين كلاسات ونضيفهم الى السلويشن

dotnet new classlib -o EF

dotnet sln add EF

بمكتبة الكلاسات هذه سنضيف ملف الداتا كونتكست ومجلد المجريشن وكذلك الريبوستوريس ( اللي تعمل امبلمنتيشن للانترفيسس )

dotnet new classlib -o CORE

dotnet sln add CORE

هنا نحط الانترفيسس اللي سيعمل لها الريبو املبنتتشن كمان سنعمل Models اللي ستمثل جداول او كيانات الداتا بيز وممكن نضيف DTSs هنا او أي شيء شيرد

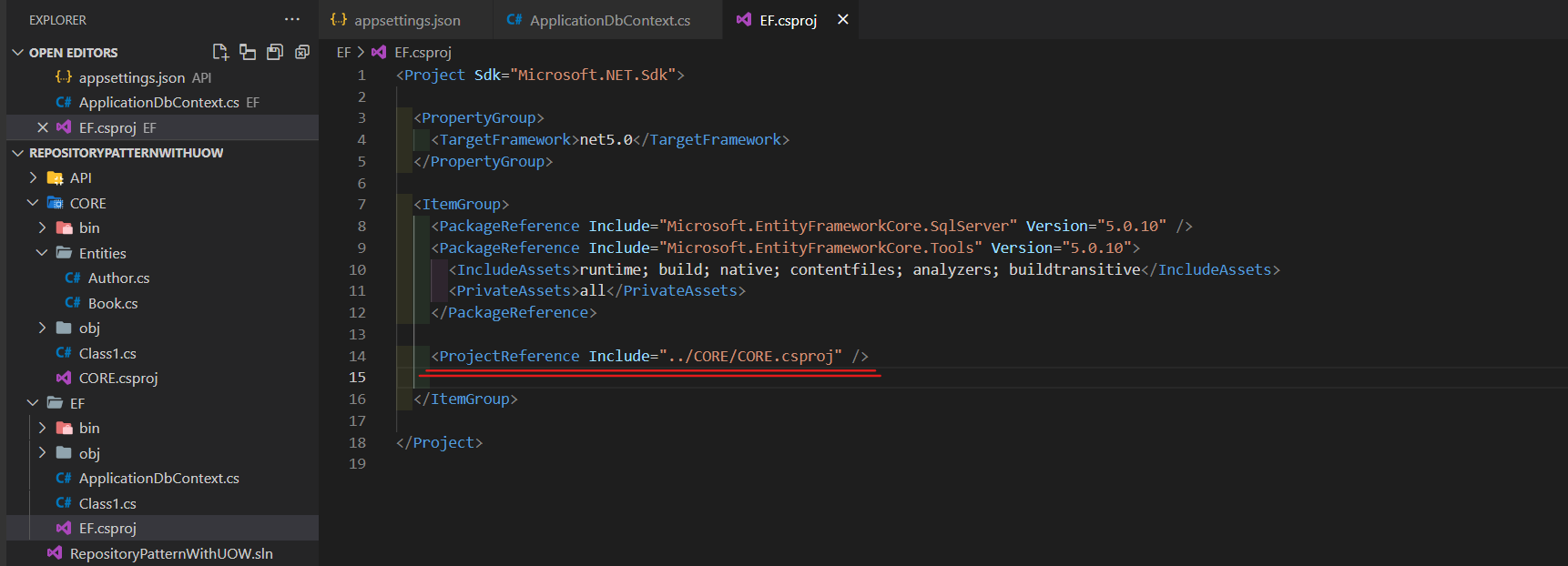
وهذا التقسيم مجرد للتنظيم ولكنه ليس قانون ونقدر نعمل التنظيم اللي نريده بالاخير ولكن هذا علشان نعرف كيف نقدر نشتغل على مشروع ملفاته متقسمه بين مشاريع ومكتبات كلاسس مختلفة .

طبعا الكلاسات موضوع بين المشروع والمكتببتين ولكنها مرتبطات ببعض لذلك لو اردنا نصل لكلاس موجود بمكتبة أخرى سنحتاج لعمل ريفرنس له بالامر

dotnet add EF/EF.csproj reference CORE/CORE.csproj

وهنا بايجيب مكتبة CORE ويعملها لها ريفرنس بمكتبة EF

او يدويا ع النحو التالي



بكل الأحوال نحن لن نحتاج لهذا لانه نعمل بشكل تعليمي فقط  
لذلك سنشتغل كل الكلاسات بمشروع ال wepApi   
و سنبدا نثبت الحزم المطلوبة للاتنتي فريمورك لانه سنتعامل مع جداول حقيقة وبيانات اثناء عملية بناء الريبوستوريس ودوالها .

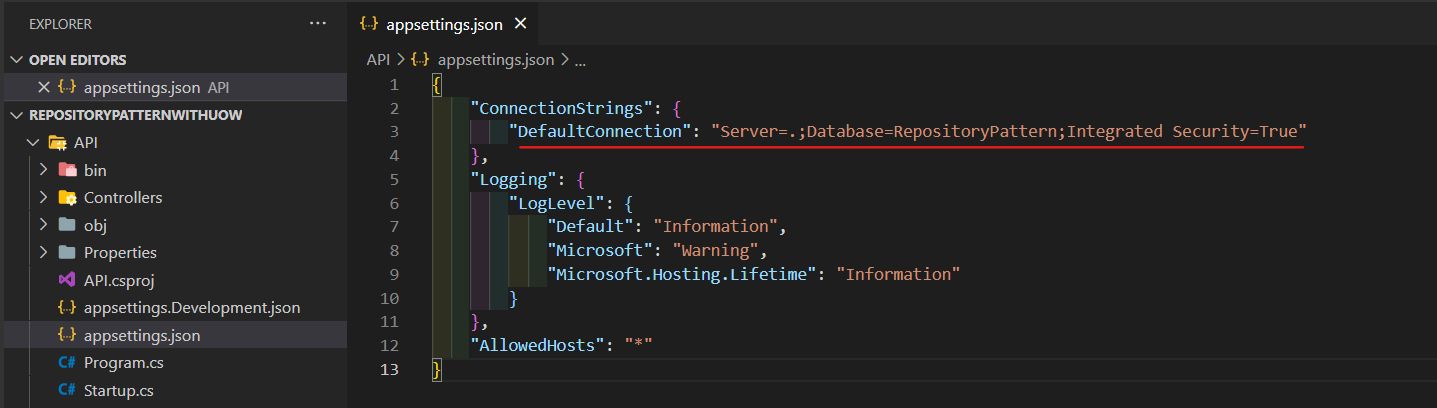
سنحتاج لتثبيت هذه الحزمتين في المشروع

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

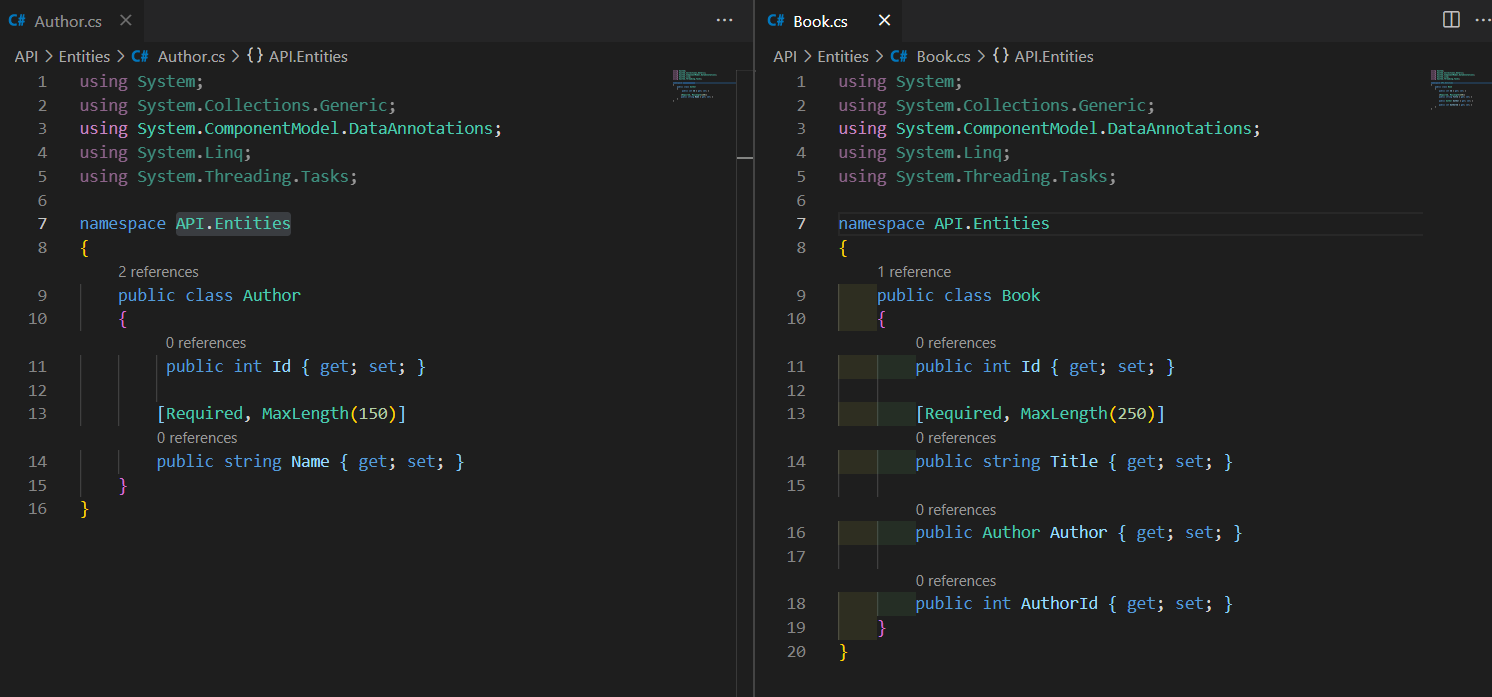
dotnet add package Microsoft.EntityFrameworkCore.Tools

// dotnet add package Microsoft.EntityFrameworkCore.Design // لا نحتاج له

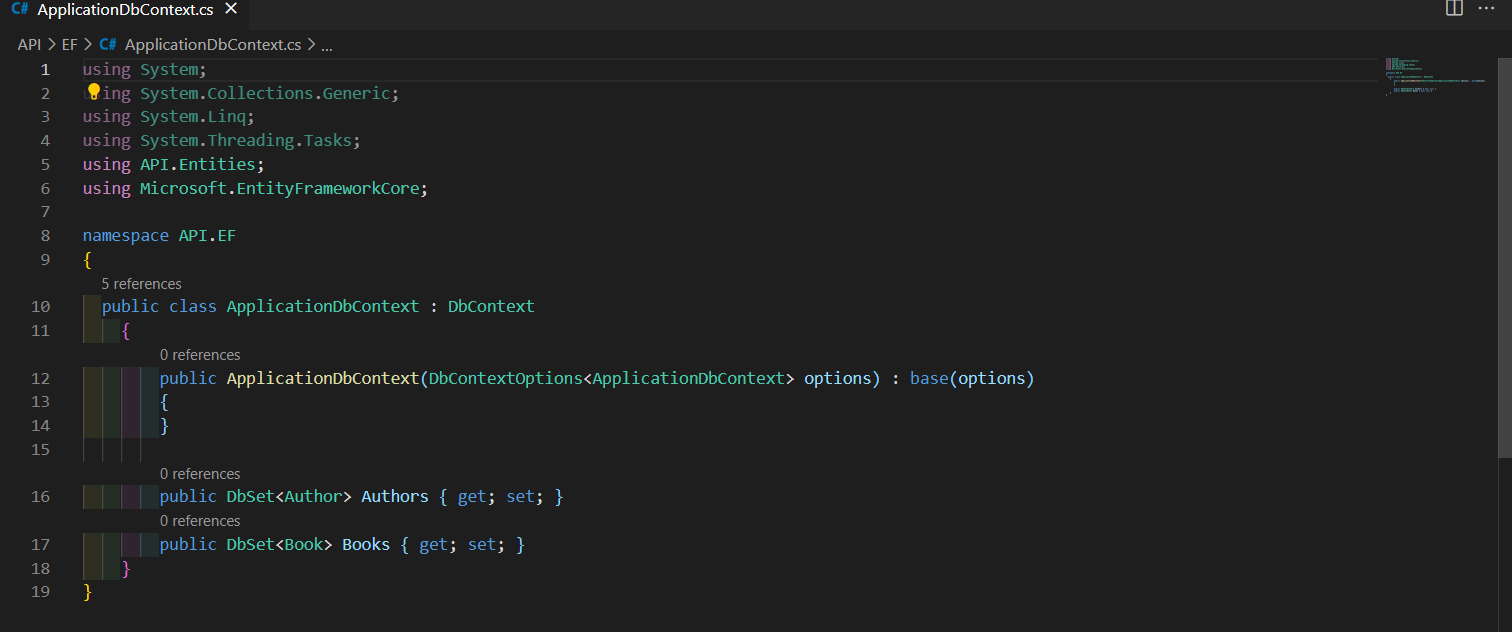
الان بملف appsettings.json نضيف سلسلة الاتصال على النحو التالي



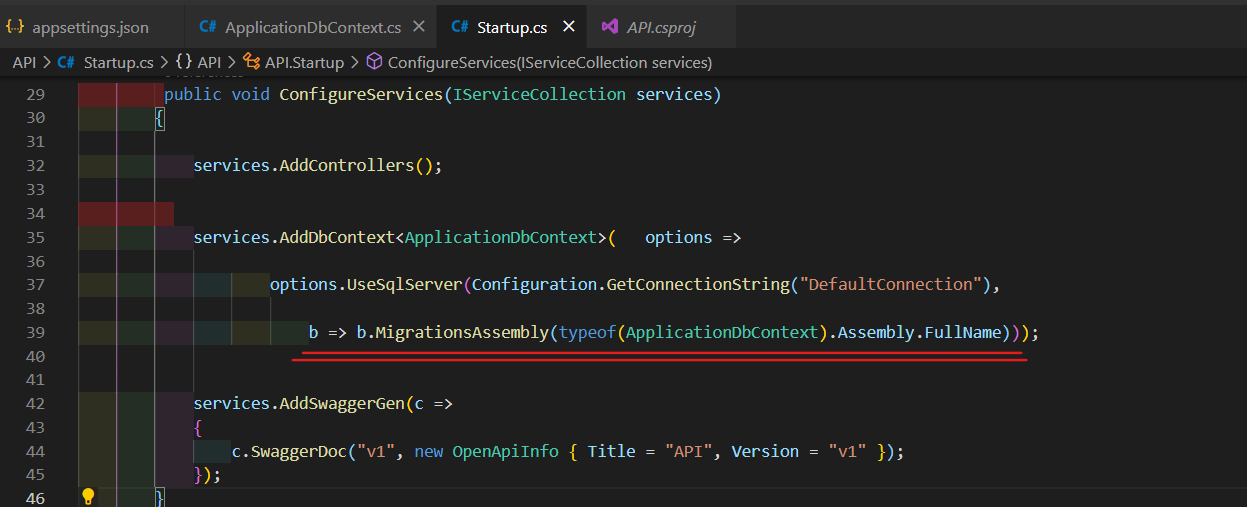
سننشى الان في مكتبة مجلد باسم Entitiesفيه كلاسين يمثلان الكيانين في الداتا بيز



الان بمجلد EF سنعمل كلاس يمثل الداتا كونتكست أي الداتا بيز ككل .



الان سنقوم بتسجيل الداتا كونتكست في كلاس startup



المؤشر بالاحمر مش مهم الا لو لو كان ملف الداتا كونتكست في مكتبة كلاسات منفصلة  
الان ننشى اول عملية ترحيل ثم نضيفها للداتا بيز بالامرين

dotnet ef migrations add InitialCreate

ثم

dotnet ef database update

نصيف بشكل يدوي مجموعة من الحقول بجدولي المؤلفين والكتب .

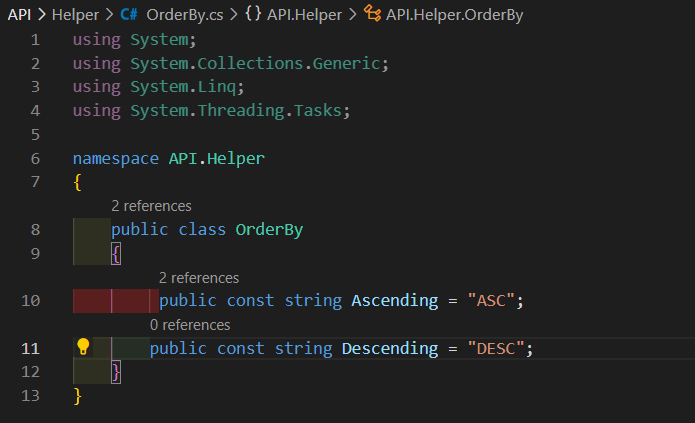
الان سنبدا باعداد الكود فعليا وسننشى أولا مجلد باسم Interfaces سيضم هنا الانترفيس اللي سيعمل لها الريبو بعد ذلك الامبلنتشن لدوالها   
وهذه ستكون انترفيس جنرك وسنحاول ان نلم فيها كل الدوال اللي ممكن يتم الاحتياج لها في جلب البيانات وتمرير البرميترات المختلفة لها , او عمليات الإضافة والتعديل والحذف .. وسنقوم بتعريف دول بالانترفيس سواء عادية وبنفس الوقت سنعرفها ك asynic

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Linq.Expressions;  using System.Threading.Tasks;  using API.Helper;  namespace API.Interfaces  {      // هنا اعطينا للانترفيس اسم عام وجلعناه جنرك وحددنا ان التي عبارة عن كلاس      public interface IBaseRepository<T> where T : class      {  /\*   هنا عرفنا هيدر دالة عادية لجلب سجل بواسطة معرفه الرقمي ثم عرفنا نفس الدالة ولكن متزامنة   وهنا سنحتاج للمعرف فقط لانه سنستخدم دالة  find   التي لاتحتاج الا للاي دي  \*/           T GetById(int id);          Task<T> GetByIdAsync(int id);  // هنا عرفنا هيدر دالة عادية لجلب كل السجلات  ثم عرفنا نفس الدالة ولكن متزامنة          IEnumerable<T> GetAll();          Task<IEnumerable<T>> GetAllAsync();  /\*  بالتالي عرفنا دالة تعيد لنا عنصر واحد بناء على شرط معين "تعبير" نمرره هنا وسميناه هنا  criteria   او ممكن نسميه   match  اضافة الى برميتر اخر في حالة انه نريد السجلات اللي ترتبد فيه بالجدول الاخر اي  includes  وخليناه الافتراضي بنل اي لو ما بعثش  وهنا خلينا كمفصفوفة لانه ممكن يكون انكلود من اكثر من كيان اخر  وهنا حددنا الامر كمتغيرات لانه مش ح نعرف ايش الشرط بايكون اللي على اساسه بانعيد السجل  حيث انه في الكلاس اللي بايعمل امبلمنتيشن لهذه الانترفيس برضه بايكون عام وجنرك  والتحديد للشروط فقط سيكون بالكونترولر كما سنتعلم لاحقا  الدالة الثانية نفس الامر ولكن كدالة متزامنة  \*/          T Find(Expression<Func<T, bool>> criteria, string[] includes = null);          Task<T> FindAsync(Expression<Func<T, bool>> criteria, string[] includes = null);  /\*  التالي عرفنا 3 دوال عادية و 3 دوال متزامنة  والاسماء واحدة ولكن البرميترات مختلفة اي انه طبقنا مبدا الاوفرلودنج  وهنا الدوال تعيد لنا مجموعة من العناصر اي لسته بناء  على شرط معين  نمرره لها  criteria  الدالة الاولى لو اردنا جلب حقول مرتبطة من جدول اخر ب  includes  وهنا مصفوفة ممكن تحوي اكثر من قيمة وكل قيمة تمثل كيان اخر قد يرتبط بالجدول المحدد  دالة اخرى حددنا لها شرط نفلتر على اساسه ثم متغيري تيك وسكب الخاصين بالبنجنيشن  اضافة الى برميتر شرطي يحدد على ضوء اي عمود بانقوم بالترتيب  ثم اتجاه الترتيب هذا هل سيكون تصاعدي او تنازلي وهنا اعتمدنا على قيم ثابته انشئناها بكلاس اخر بمجلد هيلبر من باب التنظيم  والافتراضي هنا جعلناه تصاعدي  \*/          IEnumerable<T> FindAll(Expression<Func<T, bool>> criteria, string[] includes = null);          IEnumerable<T> FindAll(Expression<Func<T, bool>> criteria, int take, int skip);          IEnumerable<T> FindAll(Expression<Func<T, bool>> criteria, int? take, int? skip,              Expression<Func<T, object>> orderBy = null, string orderByDirection = OrderBy.Ascending);          Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> criteria, string[] includes = null);          Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> criteria, int skip, int take);          Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> criteria, int? skip, int? take,              Expression<Func<T, object>> orderBy = null, string orderByDirection = OrderBy.Ascending);  // دالتي اضافة عادية ومتزامنه وتعيدان العنصر اللي انضاف          T Add(T entity);          Task<T> AddAsync(T entity);  // دالتي اضافة لسته عناصر  عادية ومتزامنه وتعيدان العنصر اللي انضاف          IEnumerable<T> AddRange(IEnumerable<T> entities);          Task<IEnumerable<T>> AddRangeAsync(IEnumerable<T> entities);  // دوال لعمل تعديل على عنصر او حذفه او حذف لسته عناصر  // اضافة لدوال تخص الاتاتش          T Update(T entity);          void Delete(T entity);          void DeleteRange(IEnumerable<T> entities);          void Attach(T entity);          void AttachRange(IEnumerable<T> entities);  /\*  دالتين   تعمل احصاء عام لكل السجلات  واحده عادية والثانية متزامنة  يليهما دالتين يعملان احصاء للسجلات بناء على شرط نمرره  criteria  \*/          int Count();          Task<int> CountAsync();          int Count(Expression<Func<T, bool>> criteria);            Task<int> CountAsync(Expression<Func<T, bool>> criteria);      }  } |

بالنسبة للبرميتر

string orderByDirection = OrderBy.Ascending

فهو مرتبط بكلاس اخر عملناه لتحديد قيمتي متغيرين تصاعدي وتنازلي

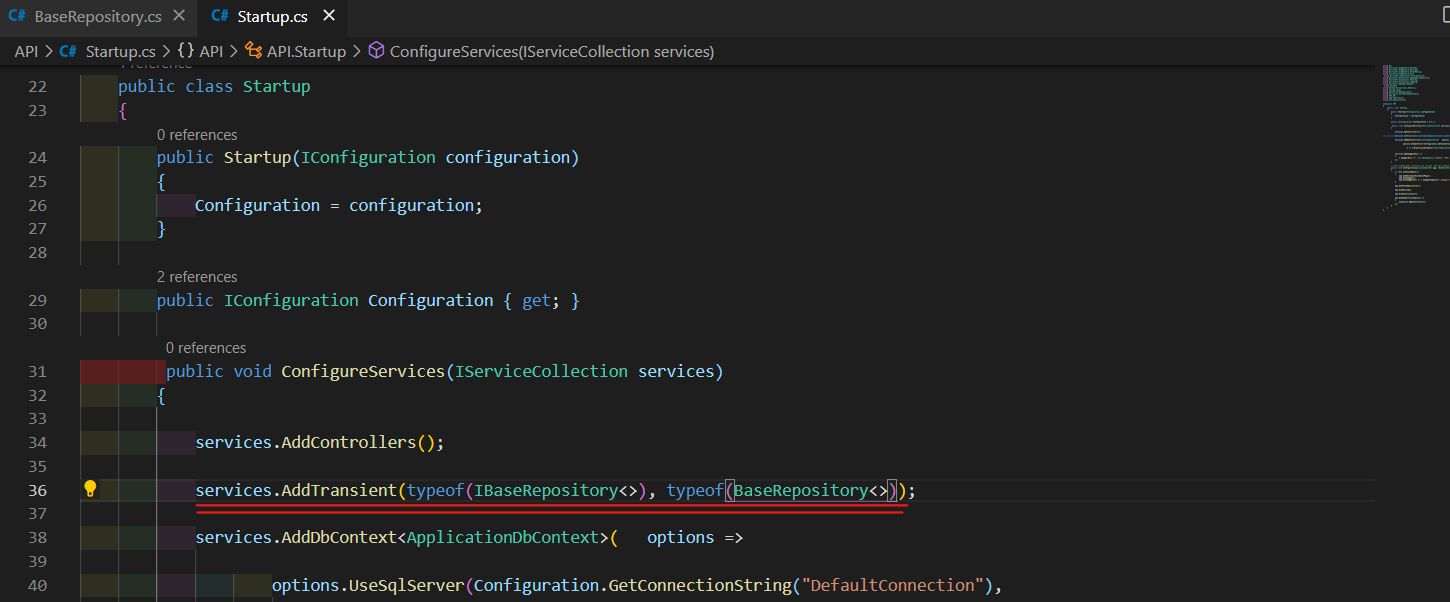


الخطوة التالية هي عمل كلاس يعمل امبلنتيشن لهذه الدوال   
مع ملاحظة ان الكلاس هنا أي الريبو لن يكون مخصص لكيان واحد كالكتاب او المؤلفين وانما هو بدوره سيكون جنيرك أي عام وممكن ينفع مع أي كيان نعمل عليه داخل الكونترولر لاحقا .

لذلك سنعطيه اسم عام BaseRepository.cs علما انه سنضعه داخل مجلد ننشئه باسم Repositories.

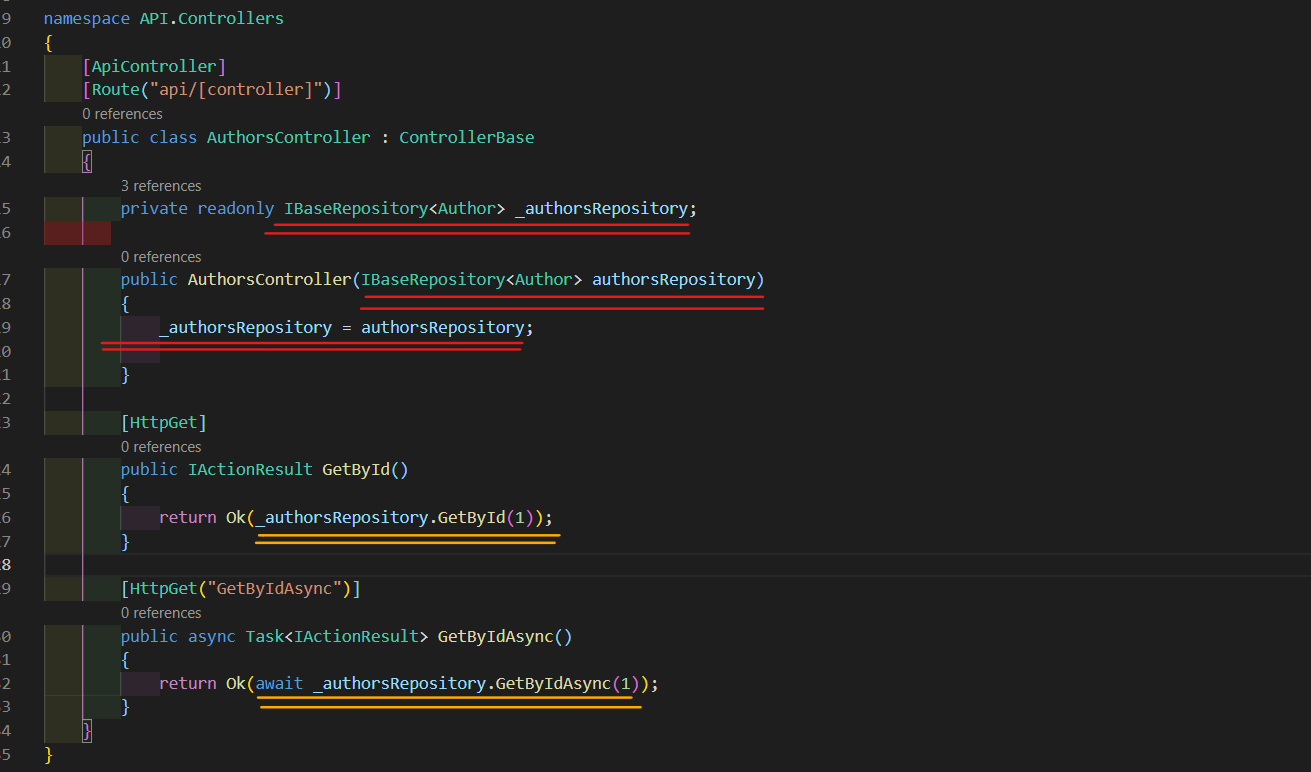
|  |
| --- |
| namespace API.Repositories  {      // هنا الكلاس هذا هو بدوره جنرك ويعمل امبلنتيشن لدوال الانترفيس اللي هي كمان جنرك      // حيث ان قيم متغير التي ستسند بالكونترولر لاحقا     public class BaseRepository<T> : IBaseRepository<T> where T : class      {          // اول خطوة هنا هي عمل حقن للداتا كونتكست لانه سنحتاج له لما نعمل امبلنتيشن للدوال          protected ApplicationDbContext \_context;          public BaseRepository(ApplicationDbContext context)          {              \_context = context;          }  /\*  سنبدا بعمل الامبلمنتيشن لدالتي جلب عنصر بواسطة الاي دي  المشكلة التي ستصادفنا انه لما نستخدم مثيل الكونتكست  عادة يكون بعدها اسم التيبل اي المؤلفين مثلا او الكتب  ولان هذا الريبو جنيرك ينفع مع كل كيان سنسنتخدم دالة  set  اللي باتسند بمكانها نوع الكيان  T  عندما نحدده  \*/          public T GetById(int id)          {              return \_context.Set<T>().Find(id);          }          public async Task<T> GetByIdAsync(int id)          {              return await \_context.Set<T>().FindAsync(id);          }  // هنا نفس الامر ولكن نعيد لسته من العناصر اي كل السجلات          public IEnumerable<T> GetAll()          {              return \_context.Set<T>().ToList();          }          public async Task<IEnumerable<T>> GetAllAsync()          {              return await \_context.Set<T>().ToListAsync();          }  /\*  هنا دوال الفلترة او الدوال اللي تعيد سجل او سجلات بناء على شرط معين  في الدالة الاولى ستستقبل شرط معين نمرره كبرميتر داخل  SingleOrDefault  وهنا سنجل او ديفلوت لانه سيرجع لنا سجل واحد او لا شيء خالص  وبعده لدينا برميتر مصفوفة باسم  includes  يحمل اسماء باسماء الكيانات اللي ممكن يرتبط بها الكيان الحالي  ولانه لانعرف عددها علمنا لووب لتلف عليها  -- ملاحظة  استخدمنا هنا  IQueryable  لانه لسا العمليات جارية على البيانات اي نخزنها ككويري ونضيف لها انكلود ثم سنجل اور ديفولت وهكذا  \*/          public T Find(Expression<Func<T, bool>> criteria, string[] includes = null)          {              IQueryable<T> query = \_context.Set<T>();              if (includes != null)                  foreach (var incluse in includes)                      query = query.Include(incluse);              return query.SingleOrDefault(criteria);          }  // نفس الدالة السابقة ولكن استخدمنا دوال لينكيو متزامنة          public async Task<T> FindAsync(Expression<Func<T, bool>> criteria, string[] includes = null)          {              IQueryable<T> query = \_context.Set<T>();              if (includes != null)                  foreach (var incluse in includes)                      query = query.Include(incluse);              return await query.SingleOrDefaultAsync(criteria);          }  /\*  هنا النتيجة اللي نريدها ستكون اكثر من سجل بناء على شرط واحتمال يكون في ربط بانكلود لجداول اخرى  ولذلك استخدمنا نفس الاكواد للدالة السابقة ولكن الفلترة هنا بالدالة  Where  اللي نمرر لها  criteria  \*/          public IEnumerable<T> FindAll(Expression<Func<T, bool>> criteria, string[] includes = null)          {              IQueryable<T> query = \_context.Set<T>();              if (includes != null)                  foreach (var include in includes)                      query = query.Include(include);              return query.Where(criteria).ToList();          }  // هنا نمرر برميترات النفجيشن للدوال القرينة بها بعد تمرير الشرط لدالة الوير          public IEnumerable<T> FindAll(Expression<Func<T, bool>> criteria, int skip, int take)          {              return \_context.Set<T>().Where(criteria).Skip(skip).Take(take).ToList();          }  /\*  هنا اضافة للبرميترات السابقة لدينا برميتران خاصين بالترتيب الاول فيه شرط الترتيب بناء على ايش ؟  والثاني يحددنا لنا نوع الترتيب تصاعدي او تنازلي  نبدا اولا بالفلترة بناء على الشرط في دالة الوير  ثم نتحقق من قيم برميترات النفجيشن لو هلها  ثم نتحقق هل هناك شرط للترتيب بناء على ايه  فلو في فهنا سنتحقق هل الترتيب تصاعدي او تنازلي  وبناء على ذلك نكتب الكود المناسب ونطبقه على الكويري  ثم نرجعه من الداله بعدما نقرنه بداله تو لست .  \*/          public IEnumerable<T> FindAll(Expression<Func<T, bool>> criteria, int? skip, int? take,              Expression<Func<T, object>> orderBy = null, string orderByDirection = OrderBy.Ascending)          {              IQueryable<T> query = \_context.Set<T>().Where(criteria);              if (skip.HasValue)                  query = query.Skip(skip.Value);              if (take.HasValue)                  query = query.Take(take.Value);              if(orderBy != null)              {                  if(orderByDirection == OrderBy.Ascending)                      query = query.OrderBy(orderBy);                  else                      query = query.OrderByDescending(orderBy);              }              return query.ToList();          }  // الدوال الثلاث التالية نفس ما سبق ولكن باستخدام دوال لينكيو متزامنة          public async Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> criteria, string[] includes = null)          {              IQueryable<T> query = \_context.Set<T>();              if (includes != null)                  foreach (var include in includes)                      query = query.Include(include);              return await query.Where(criteria).ToListAsync();          }          public async Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> criteria, int take, int skip)          {              return await \_context.Set<T>().Where(criteria).Skip(skip).Take(take).ToListAsync();          }          public async Task<IEnumerable<T>> FindAllAsync(Expression<Func<T, bool>> criteria, int? take, int? skip,              Expression<Func<T, object>> orderBy = null, string orderByDirection = OrderBy.Ascending)          {              IQueryable<T> query = \_context.Set<T>().Where(criteria);              if (take.HasValue)                  query = query.Take(take.Value);              if (skip.HasValue)                  query = query.Skip(skip.Value);              if (orderBy != null)              {                  if (orderByDirection == OrderBy.Ascending)                      query = query.OrderBy(orderBy);                  else                      query = query.OrderByDescending(orderBy);              }              return await query.ToListAsync();          }  // هنا دالة اضافة          public T Add(T entity)          {              \_context.Set<T>().Add(entity);              \_context.SaveChanges();              return entity;          }          public async Task<T> AddAsync(T entity)          {              await \_context.Set<T>().AddAsync(entity);              await  \_context.SaveChangesAsync();              return entity;          }          public IEnumerable<T> AddRange(IEnumerable<T> entities)          {              \_context.Set<T>().AddRange(entities);               \_context.SaveChanges();              return entities;          }          public async Task<IEnumerable<T>> AddRangeAsync(IEnumerable<T> entities)          {              await \_context.Set<T>().AddRangeAsync(entities);              await  \_context.SaveChangesAsync();              return entities;          }  //          public T Update(T entity)          {              \_context.Update(entity);                \_context.SaveChanges();              return entity;          }          public void Delete(T entity)          {              \_context.Set<T>().Remove(entity);                \_context.SaveChanges();          }          public void DeleteRange(IEnumerable<T> entities)          {              \_context.Set<T>().RemoveRange(entities);                \_context.SaveChanges();          }          public void Attach(T entity)          {              \_context.Set<T>().Attach(entity);              \_context.SaveChanges();          }          public void AttachRange(IEnumerable<T> entities)          {              \_context.Set<T>().AttachRange(entities);                \_context.SaveChanges();          }  //          public int Count()          {              return \_context.Set<T>().Count();          }             public async Task<int> CountAsync()          {              return await \_context.Set<T>().CountAsync();          }          public int Count(Expression<Func<T, bool>> criteria)          {              return \_context.Set<T>().Count(criteria);          }            public async Task<int> CountAsync(Expression<Func<T, bool>> criteria)          {              return await \_context.Set<T>().CountAsync(criteria);          }      }  } |

الان علشان نستخدم هذه الدوال لازم نعمل تسجيل للريبوك سيرفس



نلاحظ هنا ان التسجيل بالتايب اوف ومحدد اسم الانترفيس وكلاس الريبو كجنيرك فاضي .

الان ننشى كونترولر بالمؤلفين وننفذ بعض الدوال من الريبو كتجربة فقط للتاكد ان كل شيء تمام  
اول خطوة هي عملية حقن للسيرفس في الكونترولر كالتالي

هنا خلينا الاند بوينت تتعامل مع برميتر ثابت لكن المنطقي ان ال id يجي من الكلاينت وعلى ضوءه يتم البحث ولكن هنا فقط لأغراض التعليم لاغير

الان نشغل الرابط على

<https://localhost:5001/swagger/index.html>

الان نعود لمجلد الكنترولر ونضيف كونترولر اخر للكتب ونعمل حقن لسيرفس الريبو  
ثم نستخدم دواله ولكن هنا سنستخدم عدد اكبر من الدوال لنطبق مفاهيم أوسع وكيف نمرر الشروط للفلترة وغيره

|  |
| --- |
| [ApiController]      [Route("api/[controller]")]      public class BooksController : ControllerBase      {          // عملية حقن السيرفس الخاصة بالريبو وحددنا كلاس البوك كجنرك          private readonly IBaseRepository<Book> \_booksRepository;            public BooksController(IBaseRepository<Book> booksRepository)          {              \_booksRepository = booksRepository;            }  // هنا سيجلب لنا الكتاب اللي معرفه واحد   //ولكنه لن يجلب المؤلف حقه لان ذي داله لا تمرر انكلود للريبو وكذلك الدالة اللي بعدها تعيد لسته كتب بدون ريبو          [HttpGet]          public IActionResult GetById()          {              return Ok(\_booksRepository.GetById(1));          }          [HttpGet("GetAll")]          public IActionResult GetAll()          {              return Ok(\_booksRepository.GetAll());          }  /\*  بهذه الاند بوينت سنتعامل مع دالة  Find  والتي تستقبل برميترين الاول هو الشرط اللي على ضوءه ستتم الفلترة  واللي عنوانه يكون  title book 1  والبرميتر الثاني الخاص بالانكلود اي انه سيضمن لنا سجلات المؤلف الذي يرتبط بهذا الكتاب  \*/          [HttpGet("GetByName")]          public IActionResult GetByName()          {              return Ok(\_booksRepository.Find(b => b.Title == "title book 1", new[] { "Author" }));          }          /\*          هنا سنتعامل مع دالة جلب كل السجلات بناء على شرط والشرط هنا ان العنوان يحتوي على كلمة بوك          وكمان نريد المؤلفين اللي مرتبطين بكل السجلات الراجعة          \*/          [HttpGet("GetAllWithAuthors")]          public IActionResult GetAllWithAuthors()          {              return Ok(\_booksRepository.FindAll(b => b.Title.Contains("book"), new[] { "Author" }));          }          /\*          هنا نستدعي دالة جلب كل السجلات بناء على شرط          ونريط الترتيب يكون بناء على رقم الاي دي ولكن بترتيب تنازلي            \*/          [HttpGet("GetOrdered")]          public IActionResult GetOrdered()          {              return Ok(\_booksRepository.FindAll(b => b.Title.Contains("book"), null, null, b => b.Id, OrderBy.Descending));          }     /\*        هنا دالة ستضيف لنا سجل معين ونحن هنا مررنا الاوبجكت اللي نشتي نضيف مباشرة بشكل ستاتك            \*/          [HttpPost("AddOne")]          public IActionResult AddOne()          {              var book = \_booksRepository.Add(new Book { Title = "Test book", AuthorId = 1 });                return Ok(book);          }      } |

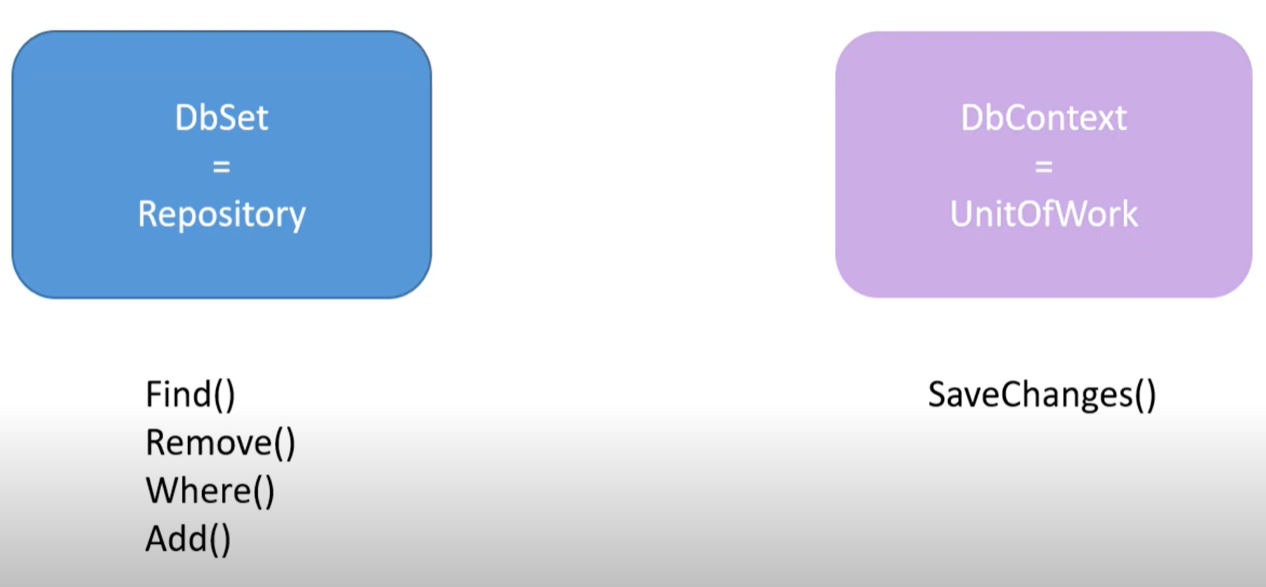
الان كل الدوال اللي جربناها تمام وشغاله والكود واضح

Unit Of Work

الان ناتي لخطوة تطويريه اكثر وهي الانتقال بالريبستوريس هذه الى Unit Of Work ولكن قبل ذلك سنفهم المفهوم هذا .  
وقبل ذلك لو لاحظنا ان ب dbContext هو التمثيل للداتا بيز وهذا الدي بي كونتكست يتكون عادة من dbSet أي كلاسات او كيانات تمثل الجداول .  
وبالتالي نحن لو اردنا انه نتعامل مع جدول ما فهنا أولا نعمل مثيل من dbContext ثم دوت . ثم اسم الدي بي ست dbSet ثم نستخدم الدالة المناسبة ك Find او where .  
الدالة الوحيدة اللي نتعامل فيها مباشرة مع الديبي كونتكست هي SaveChanges فهي تكون مباشرة على مستوى الداتا بيز وليس جداولها

 \_context.SaveChanges();

بنفس هذا المفهوم يعمل نمط Unit Of Work حيث انه سيعمل نفس مهمه dbContext بالنسبة للريبوستوريس أي ان كل ريبو سيكون فيه اشبه ب dbSet .. عدا داله سنسمسها complete سيتقوم بعملية SaveChanges لانها تتم على مستوى الداتا ككل وبالتالي ستكون مرتبطة ب UOW مباشرة كما سنفهم تباعا .

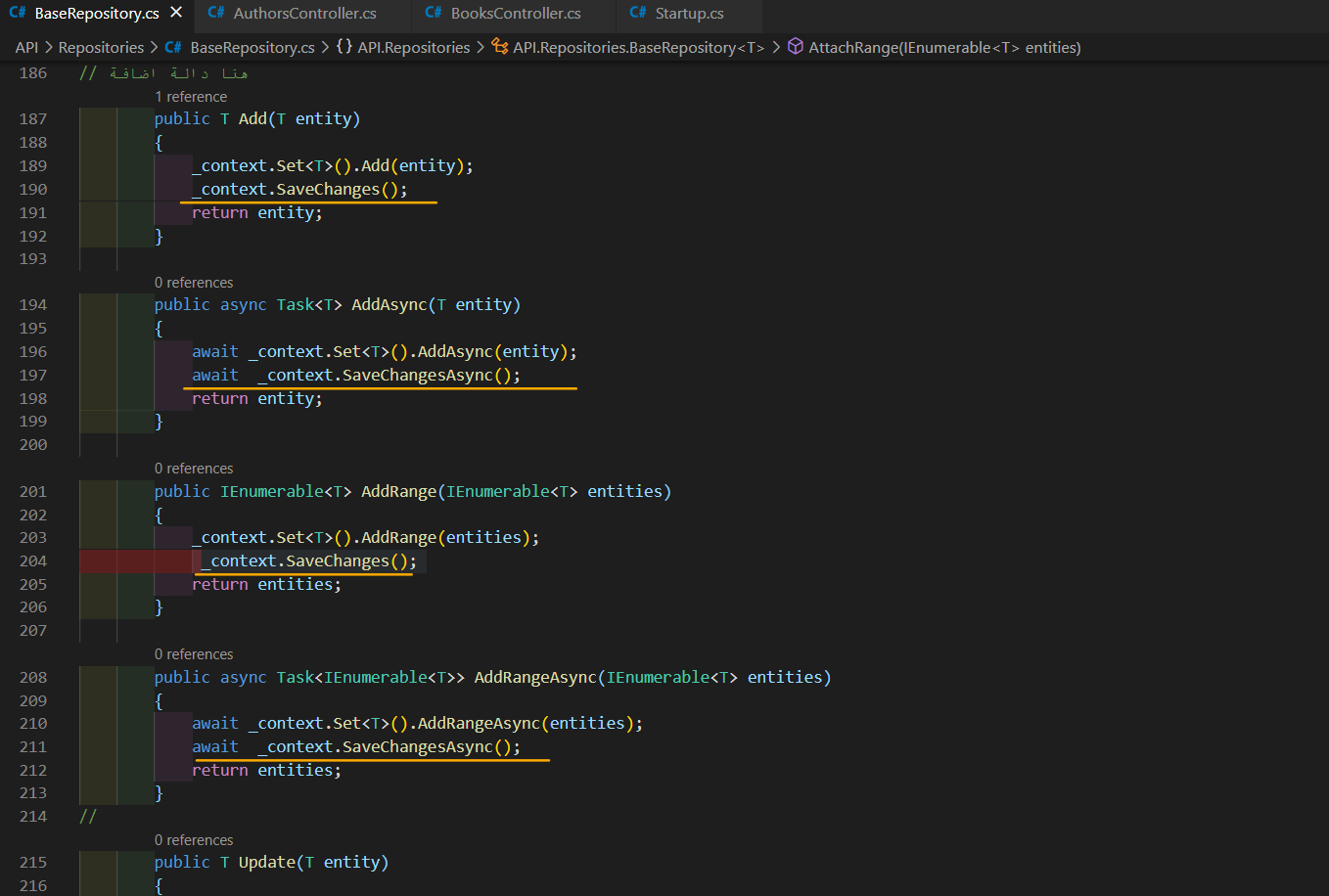


النقطة الثانية اللي لازن نفهمها حول هذا النمط انه لو اردنا نتعامل في كونترولر ما مع اكثر من كيان أي احتجنا في كونترولر الرسائل مثلا الى الوصول الى ريبو المستخدمين او الادمن إضافة الى ريبو الرسائل طبعا فهنا سنعمل حقن للاثنين ريبو في نفس الكونترولر ولكن مع نمط UOW فنحن نتعامل فقط مع حقن لمثيل واحد كما سنتعلم حيث ان UOW يعمل مثيل واحد من الداتا كونتكست .

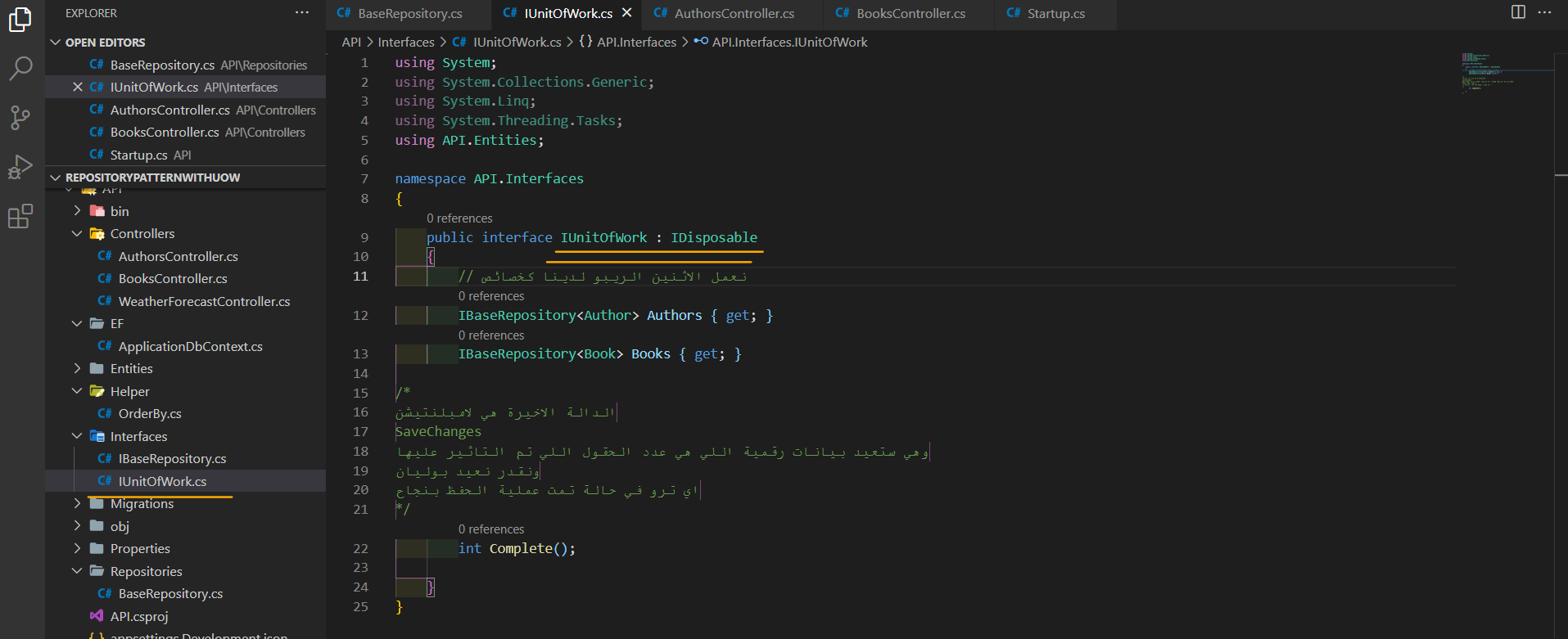
النقطة الثالثة لو لاحظنا في دوال الإضافة والتعديل وغيرها سنجد انه اضطرينا لالحاق السطر

 \_context.SaveChanges();

مع كل داله



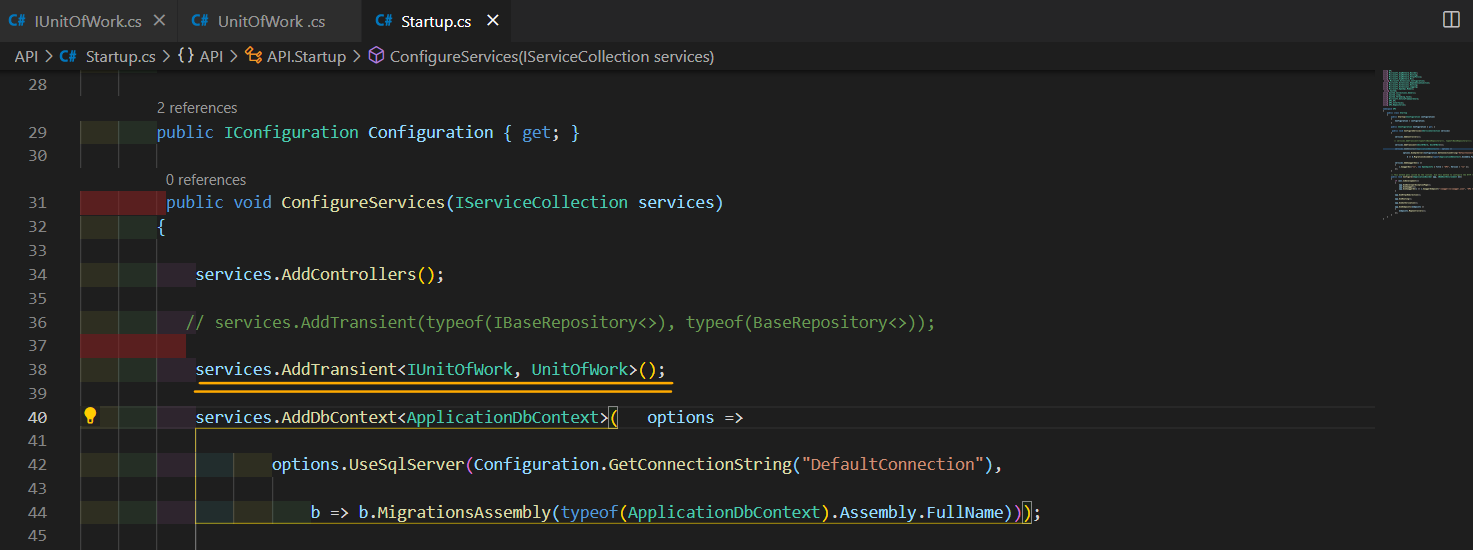
الان سنبدا بعمل التعديلات لتطوير النمط الى عملناه الى نمط UOW والبداية بانشاء انترفيس يضمن الريبو كخصائص إضافة لتعريف دالة خاصة ب SaveChanges علما ان هذه الانترفيس ترث من IDisposable وهي من اجل ان نرغم الريبو اللي بايعمل امبلنتيشن ل IUnitOfWork انه يعمل داله تعمل Dispose أي تفريغ للذاكرة .



الان نعمل كلاس ريبو يعمل امبلمنتيشن ل IUnitOfWork .

|  |
| --- |
| public class UnitOfWork : IUnitOfWork      {  // هنا اشبه بعمل مركزية للداتا كونتكست بحيث نعملها هنا ونمرر المثيل منها الى الريبوستوريس  // ثم نعمل امبلنتيشن للخصائص التي تمثل الريبستوريس ونحط لها قيم اوليه ونمرر لها الكونتكست          public IBaseRepository<Author> Authors { get; private set; }          public IBaseRepository<Book> Books  { get; private set; }          private readonly ApplicationDbContext \_context;          public UnitOfWork(ApplicationDbContext context)          {              \_context = context;              Authors = new BaseRepository<Author>(\_context);              Books = new BaseRepository<Book>(\_context);          }  // هنا دالة السيف تشنج وبعدها دالة تحرير الريسورس          public int Complete()          {              return \_context.SaveChanges();          }          public void Dispose()          {              \_context.Dispose();          }      } |

الخطوة التالية انه في كلاس startUp نعمل الريجستر لسيرفس ال UOW بدلا عن الريبو .



الان قبل ان نقوم بتعديل الكونترولرز الخاصة ب المؤلفين وبالكتاب للتعامل مع مثيل من ريبو IUnitOfWork بدلا عن IBaseRepository .  
سنقوم باجراء تعديل بسيط في BaseRepository بحيث نحذف الكود المتعلق ب SaveChanges من دوال الإضافة والحذف والتعديل ..ألخ لانه لن نحتاج لها بهذا الشكل وسنتعامل مع دالة complate على مستوى IUnitOfWork .  
الان بالعودة الى الكونترولولرز بنبدا بالخاص بالمؤلفين على النحو التالي نعدله

|  |  |
| --- | --- |
| بعد | قبل |
|  |  |

نلاحظ انه في الاكواد كنا نوصل أولا الى مثيل unitOfWork\_ ثم الخاصية التي تمثل الريبو وهي Authers ثم الدالة المطلوبة .

وبالمثل في كونترولز الكتب على النحو التالي

|  |  |
| --- | --- |
| بعد | قبل |
|  |  |

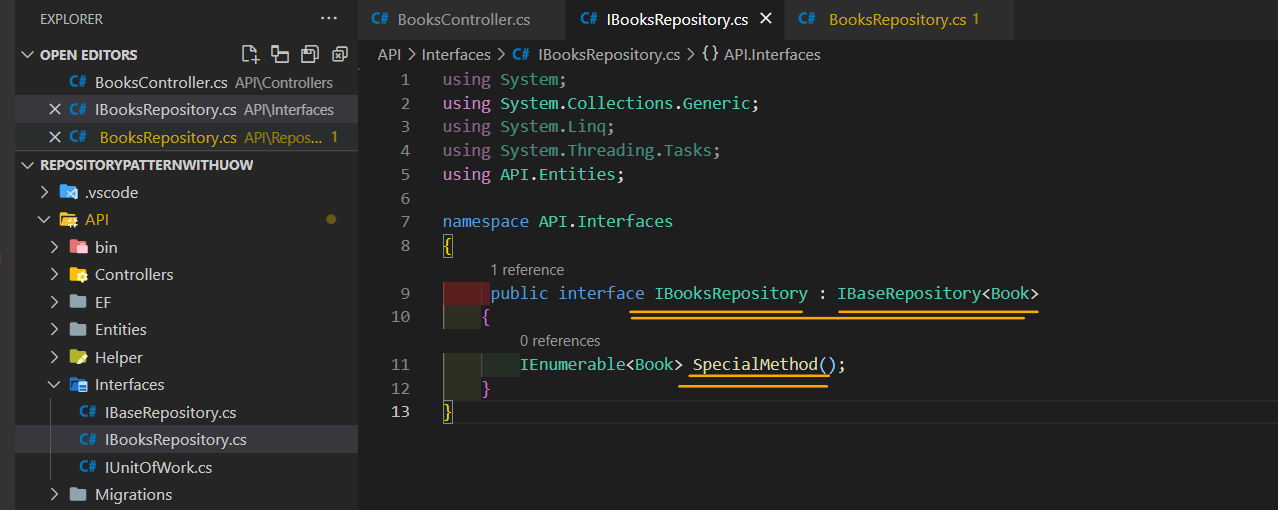
ولان الكود طويل سنقوم بادراجه هنا بعد التعديل الى نمط unitOfWork.

|  |
| --- |
| [ApiController]      [Route("api/[controller]")]      public class BooksController : ControllerBase      {          // عملية حقن السيرفس الخاصة باليونت اوف ورك          private readonly IUnitOfWork \_unitOfWork;          public BooksController(IUnitOfWork unitOfWork)          {              \_unitOfWork = unitOfWork;          }     // بالدوال سنتعامل مع مثيل اليونت اوف ورك ثم الريبو وهنا هي البوكس ثم الدالة المطلوبة  // هنا سيجلب لنا الكتاب اللي معرفه واحد   //ولكنه لن يجلب المؤلف حقه لان ذي داله لا تمرر انكلود للريبو وكذلك الدالة اللي بعدها تعيد لسته كتب بدون ريبو          [HttpGet]          public IActionResult GetById()          {              return Ok(\_unitOfWork.Books.GetById(1));          }          [HttpGet("GetAll")]          public IActionResult GetAll()          {              return Ok(\_unitOfWork.Books.GetAll());          }  /\*  بهذه الاند بوينت سنتعامل مع دالة  Find  والتي تستقبل برميترين الاول هو الشرط اللي على ضوءه ستتم الفلترة  واللي عنوانه يكون  title book 1  والبرميتر الثاني الخاص بالانكلود اي انه سيضمن لنا سجلات المؤلف الذي يرتبط بهذا الكتاب  \*/          [HttpGet("GetByName")]          public IActionResult GetByName()          {              return Ok(\_unitOfWork.Books.Find(b => b.Title == "title book 1", new[] { "Author" }));          }          /\*          هنا سنتعامل مع دالة جلب كل السجلات بناء على شرط والشرط هنا ان العنوان يحتوي على كلمة بوك          وكمان نريد المؤلفين اللي مرتبطين بكل السجلات الراجعة          \*/          [HttpGet("GetAllWithAuthors")]          public IActionResult GetAllWithAuthors()          {              return Ok(\_unitOfWork.Books.FindAll(b => b.Title.Contains("book"), new[] { "Author" }));          }          /\*          هنا نستدعي دالة جلب كل السجلات بناء على شرط          ونريط الترتيب يكون بناء على رقم الاي دي ولكن بترتيب تنازلي            \*/          [HttpGet("GetOrdered")]          public IActionResult GetOrdered()          {              return Ok(\_unitOfWork.Books.FindAll(b => b.Title.Contains("book"), null, null, b => b.Id, OrderBy.Descending));          }     /\*        هنا دالة ستضيف لنا سجل معين ونحن هنا مررنا الاوبجكت اللي نشتي نضيف مباشرة بشكل ستاتك          علما انه هنا تعاملنا مع دالة الكومبليت اللي تعمل السيف تشنج على مستوى الديبي كونتكست ككل          \*/          [HttpPost("AddOne")]          public IActionResult AddOne()          {              var book = \_unitOfWork.Books.Add(new Book { Title = "Test book", AuthorId = 1 });              \_unitOfWork.Complete();              return Ok(book);          }      } |

# Add Special Methods

هنا سنتعلم كيف نعمل لو في لدينا كيان ما يحتاج لدالة خاصة فيه وليست مطلوبة في بقية الكيانات .. أي دالة مثلا سنحتاجها في ريبو الكتب ولكنها ليست مطلوبة في المؤلفين وبقية الكيانات .. فما الحل ؟

ولنفترش اننا نريد في ريبو البوك عمل دالة خاصة لا نريده في بقية التطبيق .. لذا الخطوة الأولى عمل انترفيس باسم public IBooksRepository ستقوم بعمل امبلمنتشن للدوال ريبو الانترفيس العام IBaseRepository . وهذا معناه ان الريبو IBooksRepository سياخذ كل دوال IBaseRepository ويضيف لها دواله الخاصة وهنا لدينا دالة فقط اضفناها .



الان سننشى كلاس باسم BooksRepository في مجلد Repositories من اجل عملية الامبلنتيشن

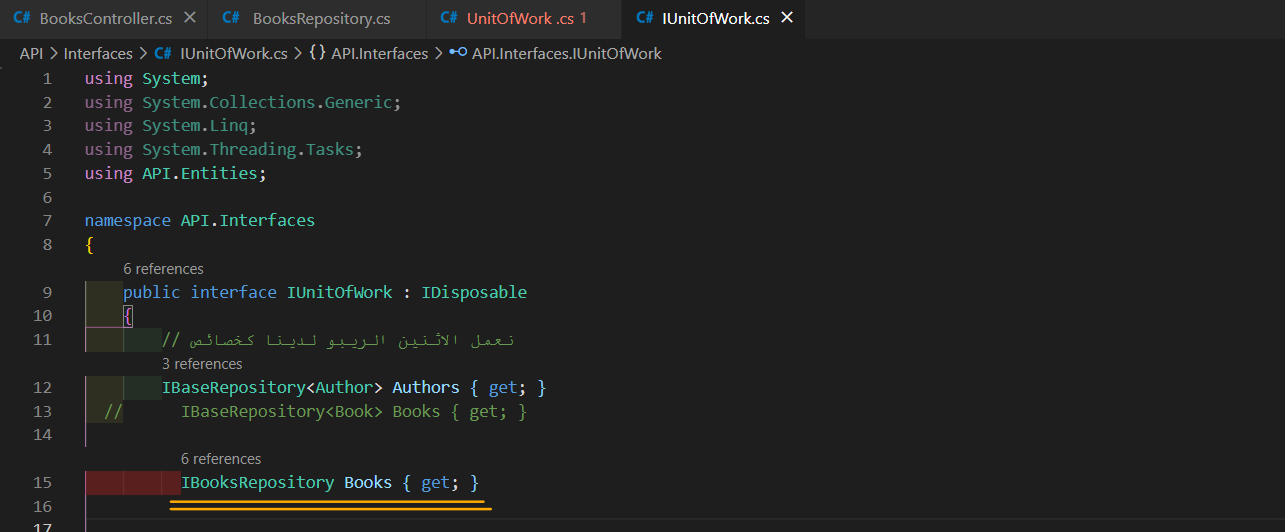
|  |
| --- |
| /\*      الريبو سيرص مت ريبو      BaseRepository      وكذلك  يعمل امبلمنتشن للدالة المضافة في الانترفيس      IBooksRepository      \*/    public class BooksRepository : BaseRepository<Book>, IBooksRepository      {          // لانه سيتعامل مع الكونتكست فسنحتاج لحمل انجكت لها          //  base(context)ولانه مستخدمه في البيس ريبستوري فحددنا بالطريقة          private readonly ApplicationDbContext \_context;          public BooksRepository(ApplicationDbContext context) : base(context)          {          }  // عمل الامبلنتيشن للدالة المضافة في الانترفيس  //IBooksRepository          public IEnumerable<Book> SpecialMethod()          {              // نكتب الكود اللي نريده هنا              throw new NotImplementedException();          }      } |

طبعا الى الان لو رحنا الى الكونترولر الخاص بالكتب وجربنا نوصل لدالة SpecialMethod فلن تظهر لنا .

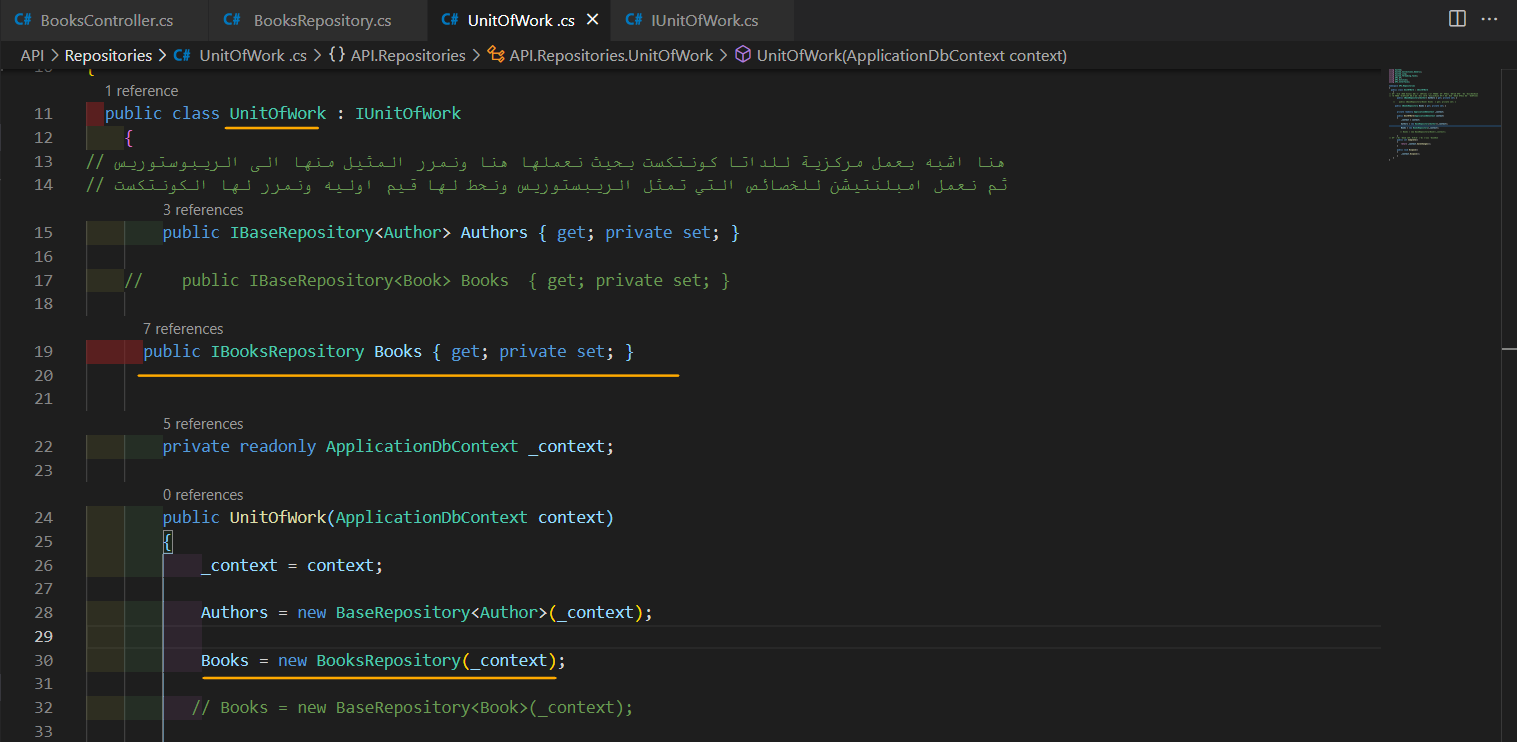
وسبب هذا ان الكونترولر يتعامل مع UOW واللي مسجل في عناصرها

        IBaseRepository<Book> Books { get; }

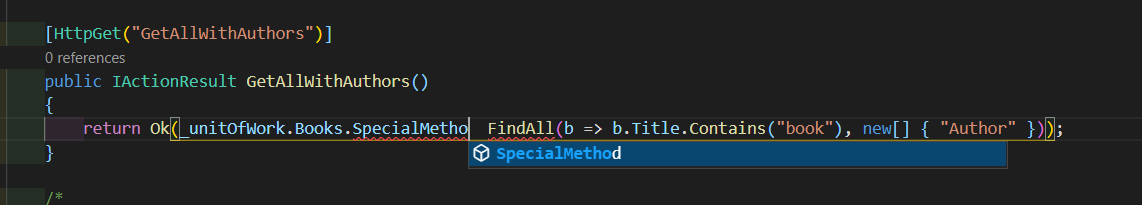
والصح انه الان نجعل العنصر هو الريبو BooksRepository لانه قد عمل امبلمنتشن ل IBaseRepository.



الان محتاجين نعدل جوا الامبلنتيشن على النحو التالي



الان لو جربنا نوصل للدالة SpecialMethod من كنتورلر الكتب سنجدها بمثيل UOW



الرابط التالي رابط جت هب للمشروع ( لكن مقسم في مكتبات كلاسات )  
<https://github.com/muhammadelhelaly/RepositoryPatternWithUOWInDotNetCore6>