Summary Of Content

- Try Catch And Finally
- Difference between IEnumerable Vs IQueryable
- Token
- Difference between view - function - stored procedure
- Polymorphism: Overriding vs. Overloading
- Difference between Query Syntax and Fluent Syntax
- Difference between Lazy Loading and Eager Loading
- Difference between abstract, sealed and static class
- Utility Functions
- Difference between .net, asp.net, .net core, .net framework
- JOIN vs. Nested Query
- View in database
- Inheritance and what is the code will be Create In the first Parent or Child
- Transaction on SQL server
- Dependency injection
- OOP Relationships
- Singleton vs Scoped vs Transient
- Caching
- Stack, heap
- GrapgQL And REST API
- SOLID Principle
- Docker
- Access Modifiers
- Normalization and Denormalization
- Authentication and Authorization
- Dapper And Entity Framework
- ModelBinder And ModelState
- HTTP Status Codes
- Value Type & Reference Type
- Design Patterns
- Database Keys
- Cookies & Session $Local Storage
- JWT
- AutoMapper

- 3-Tier Architecture
- Middlewares
- Serialization & Deserialization
- Data Annotation VS Fluent API
- Value Type & Reference Type
- Redis
- Abstract Class & Interface
- Delete, Drop and Truncate
- .Net Core, .Net Framework
- Trigger
- Delegate
- Constructor
- Synchronous & Asynchronous
- Clustered Index & Non-Clustered Index
- Entity Framework
- String & StringBuilder
- First, FirstOrDefault, Single, SingleOrDefault, Where, Find, ToList
- DTOs
- CQRS
- LINQ
- Code First & Database First
- Unit Testing
- Class & Struct
- .
- .
- .
- .
- .
- .
- .
- .
- .
- .
- .
- .

Try, Catch and Finally

ليه بستخدم try في الكود...

بستخدمها لان ممكن يطلع في الكود اللي بكتبه Exception عشان كده بقول لل C# لو الكود اللي جاي مفيهوش مشاكل نفذيه تمام

طيب لو طلع فيه Exception روح ع ال Block اللي اسمهCatch

ايه هي وظيفه الCatch

لما بتلاقي ال ..Exception بتطلع لليوزر شاشه بتقوله ايه اللي حصل مثلا او تطلب منه بيانات صحيحه

طيب لو انا دلوقتي في ال try بتاعي فتحت Connection مع ال SQL ولسبب ما حصل Exceptionفي الكود في السطر اللي قبل ما قفل فيه الConnection

ده Connection مش المفروض طالما باقى الكود متنفذش اقول لل C# يقفل ال : عشان ميعملش مشاكل لان احتمال افتح Connection مره تانيه فيحصلي مشاكل بدون سبب

عشان كده عملوFinally

دي وظيفتها تعمل Clean Up Resources تمسح كل حاجه ملهاش لازمه في حالة فشل الكود في الtry

---

Difference between IEnumerable Vs IQueryable

تخيل معايا ان انا عندي employee Table فيه 100 row ونا عايز ارجع ال employees الي ال salary بتاعهم أكبر من 2000

```
IEnumerable<Employee> employees = context. Employees
var EmpWithsalaryGT2000 = employees. Where (e => e. Salary > 2000)
```

بالشكل ده كده هيروح يجيب كل الموظفين من الداتا بيز ويحطهم عندي في ال memory وبعد كده يروح يعمل الفلتريشن ال انا طالبها منه ف انت مخيل اللغه ال بيعملها ولو معاك دانا كبيره هيحصل اي طبعا الكلام ده وانا بتعامل مع ال EF

اومال لو استخدمنا ال IQueryable ف المثال ده هيحصل اي ؟

قالك لا هو كده هيروح يجيب الداتا ويعملها فلتريشن هناك ف الداتا بيز ويرجعلي في الداتا ال انا محتاجها لانه بيشتغل بطريقه ال deferred Execution وكده زي الفل شوفت الفرق طيب هل ده معناه ان ال IEnumerable مش بيشتغل ب deferred Execution

هم الاتنين بيعملوا Deferredعادي بس ال IQueryable مش بيتعامل مع ال delegatesبطريقه مباشره زي ال IEnumerable هو بيتعامل مع ال expression وهو بيتعامل مع ال delegates بيجمع ال query كلها ف ال expression وبينفذها ف السيرفر على عكس ال IEnumerable بيتعامل مباشر مع ال delegates ف بيجيب الداتا وينفذ ال extended query انما الاتنين عندهم support ل ال deferred executionعادي

طيب م خلاص ي هندسه انا كده مستخدمش IEnumerable قالك لا هنستخدمه بس مع الداتا ال عندي ف ال memory عندك مثلا dictionary , list كدا هستخدم ال IEnumerable وبرده مع ال streaming operation

ال Enumerable بتفيد جدا لما تتعامل مع مجموعة بيانات كبيرة وعايز توفر استهلاك الmemory يعني مثلا لو عندك قاعدة بيانات كبيرة وعايز تعمل عمليات معينة على البيانات كدا هستخدم ال IEnumerable وبرده مع ال streaming operation

ال Streaming Operations : هي عمليات بتشتغل على البيانات واحدة ورا واحدة من غير ما تحمل البيانات كلها في الذاكرة. يعني بتقدر تعالج البيانات بالتدريج ومش بتحملها كلها في ال memory ويتوفر في استهلاك ال memory

في مثال بسيط، لو عندك قاعدة بيانات فيها مليون سجل، وعايز تطبق على كل سجل عملية معينة، بتستخدم الـ Enumerable مع ال Streaming Operations عشان متحملش المليون سجل في الذاكرة مرة واحدة هتجيب السجلات واحدة ورا واحدة وتعالجها

مفكرتش قبل كده ليه ف ال api مش يستخدم Cookie زى ال MVC بدل الحوارات الكثير بتاعت ال Token دي

⬅ال token بيكون أكثر أمانا من ال cookie ليه ؟

علشان ف ال web APIs انت مش بتتعامل direct مع ال provider لا انت بتتعامل مع ال consumer ال provider بـChek Authentication ويبعتها ل Customer

وال Customer هو الى بيحدد ال Token تتخزن ازاى ف ال Client Side

⬅ال token مش هتحتاج تخزنه في ال browser زي ال cookie علشان بيتبعت في كل request في ال header وده بيخليه أسهل في الاستخدام مع ال APIs

⬅ال token بيدي مرونة أكثر في التحكم في ال access وال permissions علشان ممكن تضيف تفاصيل اكثر في ال payload يناعته بدل من مجرد user id زي ال cookie

⬅ال token بيشتغل على معظم الplatforms وال languages بدون مشاكل ع عكس ال Cookie

⬅ممكن تستخدم الى token في ال authentication وال authorization في نفس الوقت بدل م تستخدام cookie لل authorization وtoken لل authentication

⬅ال token بيسهل تطبيق مفاهيم مهمة في ال statelessness في ال REST APIs.

⬅مع ال token ممكن تستخدم حاجات في ال refresh token عشان تجدد ال access بسهولة.

((ف من الاخر ال token ملاءم اكثر ل APIs من حيث ال ال security وال scalability))...

Difference between view - function - stored procedure

ال View

ال View يتمثل جدول افتراضي مش موجود فعلا في data base

بتظهر لك البيانات من خلال استعلام مخزن في ال View

طيب ايه هي استخدامات ال view :

قالك بتستخدم في تبسيط عملية إدارة ال queries المعقدة وإعادة استخدامها.

وكمان في الأمان والرقابة على الوصول للبيانات باننا بندي صلاحيه للمستخدمين ع الview

بدل الجداول الاساسية الأساسية.

وانه ينفع تغيير ال view من غير م يأثر على البيانات

ال Function

ال Function زي الإجراء بتقوم بعملية معينة وبترجع قيمة.

ممكن تاخد مدخلات وترجع مخرجات

طيب ايه هى استخدام ال function:

بتستخدم في إعادة استخدام الكود

يعني اننا ممكن نستخدم ال function نفسها اكتر من مره بمدخلات مختلفه

وبتبسط ال queries المعقده بحيث اننا نحطها ف ال function ونستدعيها بسهوله

ال Stored Procedure

ال Stored Procedure عبارة عن سلسلة أوامر SQL محزنة في data base

.يتنفذ الأوامر عند استدعائها وممكن تاخد مدخلات وترجع مخرجات

طيب ايه هي استخدامات ال Stored Procedure:

قالك برضو زياده في الامان : بحيث اننا بندي صلاحيات محددة لل User لتنفذ ال stored procedure

بدال الوصول المباشر ل ال table

وقالك اهم نقطه

تقليل حركة الشبكة : في حالة وجود عمليات معقدة بتتم على server بيؤدي استخدام ال stored procedures إلى تنفيذ الكود مرة واحدة بدلا من كل مرة يتم فيها استدعاء تلك العمليات من التطبيق.

سهوله العديل: التعديل السهل ع ال Stored Procedure من غير م يأثر ع اي تطبيق.

وبيستخدم خوارزميات معينه في تحسين اداء ال Stored Procedure في ال data base

---

Polymorphism: Overriding vs. Overloading

- **Overloading: يكون فيه عندي ف ال object اكتر من method او اكتر من Constructor بنفس الاسم ولكن مع اختلاف المدخلات**

- **Overriding: لو عندي object عامل وراثه من Object تاني وال object دا هياخدها بنفس الاسم وبنفس المدخلات ولكن مع اختلاف ال Body**

❖ **Overriding : معانا اني لو عندي اتنين كلاس وواحد منهم ورث من التاني وانا عايز استخدم ال method اللي موجوده ف ال base class**
ف انا بعيد تعريفها ف ال derived class ولكن هستخدمها بشكل مختلف
والمثال دا هيوضح الفكره اكتر

```
public class Shape
{
   public virtual double CalculateArea()
   {
      return 0;
   }
}
```

```
public class Circle : Shape
{
   private double radius;

   public Circle(double radius)
   {
      this.radius = radius;
   }

   public override double CalculateArea()
   {
      return Math.PI * radius * radius;
   }
}
```

طيب ايمته بنستحدم ال overriding
بنستخدمه لما نحتاج دالة في derived class تعمل حاجة مختلفة عن اللي بتعمله نفس الدالة في base class

- **Overloading :** يعني إنك تعمل أكثر من Method بنفس الاسم في نفس الكلاس، لكن كل Method ليها مجموعة مختلفة من الـ parameters .
  الفكرة هنا بقا إن Method دي تقدر تتعامل مع سيناريوهات مختلفة بناءً على عدد أو نوع ال parameters اللي بتستقبلها

زي المثال دا كدا

```
class Calculator
}
public int Add(int a, int b)
}
;return a + b
{

public int Add(int a, int b, int c)
}
;return a + b + c
{  {
```

Deference between Query Syntax and Method Syntax

اولا نبدا ب ال Query Syntax

الَ Query syntax دي الطريقة اللي بتكتب بيها استعلاماتLINQ لكن بطريقة تشبه جدًا كتابة استعلامات.SQL

طيب تستخدمها ايه
بص ي سيدي لو انت متعود على SQL أو بتحب تكتب كودك بطريقة أقرب لقاعدة البيانات، فدي الطريقة اللي هتحبها.

طيب نشوف مثال ع الكلام دا

لو انت عندك table فيه مجموعه من ال Customers

هنا علشان ارجع كل ال Customers اللي ف ال Table

```
var customers = from c in context.Customers
                select c;
```

هنا هرجع كل ال Customers اللي موجودين ف لندن

```
var customersByCity = from c in context.Customers
                      where c.City == "London"
                      select c;
```

ودي علشان ارجع كل ال Customers ولكن مترتبين ابجدي حسب الاسم

```
var orderedCustomers = from c in context.Customers
                       orderby c.Name
                       select c;
```

ثانيا ال Method Syntax

دي الطريقة التانية لكتابة استعلامات LINQ باستخدام Methods بتكون متاحة على Collections زي Arrays

ودي الطريقه الاقرب ل مفاهيم البرمجه

تعاله نشوف مثال ع الكلام دا

هنطبق على نفس المثال

هنا علشان ارجع كل ال Customers اللي ف ال Table

```
var customers = context.Customers.ToList();
```

هنا هرجع كل ال Customers اللي موجودين ف لندن

```
var customersByCity = context.Customers .Where(c => c.City == "London") .ToList();
```

ودي علشان ارجع كل ال Customers ولكن مترتبين ابجدي حسب الاسم

```
var orderedCustomers = context.Customers .OrderBy(c => c.Name) .ToList();
```

طيب ايه الفرق الرئيسي بينهم

- **Method Syntax** يكون أكثر مرونة وقابلية للتوسيع مع إمكانية دمج طرق مختلفة.
- **Query Syntax** بيكون اسهل للي متعود على طريقه ال SQL

الخلاصة:

الاتنين بيوصلوا لنفس النتيجة. استخدامك لأي منهم بيعتمد على تفضيلاتك الشخصية أو طبيعة المشروع اللي شغال عليه. الفكرة إنك تختار الأسلوب اللي يريحك أكتر ويوفرلك إنتاجية أعلى.

Difference between Lazy Loading and Eager Loading

Lazy Loading

دي الطريقة اللي بتحمل البيانات من Database وقت م بحتاجها فقط. يعني لما تحمل object معين من Database زي Customer table البيانات المرتبطة بيه زي Orders مش هتتحمل إلا لما تستدعيها في الكود.

- طيب ايه هي مميزات ال Lazy loading : بص ي سيدي بتقلل من كمية البيانات المحملة من Database في البداية، ودا بيساعد على تحسين الأداء
- طيب وايه هي العيوب : بسبب زيادة عدد الاستعلامات اللي بتحصل ف Database ودا ممكن يتسبب في تدهور الأداء في بعض الحالات.

Eager Loading

ودا عكس ال Lazy loading تماماً لانه بيتم فيه تحميل كل البيانات المرتبطة ب ال Object في استعلام واحد مثلاً لما تحمل Customer ومعاه Orders في نفس الوقت باستخدام جملة Include ده بيساعد في تقليل عدد الاستعلامات

- طيب ايه هي مميزات ال Eager loading : بتقلل من عدد الاستعلامات المرسله ل ال Database لانه بيتم تحميل كل البيانات المطلوبة في استعلام واحد
- طيب وايه هي العيوب : ممكن يؤدي لتحميل كمية كبيرة من البيانات اللي مش محتاجها في الوقت الحالي ودا هيأثر بالسلب على الأداء واستهلاك الذاكرة

طيب ايه ملخص كل دا

- Lazy loading : برجع البيانات اللي بحتاجها بس وده بيساعد في تقليل تحميل البيانات لكن ممكن يزيد عدد الاستعلامات بعد كده ودا ممكن يتسبب ف تدهور الاداء
- Eager loading : هنا بيرجع كل البيانات الموجوده ف ال object وده بيكون مفيد لو عايز تقلل عدد الاستعلامات لكن ممكن يحمل بيانات أكتر من اللي محتاجها في الوقت الحالي.

Difference between abstract, sealed and static class

Abstract Class

- ايه هو ال Abstract class دا كلاس مينفعش تنشأ Objects منه بطريقه مباشره بيتم استخدامه ك base class وباقي ال Classes بتورث منه بيتحتوي على Methods لكن بدون تنفيذ بتعمل ك abstract methods لكن الكلاس اللي بيورث منه هو اللي بينفذ ال method دي
- طيب بنستخدمه ايمته ؟؟ بنستخدمه اما بيكون عندي مجموعه من ال Classes ليها Common Behavioral وتضمن ان الكلاس اللي بيورث منه ينفذف بعض ال methods اللي موجوده ف ال bass class

Sealed Class

- ايه هو ال Sealed Class بص ي سيدي دا كلاس مقفول على نفسه مينفعش اي كلاس تاني ي inherit منه
- طيب ايمته بنستخدمه ؟؟ دا بنستخدمه اما بنكون عايزين نمع اي كلاس تاني انه يورث منه او يخصص منه جزء يعني انت كدا قفلت الباب ف وش اي كلاس تاني يحاول انه يورث منه

Static Class

- ايه هو ال Static Class دا كلاس اتصمم علشان علشان يحتوي على Static Methods And Static Data يعني انت متقدرش تعمل Instance من الكلاس دا وطبعا كل ال Methods اللي موجوده فيه لازم تكون Static
- طيب ايمته بنستخدمه ؟؟ دا بتستخدمه اما بتكون عايز تجمع مجموعة من Methods اللي ملهاش علاقة ب Object معين زي utility functions اللي ممكن تحتاجها في أي مكان في ال APP

الخلاصه ..

- **Abstract Class** كلاس بيكون bass class للكلاسات التانية، وما تقدرش تنشئ منه Instances مباشرة.
- **Sealed Class** كلاس مش ممكن تورث منه.
- **Static Class** دا كلاس بيكون كل Methods اللي فيه بتكون Static وما تقدرش تعمل منه Instances

---

إيه هي Utility Functionsأو Helper Methods؟🧐

Utility Functions هي عبارة عن مجموعة دوال بتقوم بمهام شائعة ومتكررة في البرنامج. الهدف منها هو تسهيل حياتك كمبرمج من خلال تجميع الوظائف اللي ممكن تحتاجها في أكتر من مكان في الكود بتاعك.

طيب، بنستخدمها في إيه؟

- **العمليات الحسابية البسيطة :**زي دالة لجمع رقمين أو تحويل درجة الحرارة من فهرنهايت إلى مئوية.
- **معالجة النصوص :**زي دالة لتحويل نص إلى حروف كبيرة أو لحساب عدد الكلمات في جملة.
- **التعامل مع الملفات :**زي دالة لقراءة محتوى ملف نصي أو كتابة نص إلى ملف.

إيه اللي بيميزها؟

- **إعادة الاستخدام :**تقدر تستخدم الدوال دي في أي مكان في الكود بدل ما تعيد كتابتها كل مرة.
- **العزل :**بتنفيذ وظيفة معينة بشكل مستقل، فبتكون سهلة في الاختبار والصيانة.
- **العمومية :**مفيدة لأكتر من جزء في التطبيق، وبتقلل حجم الكود وتوحّد استخدامه.

إمتى تستخدمها؟

- لما يكون عندك مهام متكررة وعايز تختصر الكود.
- لما تكون الوظيفة مش مرتبطة بحالة معينة أو كائن معين، زي عمليات التحويل أو الحسابات.

Utility Functionsهي جزء أساسي من كتابة كود منظم ونظيف، وبتسهل كتير من العمليات المتكررة في البرمجة.

استخدمها صح، وهتوفر على نفسك وقت وجهد كبير ✨💻 !

Difference between .net , asp.net , .net core , .net framework

.NET

- **إيه هو؟** .NET :هو إطار عمل شامل تم تطويره بواسطة شركة .Microsoft بيشمل مجموعة من الأدوات والمكتبات اللي بتساعد المطورين في بناء تطبيقات متنوعة زي تطبيقات الويب، سطح المكتب، الأجهزة المحمولة، والخدمات السحابية.
- **الفكرة** .NET :هو المصطلح الكبير اللي بيضم جميع الإطارات الفرعية المختلفة زي **ASP.NET**، **.NET Core**، و .NET.** Framework**.

ASP.NET

- **إيه هو؟** ASP.NET :هو إطار عمل فرعي مبني على .NET. ومخصص لتطوير تطبيقات الويب والخدمات. بيسمح للمطورين ببناء مواقع ويب، تطبيقات ويب، وواجهات برمجة التطبيقات.(APIs)
- **الفكرة** ASP.NET :هو الجزء من .NET. اللي بتركز على تطوير تطبيقات الويب تحديدًا.

.NET Core

- **إيه هو؟** .NET Core :هو إصدار حديث ومفتوح المصدر من .NET. تم تصميمه ليكون متعدد المنصات(cross-platform) ، يعني يشتغل على أنظمة تشغيل مختلفة زيWindows ، Linux، و.macOS
- **الفكرة** .NET Core :بيتيح للمطورين بناء تطبيقات بتشتغل على أي نظام تشغيل، وهو المستقبل للتطوير في بيئة.NET.

.NET Framework

- **إيه هو؟** .NET Framework :هو الإصدار الأصلي منNET. ، وكان مخصص فقط للعمل على نظام .Windows تم إصداره لأول مرة في عام 2002 وهو مش مفتوح المصدر. بيدعم كل الأنواع التقليدية من التطبيقات زي تطبيقات سطح المكتب (Windows Forms, WPF) وتطبيقات الويب.(ASP.NET)
- **الفكرة** .NET Framework :كان الخيار الوحيد لتطوير تطبيقات .NET. قبل ظهورNET Core. ، وهو بيشتغل بس على نظام Windows

إيه العلاقة بينهم؟

- :**.NET**المصطلح الشامل اللي بيضم كل من **ASP.NET**، **.NET Core**، و.**.NET Framework**
- :**ASP.NET**جزء من .NET. مخصص لتطوير تطبيقات الويب.
- :**.NET Core**النسخة الحديثة والمفتوحة المصدر والمتعددة المنصات من.NET.
- :**.NET Framework**النسخة الأصلية والمخصصة فقط لنظام Windows من.NET.

إيه اللي لازم أستخدمه؟

- لو بتشتغل على مشروع جديد وعايز يكون متعدد المنصات، استخدم) .NET Core أو .NET 5/6وما بعدها لأنها بتجمع بين .NET Coreو .NET Framework في منصة واحدة.(
- لو بتطور تطبيق قديم بيشتغل علىWindows ، هتلاقي نفسك غالبًا في **.NET Framework**.
- لو بتبني تطبيق ويب، ممكن تستخدم **ASP.NET**على .NET Core. أو .NET Framework. حسب احتياجاتك.

JOIN vs. Nested Query

ايه الافضل فيهم من حيث الاداء ؟؟؟

لما بنشتغل مع قواعد البيانات كتير مننا بيسأل أيه الافضل من حيث الأداء استخدام JOIN ولا Nested Query

تعالا اقولك ايمته نستخدم كل حاجه فيهم

الاول تعالا نفهم يعني ايه ال Join و Nested Query ؟؟؟؟؟

اولا ايه هو ال Join

هو عملية بتربط بين جدولين أو أكثر على أساس علاقة معينة وغالبًا بيتم دمج البيانات بناءً على عمود مشترك

يعني ايه برده ؟؟

تعالا اقولك مثال ع الكلام دا

لو انا عندي Database فيها جدولين Customers وجدول تاني Orders وعايز اعمل Select ل orders مع الداتا الخاصه ب كل Customer اللي عمل ال order دا

```
SELECT Customers.Name, Orders.OrderID

FROM Customers

JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

ثانيا ال Nested Query

Nested Query == Subquery

ودا بيكون Query داخل Query تانيه طيب ايمته بنستخدمه

بنستحدم ال Nested Query لو هعمل Filter ل داتا بناءا على استعلام تاني

طيب يعني ايه برده ؟؟

تعالا نقول مثال للكلام دا

لو عندي نفس ال Database اللي استخدمناها ف ال Join وكنت عايز اعمل Select ل ال Customers اللي عملوا Orders

ف المثال دا ممكن نستخدم ال Nested Query

```
SELECT Name

FROM Customers

WHERE CustomerID IN (SELECT CustomerID FROM Orders);
```

فهمنا كدا يعني ايه join and Nested Query , تعالا بقا نتكلم ف ال performance

ال Join

- في الغالب، الـ JOIN بيكون أسرع من الـ Nested Query في معظم الحالات، خاصة لما تكون Data كبيرة

لأن Database بتستخدم Advanced Optimization Techniques عشان تربط بين ال Tables بكفاءة.

- الـ JOIN بيساعد ال Database في استخدام Indexes بشكل أفضل وبالتالي بيساعد على تقليل وقت الاستعلام.

طيب وال Nested Query

الـ Nested Query ممكن يكون أقل كفاءة في بعض السيناريوهات خاصة لو كان inner query بيعالج Data كبيرة أو بيتم تنفيذه أكتر من مرة ده ممكن يؤدي لتحميل زائد على ال Database

تمام كدا

طيب برده معرفناش ايمته نستخدم Join وايمته نستخدم ال Nested Query

استخدم JOIN لما يكون عندك جداول مرتبطة ببعضها وعندك Relationships واضحة بينهم ده هيساعدك في الحصول على Data بسرعة وكفاءة أعلى.

استخدم Nested Query لما يكون عندك inner query بسيط أو لو كنت محتاج تعالج Data بناءً على نتيجة استعلام query تانيه مش مرتبطه مباشرة بالجدول الرئيسي.

كدا فهمنا الدنيا الحمد لله .

تعاله اقولك ايه بقا خلاصة كل الكلام دا

الـ JOIN بيكون أفضل من حيث ال performance في معظم الحالات خاصة مع ال Data الكبيرة لكن القرار النهائي يعتمد على السيناريو اللي بتتعامل معاه . ويفضل تجرب بنفسك وتشوف ايه الافضل تستخدمه بالنسبه للحاجه اللي انت شغال عليها

ايه هو ال View وهل التغيير فيه هيغير ف ال Database ولا لا

الـ **View** في Database هو نوع خاص من Query بيتم حفظه ك Object في Database

باختصار يعني هو جدول وهمي virtual table يعتمد على نتائج Query معين الـ View مش بيخزن البيانات بشكل فعلي لا هو بيعرض ال Data اللي راجعه من ال Database بناءا على الشروط الموجوده ف ال Query

طيب ايه هي فايدة ال View ؟؟

بص انا اكتر من من Benefit ل ال view

1) Simplifying Complex Queries
2) Enhancing Security
3) Customized Data Display

هنتكلم على كل جزء فيهم

1. Simplifying Complex Queries

ممكن تستخدم الـ View لتبسيط ال Queries المعقدة اللي بتستخدمها بشكل متكرر بدل ما تكتب Query كل مرة

اقدر اخزنه كـ View واستدعيه بسهولة.

2. Enhancing Security

يعني ممكن ب استحدامي ل ال View اقدر اني اعمل Hide ل Data او Columns معينه من ال original table

لما ادي ل ال users صلاحيات ال view

3. Customized Data Display

يعني اني اقدر تستخدم الـ View لعرض ال Data بطريقة معينه تناسب احتياجات محددة من دون التأثير على Original table

تمام كدا عرفنا ايه هو ال view وايه هي ال فوايده

طيب لو فيه اي تغيير ف ال view دا هيغير ف ال Real Data ولا لا

- في أغلب الأحيان، التغيير في الـ View مش بيأثر على Rela Data لأنه بيكون Read only يعني أي تعديل بتعمله من خلال الـ View مش بيتخزن في Databases

- لكن في بعض الحالات الخاصة ممكن يكون عندك View قابل للتحديث، وساعتها التعديل هيترجم لتغيير في الجداول الأصلية

بس ده بيتطلب شروط معينة زي إن يكون الـ View بسيط وما يحتويش على Aggregate Function أو complex expressions

طيب ايه هو ملخص كل الكلام دا

الـ View بيساعدك في تنظيم ال Data وتبسيط ال Queries لكن التغييرات من خلاله غالبًا مش بتأثر على Original Data إلا لو كان Updatable View

---

لو كلاس بيورث التاني ايه الكود اللي بيتنفذ الاول اللي ف ال Parent ولا اللي ف ال Child

لما يكون عندك كلاس بيورث من كلاس تاني بمعنى إن عندك **Parent Class** و**Child Class** ، **الترتيب اللي بيتنفذ بيه الكود** بيكون كالتالي:

1. **الـ Constructor الخاص بالـ Parent Class بيتنفذ الأول.**
2. **بعد كده، الـ Constructor الخاص بالـ Child Class بيتنفذ.**

ليه بيحصل كده؟

ده بيحصل لأن الكلاس الأب (Parent Class) لازم يتأكد إن كل خصائصه ومتغيراته تم تهيئتها بشكل صحيح قبل ما يبدأ الكلاس الابن (Child Class) في تنفيذ أي كود خاص بيه. ده مهم لضمان إن الكائن اللي بينشأ يكون في حالة صحيحة وجاهز للاستخدام.

ايه هو ال Transaction ف ال SQL

الـ **Transaction**في SQL هو مجموعة من العمليات (مثل إدخال أو تحديث أو حذف بيانات) التي تتم كوحدة واحدة، يعني كل العمليات اللي داخل الـ Transaction بتتنفذ سوا أو ما بتتنفذش خالص. الهدف من الـ Transaction هو ضمان سلامة البيانات، بمعنى إنه لو حصل خطأ في أي عملية من العمليات داخل الـTransaction ، يتم إلغاء كل العمليات اللي تمت قبله وترجع البيانات لحالتها الأصلية.

خصائص الـ:Transaction (ACID)

1. **Atomicity الذرية**
   - بتضمن إن كل العمليات في الـ Transaction تتم بالكامل أو لا تتم على الإطلاق. لو حصل خطأ، يتم إلغاء كل العمليات اللي تم تنفيذها مسبقًا.
2. **Consistency التناسق**
   - بتضمن إن الـ Transaction بينقل قاعدة البيانات من حالة صحيحة إلى حالة صحيحة أخرى. يعني لو فيه قيود أو شروط على البيانات، بتفضل البيانات متوافقة مع الشروط دي قبل وبعد الـTransaction.
3. **Isolation العزل**
   - بتضمن إن العمليات اللي داخل الـ Transaction بتتم بمعزل عن أي عمليات أخرى بتتنفذ في نفس الوقت، بحيث إن الـTransaction ات المتعددة ما تأثرش على بعض.
4. **Durability الدوام**
   - بتضمن إن بمجرد انتهاء الـ Transaction بنجاح، التغييرات اللي حصلت على البيانات تفضل موجودة حتى لو حصل انقطاع في الطاقة أو تعطل النظام.

مثال بسيط:

تخيل إنك عايز تنقل مبلغ من حساب بنكي لآخر. الـ Transaction هنا هيكون بالشكل ده:

1. **خصم** المبلغ من الحساب الأول.
2. **إيداع** المبلغ في الحساب الثاني.

```
BEGIN TRANSACTION;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;

COMMIT;
-- لو أي عملية من العمليات دي فشلت (زي انقطاع الكهرباء أثناء العملية)، قاعدة البيانات هتلغي كل العمليات وترجع لحالتها الأصلية باستخدام .ROLLBACK
```

الخلاصة:

الـ Transaction بيضمن إن كل العمليات داخل الوحدة دي إما تنجح كلها أو تفشل كلها، عشان يحافظ على سلامة البيانات ويمنع أي تلف فيها.

ايه هو ال Dependency injection

Dependency Injection دا Design Pattern بيستخدم في تطوير ال Software علشان يفصل مكونات ال System عن بعضها ودا بيساعد على تسهيل إدارة Dependencies الفكرة الأساسية من Dependency Injection هي أن ال Objects لا تقوم بإنشاء ال Dependencies اللي بتحتاجها بنفسها لكن بيتم توفير ال Dependencies من مصدر خارجي زي ال Framework

طيب ال Dependency injection بتشتغل ازاي ؟؟

ف ال Traditional Programming ال Object بيقوم بإنشاء ال Dependencies اللي بيحتاجها بنفسه ودا بيؤدي إلى ترابط قوي بين ال Objects ودا بيخلي من الصعب اي تعديل أو استبدال ال Dependencies. لكن باستخدام Dependency Injection بيتم حقن ال Dependencies اللي بيحتاجها ال Object من مصدر خارجي ودا بيقلل من الترابط وبيسهل عملية الصيانة والتطوير

طيب ايه الفايده من ال Dependency injection ؟؟

1. More Flexibility : الكود بيكون مرن وسهل اني اعدل عليه يعني اقدر اغير او اعدل على اي Dependency من غير م اعدل كتير ع الكود

طيب ايه الفائدة من ده ؟؟

Changing Dependencies : لو احتجت تغير ال Dependency زي استبدال Service بأخرى أو تحديث تنفيذ الكود فببساطة تقدر تعمل كدا من غير الحاجة لتعديل الكود الأساسي الذي بيعتمد على ال Dependencies

Easier Maintenance : الكود الأساسي بيكون أقل تعقيد لأن ال Dependencies بيتم إدارتها في مكان تاني ودا معناه أن التعديلات على ال Dependencies مش هيحتاج تغييرات كبيرة في الكود الأساسي ودا هليسهل عملية الصيانة.

مثال علشان نفهم اكتر :

لو عندك Application بيعتمد على خدمة معينة زي خدمة إرسال رسائل وقررت أنك تغير طريقة إرسال الرسائل أو تستخدم خدمة مختلفة باستخدام **Dependency Injection**، كل اللي هتحتاج تعمله هو تغيير injected dependency في ال App والكود الأساسي اللي بيعتمد على الخدمة هيفضل زي ما هو من غير ما تحتاج تعدل عليه.

وبناءا على الكلام دا ان **Dependency Injection** بيسمح بمرونة أعلى في التعامل مع ال Dependencies ودا بيخلي الكود أسهل في التعديل والتحديث.

2. Easier Testing : لما بنحب نعمل Unit Test بنقدر نستبدل ال Dependency ب mock objects بسهوله يعني ايه برده الكلام دا ؟؟

معناه إنك لما تيجي تختبر جزء معين من الكود زي Class او function ممكن تكون فيه أجزاء تانية من الكود (Dependencies) الكود ده بيعتمد عليها.

في حالة الـ **Dependency Injection** ال Dependencies دي بتكون منفصلة عن الكود اللي بتختبره وده بيسمحلك إنك تستبدلها Mock Objects أثناء الاختبار

ال Mock Objects دي بتتصرف زي ال Dependencies الحقيقية لكنها بتكون أبسط وأسهل في التعامل أثناء الاختبار.

مثال ع الكلام دا علشان نفهم اكتر :

لو عندك Class بيتعامل مع Database ممكن تستبدل ال object اللي بيتعامل مع Database ب Mock Object ودا مش بيوصل ل ال Database الحقيقية الهدف من ده إنك تختبر الكود بتاعك في ظروف معينة من غير ما تضطر تتعامل مع System أكبر أو أكتر تعقيد.

بناءا على الكلام دا ان **Dependency Injection** بيسهل إنك تعمل اختبارات دقيقة وسريعة من غير ما تعتمد على ال Dependencies الفعلية ل ال System وده بيسهل عملية التأكد من صحة الكود.

3. Simplified Maintenance : لو في حاجة عايزين نعدلها أو العملية بتكون أسهل بكتير
يعني أن التعديلات أو التوسعات التي محتاج ادخلها في النظام اقدر انفذها بشكل أبسط من غير م احتاج اني اغير كثير ف الكود الأساسي.

مثال ع الكلام دا علشان نفهم اكتر :
افترض أنك شغال على تطبيق يتطلب خدمة معينة زي خدمة للدفع الإلكتروني لو قررت تغيير طريقة الدفع أو إضافة طريقة دفع جديدة تقدر تعدل أو تضيف ال Dependencies الخاصة بالخدمة فقط والكود الرئيسي للتطبيق هيكون زي م هو ودا بيخلي عملية الزياده أسهل.

بناءا على الكلام دا ان **Dependency Injection** بيجعل عملية تعديل واضافة الكود أكثر سهولة وفعالية لأنه بيقلل من الترابط بين الكود وال Dependencies ودا بيخليك تضيف تغييرات وإضافة ميزات بشكل أسرع وأبسط.

طيب ايه هي انواع ال Dependency injection

1. Constructor Injection
2. Property Injection
3. Method Injection

هنشرحهم ف بوست تاني ان شاء الله .....

ايه هي الـ OOP Relationships

(Composition)

- **يعني إيه؟** الكائن بيحتوي على كائن تاني كجزء منه. بمعنى إن الكائن الكبير بيعتمد على الكائن الصغير في عمله.
- **مثال :**تخيل إن عندنا كائن سيارة(Car) ، والكائن ده عنده محرك محرك (Engine) جواه. هنا بنقول إن "السيارة تحتوي على محرك."

(Inheritance)

- **يعني إيه؟** كائن بيورث صفاته وسلوكياته من كائن تاني. يعني الكائن الجديد بيكون نسخة معدلة من الكائن الأصلي.
- **مثال :**لو عندنا كائن حيوان(Animal) ، وكائن تاني كلب (Dog) بيورث من .Animalهنا بنقول إن "الكلب هو نوع من الحيوان."

(Association)

- **يعني إيه؟** علاقة بسيطة بين كائنين بيشتغلوا مع بعض بشكل مستقل.
- **مثال :**كائن معلم (Teacher) وكائن طالب(Student) ، المعلم بيدرس للطالب، بس الاتنين بيكونوا كائنات مستقلة.

(Aggregation)

- **يعني إيه؟** نوع من التركيب، بس الكائنات اللي جواه ممكن تعيش من غير الكائن الكبير.
- **مثال :**كائن فريق (Team) بيحتوي على مجموعة لاعبين .(Players) اللاعبين ممكن يكونوا موجودين من غير الفريق، لكن الفريق بيجمعهم.

(Interface Implementation)

- **يعني إيه؟** الكائن بيلتزم بتنفيذ واجهة معينة، يعني بيتعهد بتنفيذ مجموعة من الدوال اللي الواجهة دي بتحددها.
- **مثال :**لو عندنا واجهة Animal وكائن Dog بينفذها، لازم Dog ينفذ كل الدوال اللي موجودة في .Anima

الفرق بين Singleton vs Scoped vs Transient

اما بنستخدم الـ Dependency Injection بنلاقي 3 أنواع أساسية من Object Lifetimes

Singleton , Scoped , Transient وكل حاجه فيهم ليها طريقه علشان ت Manage ال Objects Lifetimes

هنحاول نشرحهم بطريقه بسيطه ان شاء الله علشان نفهم الفرق بينهم وايمته نستخدم كل حاجه فيهم

Singleton

إيه فكرته؟

الـ **Singleton** بيمثل Object بيتم إنشاؤه مرة واحدة فقط ف ال App يعني بمجرد ما يتم إنشاؤه لأول مرة كل مرة تستخدمه فيها بعد كده هتحصل على نفس النسخة اللي معموله

طيب ايمته تستخدمه؟

هقولك لو عندك Data مشتركة زي Application Service او Logging services اللي هتفضل ثابته طول فترة تشغيل ال App

مثال علشان الكلام يكون واضح

لو عندي object مسئول عن انه يحفظ ال App Setting كدا اذا مش محتاج اعمل Create كل مره ل نسخه منه تاني لا انا بستخدم ال Singleton علشان الكود اللي انا عملته او النسخه تفضل ثابته متتغيرش

Scoped

إيه فكرته؟

الـ **Scoped** بيتم إنشاء object واحد لكل HTTP Request يعني لما ييجي Request جديد من ال User اذا ال Object ده بيتعامل معاه طوال فترة ال Request وبمجرد انتهاء Request خلاص كدا ال Object دا مهمته خلصت ف بيتم التخلص منه. لكن لو في Request جديد بنعمل Create ل Object جديد علشان يتعامل مع ال Request دا

طيب ايمته تستخدمه؟

استخدم Scoped لما تحتاج Object علشان يتعامل مع Data معينة لكل Request زي ال Objects اللي بتتعامل مع Database زي الـ DbContext في Entity Framework

مثال علشان الكلام يكون واضح

لو عندك Request من Browserعشان تجيب Data من Database اذا ال Scoped Object زي الـ DbContext هيتم إنشاؤه بداية من ال Request وينتهي بعد ما Request يخلص.

Transient

إيه فكرته؟

الـ Transient بيتم إنشاء Object جديد كل مرة تستدعي فيها ال Object يعني لو استدعيت نفس ال Object مرتين في نفس ال Request هيتم إنشاء نسختين مختلفتين من ال Object دا

طيب ايمته تستخدمه؟

بتستخدمه لما يكون ال Object مش بيحتفظ بأي Data أو حالة ثابتة أو لما يكون خفيف وسهل الإنشاء زي خدمة لعمليات الحسابات.

مثال علشان الكلام يكون واضح

لو عندك خدمة بتنفذ عملية سريعة زي حساب تكلفة منتج كل مرة هتطلب الخدمة هتحصل على نسخة جديدة من ال object

طيب ايه ملخص الكلام دا ي باشا

- **Singleton دا Object واحد بيعيش طول مدة تشغيل ال App مناسب للاستخدام في Shared Data**
- **Scoped بيتم إنشاؤه لكل Http Request وبيعيش طول مدة ال Request**
- **Transient بيتم إنشاء Object جديد كل مرة يتم استدعاؤه حتى في نفس ال Request**

LifeTime	Example	When to Use
Singleton	**services.AddSingleton<AppConfig>();**	**For shared data like configuration or logging services**
Scoped	**services.AddScoped<ApplicationDbContext>();**	**For objects that need to last throughout a request, like DbContext**
Transient	**services.AddTransient<DiscountService>();**	**For stateless, lightweight objects that don't hold any state**

Caching

ليه الـ Caching مهم جدًا في تحسين أداء ال APPs؟🤔 !

تخيل معايا كدا انك مثلا كل يوم تنزل تشتري نفس الحاجه من السوبر ماركت , متعبه صح ؟

طيب ليه تعمل نفس التعب دا كل يوم وانت تقدري تجيب كميه مثلا وتخزنها عندك ف البيت علشان متتعبش مشوار كل يوم للسوبر ماركت

اهو ال Caching نفس الموضوع . ودا موضوعنا انهارده

ايه هو الـ Caching ؟

بص ي سيدي ال Caching دا عملية تخزين البيانات مؤقتًا في ذاكرة سريعة زي RAM بحيث انك تقدر توصول ليها بسرعة وقت م تحتاجها

ودا بديل من انك كل مره تروح تطلبها من المصدر الأساسي اللي انت مخزن فيه زي Database أو API خارجي معني كدا ان الهدف الأساسي من الـ Caching هو تحسين أداء التطبيقات عن طريق تقليل الوقت المستغرق في استرجاع البيانات المتكررة

طيب ايه هي انواع ال Caching ؟

1. in-Memory Cache
2. Distributed Cache
3. Output Caching
4. Caching Database Queries

اولا in-Memory Cach

البيانات بتتخزن في RAM وده معناه إنها بتبقى جاهزة فوراً، أسرع حاجة ممكن تحصل عليها. لكن لو قفلت ال APP أو حصل Restart اذا كدا ال Data هتطير

طيب ايمته تستخدمه ؟

لما تكون ال Data مش مهمة تحتفظ بيها لفترة طويلة، زي بعض المعلومات المؤقتة اللي مش مشكلة لو فقدتها بعد إعادة التشغيل.

- لكنه يتميز بالسرعة العالية لأن البيانات موجودة في ال RAM

ثانيا Distributed Cache

بيتم تخزين ال Data في System تخزين خارجي مثل Redis أوMemcached

طيب ايمته تستخدمه ؟

لو ال APP اللي انت شغال عليه كبير وبيشتغل على أكتر من Server وعايز تضمن إن الكل يقدر يوصل لنفس ال Data المشتركة بسرعة.

ثالثا Output Caching

بتحفظ نسخة جاهزة من صفحة ويب أو جزء منها بحيث إنك تقدر تعرضها فوراً من غير ما تستدعي كل البيانات وتعيد الحسابات من الأول.

إمتى تستخدمه؟

لما الصفحة بتاعتك مش بتتغير كتير وعايز تعرضها بسرعة للزوار زي مثلا لو عندك صفحة Home بيدخلها الآلاف كل يوم تقدر تحتفظ بنسخة جاهزة منها.

رابعا Caching Database Queries

ودي معناها انك بتحفظ نتيجة ال Queries الخاصه ب Database اللي بتكررها كتير عشان متروحش تعمل نفس ال Query كل مرة.

إمتى تستخدمه؟

لما تكون عندك Queries بتستخدمها بشكل متكرر على نفس ال Data زي List خاصه ب Products مثلا

- بيسرع ف انك تعمل recovery ل Data وبيقلل من التحميل على Database

ليه الـ Caching مفيد؟

- **Faster Performance**

يعني ان اداءه سريع بدال من انك تجيب ال Data من ال Database كل مرة لا انت بتجيبها ف وقت

- **Reduces Database Load**

بتخفف ال الضغط على ال Database ودا من خلال انك بتقلل عدد ال Queries اللي بتروح ل Database وده بيساعد على تحسين الأداء بشكل عام.

وطالما عندي ميزات اكيد برده عندي عيوب او اوقات مينفعش تستخدم فيه ال Caching

- في حالة ان ال Data بتتغير بشكل سريع ومتكرر وقتها ممكن استخدامك ل الـ Caching يعرض Data قديمة.
- إذا كان ال APP بيشتغل على نظام محدود زي mobile devices وقتها خلي بالك من استخدام الـ Caching علشان تتجنب استهلاك ال RAM بشكل كبير.

باختصار كدا هقولك إزاي تختار النوع المناسب؟ 🎯

- **In-Memory Cache**: لو عايز تسترجع ال Data بسرعه بس مش مشكلة لو البيانات راحت لما التطبيق يقفل.
- **Distributed Cache**: لو عندك APP كبير وعايز الكاش يكون متاح لكل السيرفرات.
- **Output Caching**: لو عندك Pages ثابتة وبتزورها كتير.
- **Database Query Caching**: لو عندك Queries بتعملها كل شوية ومش عايز تعيدها كل مرة.

الخلاصة؟

الـ Caching زي مكتبة سريعة بتحفظ فيها أهم الكتب اللي بتستخدمها كل يوم، عشان تقدر تروح لها على طول من غير ما تضيع وقت!

لكن خلي بالك انت المفروض تستخدمه ايمته وايه النوع اللي انت هتستخدمه

الفرق بين ال Stack , heap

تعاله ندخل بقا في ما وراء الكواليس وايه اللي بيحصل فيها ؟؟

بص ي زميلي تخيل كدا معايا انك مثلا بترص اطباق فوق بعض ف انت كدا اما تحتاج تاخد طبق منهم اكيد هتاخد اخر طبق من فوق

يعني اخر طبق حطيته هو اول طبق هتاخده اهو كدا ال Stack برده

انما تخيل كدا انك بتدور على فردة الشراب اللي بتوه منك كل مره ف وسط الاشربه وبتفضل تدور عليها ويبقى حظك حلو لو لقيتها اصلا 😅😅 اهو كدا ال heap

تعاله بقا نتكلم بجد ✍️

يعني إيه Stack ؟

الـ **Stack** دا بيكون مساحة صغيرة في ال Memory بنستخدمها للحاجات اللي بتخلص بسرعة.
زي الأرقام أو النصوص الصغيرة بتتحط في الـ stack ودا ليه؟

لأن الـ Stack بيمشي بنظام آخر حاجة داخلة أول حاجة خارجة (LIFO (Last In First Out

يعني:

- لما تنادي على Function معينه ف ال APP، كل ال variables اللي جواها بتتحط في الـ Stack
- بمجرد ما تخلص ال function وتخرج منها كل الحاجات اللي اتحطت في الـ Stack بتتمسح على طول ي زعيم

تمام كدا فهمنا الدنيا اللى حد ما

طيب تعاله اقولك ع المميزات والعيوب اللي موجوده ف ال Stack

مميزات : الـ Stack سريع جداً ودا ليه ان شاء الله ؟ 🤔

لأن كل ما تخلص من Function أو Variable تلقائي كدا ال Memory اللي كانوا موجودين فيها بتتمسح فوراً.

عيوب : مساحته صغيرة ومحدودة للاسف يعني متقدرش تخزن حاجات كبيرة أو حاجات بتحتاج تعيش لفترة طويلة

طيب إيه هو الـ Heap ؟

بص ي زعيم الـ Heap دا هو جزء تاني من ال Memory بيخزن الحاجات الكبيرة أو الحاجات اللي بتعيش لفترة أطول
يعني مثلا لما بتعمل Object جديد مثلاً باستخدام new ال Object دا بيتخزن في الـ Heap

الفرق هنا بقا إن الـ Heap مش بيمشي بنفس نظام الـ Stack اللي هو LIFO لا خالص لان ال Data بتتحط في أي مكان فاضي في الـ Heap وتفضل موجودة لحد ما ييجي الـ Garbage Collector ويحذفها بمجرد ي يشوف انك مش محتاجها تاني

طيب اصلا ايه هو ال Garbage Collector علشان متسألش : طبعا زي م قولنا ان ال Objects الجديده بتاخد مساحه ف ال Stack and Heap يقوم بقا ال Garbage Collector بيشوف ال Objects اللي مفيش حاجة في الكود بتستخدمها وبعدين يمسحها من ال Memory عشان يوفر مساحة لل Objects الجديدة

طيب ايه هي المميزات والعيوب ؟

مميزات :مساحت ال Heap كبيرة جداً وكمان بيسمحلك انك تخزن Data معقدة زي ال Objects الكبيرة اللي بتعيش لوقت طويل.

طيب وايه هي العيوب : مش منظم زي الـ Stack لا دا شبه الاوضه اللي كل حاجه فيها مبهدله ودا بيتسبب ف انه بياخد وقت أطول عشان يحجز ويفرج عن ال Memory

طيب إمتى استخدم كل واحد فيهم ؟ بص ي زعيم

- الـ **Stack** مناسب للحاجات المؤقتة والسريعة اللي بتخلص بسرعة زي Local Variables
- الـ **Heap** مناسب لما تكون بتعمل Objects كبيرة أو Data بتعيش لفترة أطول زي ال Objects اللي بتتعامل معاها في معظم أجزاء ال APP

الفرق بين GrapgQL And REST API

تخيل كدا معايا انك بتقلب ف التليفزيون لقيت فيلم ف انت سبته علشان عايز تشوف لقطه معينه , بمعنى انك لازم نشوف الفيلم كله علشان تشوف اللقطه دي اهو دا نفس شرح ال REST تخيل بقا انك بدال م تشوف الفيلم كله ع التليفزيون قولت لا انا اروح احسن اجيب اللقطه دي لوحدها من اليوتيوب علشان كدا كدا مش محتاج اشوف الفيلم كله اهو دا نفس شرح ال GraphQL

كدا الدنيا وضحت الى حد ما . تعاله بقا ندخل ف الموضوع

اولا يعني ايه REST Api

REST دا اختصار لـ Representational State Transfer طبعا مقراتش هي اختصار ل ايه اصلا 😅 😅 مش مشكله يلا نكمل

ال REST API دا architectural style بيعتمد على HTTP علشان يحصل Communication بين ال APPs

وال APP اللي بيطلب ال Data بيبعت Request لل Server وبعد كدا ال Server بيرد عليه بال Data اللي هو محتاجها.

يعني أي APP يقدر يتواصل مع ال Server ويطلب منه ال Data عن طريق REST API باستخدام URLs محددة

والفكره الاساسيه هنا ان كل Resource على ال Server ليه URL خاص بيه وبتقدر تتفاعل مع ال Resources دي

عن طريق Standard HTTP زي Post , Get , Put , Delete

كدا فهمنا الدنيا تعالا نتكلم بقا ف شوية مميزات وعيوب ف ال REST

المميزات :

بص ي زعيم من مميزاته انك سهل تستخدمه حاجه سهله كدا وكمان اي Resource انت عايز توصله بيكون ال URL الخاص بيه واضح ومفسر

طيب وايه هي العيوب :

هو عيبه الوحيد انك مثلا لو كنت عايز معلومات معينه عن User هتاخد كل التفاصيل زي الاسم، البريد، العنوان... حتى لو كل اللي محتاجه هو الاسم بس

ايه هو ال GraphQL

GraphQL دا query language و API runtime تم تطويره بواسطة Facebook واللي بيديك مرونة أكتر في الحصول على

ال Data باستخدامك لل GraphQL بتقدر تطلب ال Data اللي إنت محتاجها بالضبط وكمان بتقدر تجيب Data من أكثر من Resource في Request واحد دا غير انه مفيش HTTP Methods زيREST كل حاجة بتتعامل مع طلب واحد بـ GraphQL اللي بيرجع لك اللي طلبته فقط.

تعاله بقا نتكلم عن ال مميزات والعيوب

ميزات :

باستخدامك ل GraphQL بيقدر بيديك Controle أكبر وبيقلل من ال Data اللي بترجعلك ودا بسبب انه بيرجعلك اللي انت محتاجه بس يعني أداء أفضل وشبكة أخف.

يعني مثلا

لو عايز ترجع اسم ال User وال E-mail بس بتقدر تعمل كدا ب ال Query اللي انت بتكتبها من غير م ترجع ال Data كلها

طيب وايه هي العيوب :

ال Queries اللي انت بتكتبها لازم تكون واضحه يعني تكون كاتب اللي انت عايزه بالظبط ودا ممكن يخليها معقده شويه ودا بيخليك تواجه صعوبه لو مش متعود على شكل ال Queries دي

وفي النهايه ملخص الكلام دا ؟؟

REST Api :

تستخدمه لو ال APP بسيط ومش معقد، ده ممكن يكون أفضل ليك.

GraphQL :

بيديك Controle كامل تقدر تطلب اللي انت عايزه بس وده بيكون مناسب أكتر لل APPs المعقدة اللي بتحتاج تجمع Data من Resources مختلفة.

ايه هي ال SOLID Principle

بص ي زعيم دي مجموعة من ال Principles بتساعدك انت ك Developer إنك تكتب كودك بشكل منظم وسهل وبيخليك من السهل انك تطوره او تعدل عليه

خلينا نوضح كل Principle بطريقة بسيطة :

1) Single Responsibility Principle (SRP)

المبدأ هنا ان كل Class في ال App بتاعك لازم يكون ليه Responsibility واحدة وبس

يعني ايه الكلام دا ؟؟ يعني يقصد إن كل Class أو Module لازم يكون ليه مهمة واحدة بس علشان ينفذها كويس يعني مينفعش تجيب نجار علشان يصلح حاجه ف السباكه. الفكرة هنا إنك لما تفصل المهام دا بيخلي تعديل أو تطوير الكود بيكون أسهل وأقل تعقيد

يعني مثلا لو عندك كلاس مثلا اسمه Customer المفروض كدا انه بيتعامل مع ال Data الخاصه ب ال Customer بس يعني مينفعش تضيف فيه كود خاص ب ال Payment كمان لا طبعا المفروض اني اعمل Class تاني علشان ينفذ الجزء دا

طيب وانا ايه اللي هستفيده لو عملت كدا ؟؟ دا هيساعدني اني لو حبيت اعدل حاجة في ال Payment مثلا مش هروح اعدل في كود ال Customer اكيد مش هعمل كدا

2) Open/Closed Principle (OCP)

ودا معناه إنك تقدر تضيف اي Features جديدة للكود من غير ما تعدل في الكود الأساسي عشان الكود يفضل ثابت من غير ميحصل اي Errors ف الكود القديم

يعني مثلا افترض إنك عندك Class بيتعامل مع ال Payment باستخدام credit card بعدين جاتلك فكرة انك تضيف الدفع كمان باستخدام PayPal ف انت كدا بدال م تروح تعدل في الكود الأصلي لا انت تقدر تضيف Class جديد يتعامل مع PayPal والاثنين يشتغلوا من غير ما تعدّل حاجة في الكود الأساسي.

طيب وايه الفايدة كده اللي هتعود عليه اما اعمل كدا ؟؟ قالك انك تضمن ان الكود بتاعك بيكون مستقر وأي إضافة Feature جديدة مش هتبوّظ الشغل القديم اللي شغال كويس

3) Liskov Substitution Principle (LSP)

بيقولك هنا ان اي Sub Class لازم يقدر يحل محل Parent Class من غير ما يكسر اي حاجه ف الكود

يعني المفروض ان اي Inheritance في الكود لازم تكون منطقية وال Objects اللي بتورث من Class معين لازم تقدر تستخدم نفس ال Functions من غير اي مشاكل

يعني مثلا تخيل معايا كدا إن عندك Bird Class ودا فيه اكتر من Function وفيه Function مثلا اسمها Fly تمام كدا

جيت انا حبيت اعمل Class واسميه مثلا Penguin وعايز اخليه يورث من ال Bird Class بس عندي مشكله ان البطريق مش بيطير ف اذا كدا ال Function دي مش هتكون صح كدا بالنسبه لل Penguin ف هقوم انا كدا اعمل New Class واسميه مثلا FlyingBird ودا طبعا هيكون Sub Class علشان يورث من ال Parent Class اللي هو ال Bird كدا اذا خليت Bird Class مفيهوش

ال Fly Function ف كدا بقا عادي اني اخلي Penguin Class يورث من ال Bird Class

وبكدا خليت الكود بتاعي يتوافق مع ال LSP

4) Interface Segregation Principle (ISP)

بيتكلم عن إنك لازم تخلي الـ interfaces بتاعتك صغيرة بدل ما تجمع وظائف كتير في interface واحد ضخم.

يعني إيه الكلام ده؟ ما تخليش الكلاس بتاعك يكون مجبر إنه يطبق حاجات هو مش محتاجها أو مش بيستخدمها يعني بدل ما تعمل Interface كبير بكل الوظائف اللي ممكن أي Class يحتاجها لا انت اعمل Interfaces صغيرة على حسب الحاجة.

طيب ليه ال ISP مهم لما تعمل Interface فيها مجموعة وظائف مش كل ال Customers محتاجينها هيكون عندك مشكلة إن فيه Classes هتضطر تنفذ الوظائف دي حتى لو مش هتستخدمها وده بيخلي الكود بتاعي معقد وصعب التعديل فيه .

تخيل معايا مثلا اني عندي System بيتعامل مع مجموعه من ال Printers ف انا عملت interface اسمه IPrinter وضفت فيه مجموعه من الوظائف اللي موجوده ف الطابعات ... , Fax , Scan , Print طبعا فيه Printers مفيهاش كل الحاجات دي ف مش صح اني اعمل International فيه كل دول لا الصح اني اعمل لكل Feature فيها Interface خاص بيها

طيب ايه الفايدة من كل دا

بص ي سيدي دا هيخلي الكود يكون أخف وأسهل لأن كل جزء بيستخدم اللي هو محتاجه بس.

5) Dependency Inversion Principle (DIP)

هنا بيقولك ان High-level modules مينفعش انها تعتمد على Low-level modules لا طبعا الاثنين المفروض انهم يعتمدوا على abstractions

بمعنى اخر ان مبدأ **DIP** بيقولك إن الكود اللي بيقوم بمهام كبيرة أو معقدة لازم يعتمد على interfaces او abstractions

وده بيخلي الكود مرن وسهل في التعديل من غير ما يحصل اي مشاكل كبيرة.

تعاله نقول مثال بسيط يوضح الدنيا ان شاء الله

تخيل عندي Class مسئول عن انه بيرسل Notifications وبيبعت ال Messages عن طريق Gmail وانا حبيت اضيف Feature تانيه انه يبعت ال Messages دي عن طريق ال SMS وكلاس Notification بيعتمد بشكل مباشر على EmailSender علشان يبعت Notifications. هتضطر تعدل في كود Notification نفسه وده طبعا بيكسر مبدأ DIP

طيب وايه الحل ؟؟ تعاله نقوله ف خطوات سريعه كدا

a) هنعمل Interface اسمه مثلا INotificationSender
b) وبعدين هنعمل Class من ال EmailSender علشان ينفذ ال Interface دا
c) هنعمل Create ل New Class علشان يبعت ال SMS
d) وبعدين Notification Class ي Implement ال Interface اللي اسمه INotificationSender

وكدا خليت الـ Notification بقى مرن جدًا ومش محتاج اعمل عليه تعديل لو غيرت طريقة الإرسال

لأنك بتعتمد على INotificationSender Interface

ملخص الكلام دا ان

Before DIP : ال Notification Class كان بيعتمد على ال EmailSender ودا كان بيخليه من الصعب انك تعدل عليه

After DIP : ال Notification Class بيعتمد على ال INotificationSender Interface ودا بيخلي الكود أسهل في التعديل والاضافه من غير ما تحتاج تغييرات كبيرة في ال APP

ايه هو ال Docker

تخيل إنك بتشتغل على Project كبير وعندك 100 حاجة محتاج تظبطها عشان المشروع يشتغل زي الـ Database , Server , Libraries , Programming Language والمشكله دي بتواجهها كل ما تروح لجهاز تاني تلاقي نفسك مضطر تقعد تظبط كل الحاجات دي من الأول.

كابوس، صح؟

قالك هنا بقا بيجي دور Docker هو عبارة عن Containers بتحط جواها ال APP بتاعك وكل اللي محتاجه عشان يشتغل. يعني بدل ما تقعد تظبط كل حاجة تاني من الصفر على كل جهاز لا انت بتاخد ال Container دي وتنقلها لأي جهاز تاني... وال APP هيشتغل زي ما هو بالظبط من غير اي وجع دماغ.

طيب Docker بيعمل إيه بظبط؟

زي م قولنا فوق ال Docker بياخد الكود بتاعك ومعاه كل الحاجات اللي محتاجها زي Libraries ، Database، Servers ..... ويحطهم كلهم فيContainer . فبدل ما تقعد تعمل الحاجات دي كلها لا انت كل اللي هتحتاج تعمله هو تشغيل ال Container دا وال App هيشتغل على أي جهاز وكانه الجهاز اللي انت كنت شغال عليه متغيرش

طيب ايه هي ال Docker Main Components ؟؟

- Docker Image

ده الأساس وهو عبارة عن حزمة بتحتوي على كل App Filles وال Settings المطلوبة ممكن تعمل Docker Image لل APP بتاعك وتحفظه عشان تستخدمه في أي وقت

- Docker Container

بعد ما بنعمل الـ Image بنحتاج حاجة تشغل الـ Image دي هنا بتيجي فكرة الـ **Containers** ودي عباره عن بيئه تشغيل معزوله علشان تشغل ال App بتاعي من خلال انها بتاخد الـ Image اللي عملتها وبتشغلها في ال Container دا بحيث تقدر تشوف ال APP شغال في عزلة تامة عن باقي ال System

- Dockerfile

الـ **Dockerfile** دا ال File اللي بنكتب فيه كل ال Instructions اللي Docker محتاج يعرفها عشان يبني الـ Image الخاص ب ال APP بتاعي ودا من خلال انه بيقول لـ Docker إيه اللي محتاج يعمله خطوة بخطوة زي ايه مثلا

- ايه النظام اللي هيشتغل عليه زي ال Windows or Linux
- ياخد Copy من الكود الخاص ب ال APP بتاعي
- يعمل Installation للمكتبات اللي ال APP بيحتاجها.
- يحددلي ازاي اشغل ال APP

إيه الفرق بين Docker و Virtual Machines

عارف لما كنت بتستخدم ال Virtual Machines زي VirtualBox أو VMWare دي بتشتغل كأنك مشغل جهاز كامل جوه جهازك بيكون فيه نظام تشغيل كامل وإعدادات بتكون منفصله لوحدها وطبعا الحكاية دي تقيلة على الجهاز وممكن تستهلك موارد كتير.

لكن عندي ال Docker مختلف لانه بيستخدم نظام التشغيل الموجود أساسا وبيشغل ال Containers بتاعته كجزء من النظام الأساسي فالموضوع كدا بيكون خفيف وسريع. ال Containers بتاعة Docker بتشارك نظام التشغيل الأساسي بدل ما تكون نظام تشغيل كامل زي ال Virtual Machines وده بيخليها أسرع وأخف على الجهاز.

طيب بعد كل دا نقدر نلخص ف كام Point كدا ليه ال Docker مهم

ال Docker بيغير قواعد اللعبة لل Developers اللي عايزين يكون شغلهم كفائته أعلى ويضمنوا إن ال APPs بتشتغل في كل بيئة بنفس الشكل ودا لانه

1. بيسهل عليه ان انقل ال APP لاي جهاز تاني

المشكلة الكبيرة اللي بتقابل ال Developers هي إن ال APPs ممكن تشتغل على جهاز واحد ومتشتغلش على جهاز تاني بسبب اختلاف الاعدادات قام Docker حل المشكلة دي تمامًا ودا ازاي !! قالك بمجرد ما تعمل **Container** لل APP بتاعك هيشتغل بنفس الكفاءة سواء على الجهاز بتاعي أو ف اي مكان تاني

2. **Isolation**

يعني ايه ؟؟ تخيل كدا ان عندك اتنين APPs وكل واحد محتاج Library معينة بس كل واحد عايز Version مختلف من ال Library دي.

في العادي طبعا ممكن يحصل تعارض و App واحد يوقف لكن معDocker كل APP بيشتغل في حاوية Container خاص بيه ف كدت مفيش حاجة هتتعارض. كل Container معزول عن التاني وده بيخلي ال Apps تشتغل بشكل مستقل

3. **Higher Efficiency**

ال Containers أخف بكتير من الـ Virtual Machines ودا لانهم بيشاركوا OS Kernel يعني ال Container مفيهوش نظام تشغيل كامل زي الـ VM وده بيخليه أسرع وبكدا ممكن اشغل عدد أكبر من ال Containers على نفس السيرفر مقارنة بالـ VMs

ايه حكاية الـ Access Modifiers ؟

لو تخيلت معايا كدا ان الكود اللي انت بتكبه دا وكأنه بيتك مثلا ف طبيعي انك هتكون عايز تتحكم ف مين يدخل ومين لا واللي يدخل ايه اللي مسرح بيه انه يشوفه وايه لا , باختصار يعني بتتحكم ف خصوصيت بيتك " اللي هو ال Code اللي انت كاتبه " .

اهو دا نفس فكرة ال Access Modifiers , يلا بينا نبدأ نشرحهم واحده واحده

Public

ال Public Modifier بيخلي اي Member سواء كان Class , Method , Variable تقدر انك توصله من اي مكان ف ال Program سواء كان ف نفس ال File او كان ف File تاني ودا طبعا مفيد لو محتاج اني استخدم Method معينه ف اي مكان ف البروجيكت ودا طبعا بيخلي انه مفيش اي Privacy . يعني تقدر تقول ان ال Public مش بتزعل حد زي كانك عامل بيتك مكان عام وسايب اي حد يدخل يشوف اي حاجه فيه

Private

نيجي بقا لل private Modifier ودا بيخلي أي Member سواء كان Class , Method , Variable مش متاح لأي Member غير اللي داخل الـ Class نفسه. يعني لو عندي اي Variable او Function داخل ال Class وعاملها private ف اذا كدا مفيش أي كود خارج ال Class ده يقدر يوصل ليها. ودا طبعا مفيد علشان يمنع اي حد من انه يغير ف الكود دا . يعني تقدر تقول ان ال Private زي الاوضه المقفوله محدش يقدر يستخدم حاجه منها غير اللي داخلها بس

Protected

الـ protected Modifier دا بيخلي الـ Member متاح بس لل Classes اللي بتكون عامله Inherit من ال Super Class يعني لو عندك Class فيه Protected Member ف كدا ال Classes اللي بتورث منه هي الوحيدة اللي تقدر تستخدمه لكن أي كود خارجي مش هيقدر يوصل له . زي ما يكون عندي بيت محدش يقدر يدخله غير اللي معاه مفتاح البيت بس لكن اي حد تاني ميقدرش يدخله

Internal

اما بقا الـ internal Modifier ف دا بيخلي الـ Member متاح بس من جوه نفس الـ Assembly اللي اتعمل فيه. يعني لو عندك Project فيه أكثر من File أو Assembly ف كدا كل ال Functions مثلا اللي معمولة internal مش هيقدر يوصل لها غير ال Files اللي جوه نفس الـ Assembly . زي لو عندي مثلا اجتماع خاص ب مجلس الادراه ف محدش يقدر يدخله غير لو كان من مجلس الادراه بس

Protected Internal

الـ protected internal Modifier دا بقا مزيج بين protected و internal يعني بيكون عندي ال Member

سواء كان class, method, variable بيكون متاح لل Classes اللي بت Inherit منه وللكود اللي في نفس الـ Assembly ده بيديني نوع من المرونة بحيث إني اقدر اخلي ال members متاحة لأماكن محددة داخل ال Project بس

Private Protected

واخيرا الـ private protected Modifier دا بيكون restricts أكتر بيخلي ال Member متاح بس لل Classes اللي ورثت منه وفي نفس الوقت يكونوا جوه نفس الـ Assembly يعني لو عندي كلاس في Assembly معين

وعامل مثلا Private Protected Property مفيش حد يقدر يوصل له إلا لو كان بي Inherit منه

وكمان بيكون جوه نفس الـ Assembly

تقدر تقول مثلا ان عندك غرفة خاصة جوه بيتك ومحدش يقدر يدخلها غيرك انت وعيلتك اللي ساكنه معاك بس

ملخص سريع كدا ع اللي فات

- **Public**: Accessible by everyone, everywhere.
- **Private**: Accessible only within the class.
- **Protected**: Accessible within the class and by subclasses.
- **Internal**: Accessible only within the same assembly.
- **Protected Internal**: Accessible within the assembly and by subclasses.
- **Private Protected**: Accessible by subclasses within the same assembly only.

الفرق بين Normalization and Denormalization

تخيل نفسك ف الاوضه بتاعتك , ف هل انت بتحبها تكون مرتبه وكل حاجه ف مكانها الهدوم ف الدولاب والكتب ع المكتب

ولا انت على الله حكايتك ومش فارق معاك الترتيب وبتحط اي حاجه ف اي حته علشان المهم عندك ان الحاجه تكون قريبه منك وتجيبها من غير م تتحرك . اهو دا بالظبط الفرق بين Normalization and Denormalization

ايه هو ال Normalization

ال Normalization يعني التنظيم ودا معناه اني بقسم ال Data بتاعتي ل Tables صغيره ودا لهدف معين وهو اني احتفظ ب Clean Data وكمان أقل من ال Duplication وكمان بقلل نسبة الاخطاء والكلام دا بيتم على Process معينه (1NF,2NF,3NF)

a) First Normal Form : كل Column بيكون فيه نوع واحد من ال Data يعني مثلا Column ل Names وهكذا وكل Row بيكون Unique يعني ايه برده قالك يعني ميكونش فيه Two Row فيهم نفس الداتا بالظبط

b) Second Normal Form : احذف ال Columns اللي مفيهاش Primary Key يعني مثلا لو عندي Table فيه ال Orders وال Personal Data الخاصه ب ال Customers ف الافضل اني اعمل اشيل ال Customers Data واعملها ف Table تاني لوحدها

c) Third Normal Form : وهنا بقا بعزل ال Columns عن بعضها يعني ان اي Column لازم يكون بي Depend على PK وليس على Another Column , يعني مثلا لو عندي Table خاص ب ال Orders وفيه Column خاصه ب Customers Address وفيه Column تاني اسمه City ودا بيعتمد على Customer Address ف دا كدا مش صح ومن الافضل اني اخلي ال City ف Table تاني

طيب تعاله نشوف الميزات والعيوب ال Normalization

- الميزات

بص ي سيدي استخدام ال Normalization بيساعدني على اني اوفر مساحه ف ال Database ودا بسبب اني بقلل من ال Data المكرره , ال Data عندي بتكون مترتبه , وبيسهل عليه ف التعديل يعني اما بعدل ف Table معين التعديل بينعكس ف باقي ال Tables

- العيوب

بسبب ان ال Tables بقت كتيره دا اثر على ال Queries لانه خلاها بتكون ابطأ وكمان بتكون معقده اكثر لو احتاجت ارجع Data من اكثر من Table

ايه هو Denormalization ؟

ال Denormalization يعني انك بتلغي التنظيم أو انك بتعمل دمج بين ال Tables عشان نحسن من أداء **Queries** بمعنى انك بتضيف بعض ال Data المكررة في Tables عشان Queries تبقى أسرع وأسهل.

تعالَ نوضحها بمراحل زي ما شرحنا الـ:Normalization

a) Adding Redundant Data

يعني انك ممكن تدمج Data من Tables مختلفة في Table واحد. يعني مثلاً بدل ما يكون عندك Table خاص بالـ Customers وجدول خاص بالـOrders ، ممكن تدمجهم في Table واحد بحيث كل Order يحتوي على Customer Data وبالتالي مش محتاج تعمل Join بين Tables عشان ترجع Data دي.

a) Removing Joins

بنحاول نقلل من استخدام الـ Joins في ال Queries، يعني بدال من تقسيم Data على Tables كتير بنحاول ندمجها عشان Data المطلوبة تكون في نفس المكان.

طيب، تعالَ نشوف المميزات والعيوب

المميزات:

ال Performance بيكون اسرع ودا بسبب اني جمعت ال Data ف Tables أقل وكمان ال Queries بتكون ابسط واسهل ودا بسبب ان ال Data اللي انا ممكن اكون محتاجها هتكون موجوده ف Table واحد ف مش هستخدم ال Join

العيوب:

بيزيد عندي التكرار ف ال Data الموجوده عندي ودا بيزود حجم Database وبيخلي كمان التعديل ف ال Data اصعب لاني لو حبيت اعدل ف Data معينه هحتاج اغيرها ف اكتر من Table ودا كمان ممكن يخلي تناقض ف ال Data

خلاصة الكلام دا

- الـ **Normalization** بيساعد على انه يحافظ على Clean Data بدون تكرار، وبيحسن في ان ال Data تكون كامله
- الـ **Denormalization** هدفه هو تحسين أداء الـ Queries عن طريق تقليل الـJoins ، لكنه بيزود التكرار وبيصعب التعديل ف ال Data

---

الفرق بين Authentication and Authorization

ف الاول كدا نقول مثال يسهل علينا الدنيا
بما انك طبعا شخصيه مهمه ف المجتمع ف تخيل انك رايح حفله VIP ف انت اول م بتوصل بيكون فيه جارد ع الباب

بيقولك انت مين ؟؟ "Who" ف دي ال Authentication طيب لو معاك فعلا ال Invitation وقالك اتقضل وبعد كدا دخلت لقيت جارد تاني بيقولك ثواني الحفله ف المكان دا ومش مسموحلك تدخل باقي الامكان اهو دا ال Authorization "what"

تمام يلا نبدأ ف الموضوع بقا

ال Authentication

الـ Authentication هو اول Step ف اي Security System ودا لانه هدفه هو انه يعمل Verifying your Identity يعني بيقولك انه ال Process اللي بتعمل Confirm لهويتك "بيسالك انت مين " ? Who are you لكنه ملهوش اي علاقه ب انت هتعمل ايه بعد كدا . تقدر تقول انه البواب الخاص ب ال Website او ال APP بتاعك 😊

طيب بيشتغل ازاي ؟؟

هنشوفه ف خطوات بسيطه كدا

- Username and Password

دي اكتر طريقه معروفه انك بتبعت الايميل والباسورد وبعدين ال System بيتاكد منهم من خلال انه بيقارنهم ب ال Data اللي متخزنه عندي ف ال Database

- Biometric Verification (Fingerprint or Face ID)

دي طريقه بتستخدم سمات زي fingerprints, facial recognition, or voice ودي طبعا بتكون More Secure من ال الطريقه الاولى لانه دي حاجات Unique مش بتكرر

- Two-Factor Authentication(2FA)

دي خطوة بتكون اضافية بعد ما تدخل Password بتدخل كود اضافي بيوصل على Phone or Email وده بيزود ف ال Security

Examples

زي مثلا انك يكون عندك حساب بنكي ف اما بتطلب انك تعمل Login ليه ف بيطلب منك انك تكتب ال Email And Password وكمان ممكن يكون فيه 2FA علشان يكون More Secure

طيب هو ليه مهم؟
بص ي سيدي الـ Authenticationمهم لأنه بيحمي Accounts الخاصه بيك من الناس اللي ممكن يحاولوا يدخلوا باسمك يعني بيضمن إن اللي بيحاول يعمل Login هو انت مش حد تاني

Authorization

كدا بعد ما ال System يتأكد إنت مين وخلاص عملت Login من خلال Authentication

هنا بقا بييجي دور الـ Authorization ودا اللي بيحدد ايه ال Actions اللي مسموح لك انك تعمله

يعني بيشوف ال Identity and Permissions عشان يحدد إنت تقدر تعمل إيه ومش مسموح لك بايه

يعني الفكرة هنا مش انت مين لا الفكرة نت تقدر تعمل ايه " What You Can Do "

طيب بيشتغل ازاي ؟؟ بص هو بيشوف بعض الحاجات زي

ال Permissions

يعني بيشوف الصلاحيات بتاعتك وهل انت مصرح ليك ب انك تشوف ال Data ولا تعدل فيها ولا تضيف فيها

ال Roles

يعني ال System بيشوف هل انت User , Admin or owner لان طبعا كل Role وليه Permissions معينه

ال Access Controle Lists

ودي بقا بتشوف مين يقدر يشوف ويعدل ف ال Data والكلام دا بناءا على ال Role الخاص بيه

Examples

زي مثلا Facebook Groups لو ال Admin ف هو يقدر يعدل ف ال Posts او يحذفها على عكس ال User العادي ميقدرش بعمل كدا

طيب ليه ال Authorization مهم ؟؟

الـ Authorization بيساعد في حماية ال Data من الوصول اليها من خلال ال User الغير مصرح بهم من انهم يشوفوا ال Data دي , يعني لو عندك أكتر من مستوى لل Permissions بيضمن إن كل user يقدر يشوف ويعدل اللي له علاقة ب ال Role الخاص بيه فقط

خلاصة كل دا

- **Authentication زي ID لانه بيقول لل System (أيوة ده أنا فعلاً)**
- **Authorization بقى بيقول لل System (اه تمام، عرفت انت مين تعاله نشوف ال Permissions الخاصه بيك**

الفرق بين الDapper And Entity Framework

تيخيل انك بتشطب بيتك بس فيه مشكله انك محتار بين اتنين مهندسين واحد منهم قالك هنفذلك البيت بطريقه منظمه وشيك بس هحتاج منك وقت , والتاني معروف عنه ان بيخلص شغله بسرعه جدا بس قالك لازم تشتغل وتتعب معايا علشان ننجزه ,هو دا بالظبط الفرق بين ال Dapper And Entity framework

ايه هو Entity Framework

بص ي سيدي قالك ال EF هو Object Relational Mapper ودا بيتيحلك انك تتعامل مع ال Database كأنك بتتعامل مع Objects في الكود بتاعك بدل ما تكتب SQL بشكل مباشر.

الفكرة الأساسية هنا بقا انك تقدر تشتغل على Data

بأسلوب OOP من غير م تدخل في تفاصيل ال Database تمام كدا فهمنا يعني ايه ال . EF تعاله نتكلم ف شوية حاجات تخصها

📌 طيب ايه هي مميزات الEF

ال EF بيختصر عليه الوقت ومجهود اللي ممكن اضيعهم لو استخدمت ال SQL ودا علشان هيساعدني اني اكتب كود اقل
ال EF بتدعم جيمع ال Databases زي SQL Server ، MySQL، PostgreSQL وغيرهم
ال EF بتتعامل مع الكود بتاعي ك Classes ودا بيساعدني ف اني اخلي الكود منظم ومفهموم اكتر
واخيرا اقدر اني اكتب ال Queries دي باستخدام Linq

📌 طيب وايه هي العيوب

عندي ال Performance مش بيكون الافضل في بعض الاوقات ف ال Projects الكبيره اللي بتكون محتاجه Performance عالي وخصوصا ف ال Complex Queries

تفاصيل ال SQL بتكون مخفيه ودا مش بيكون مفيد اما بكون محتاج Controle كامل ف ال Queries بتاعتي

📌 طيب ايمته استخدم الEF

لو ال Project بتاعك محتاج Development سريع ووقت التنفيذ مش هو العامل الأساسي
لو انت بتحب تشتغل في بيئة OOP وتحب تكتب كود مفهوم ومنظم
لو عايز تشتغل على حاجه سهلة ومش عايز تتعامل مع SQL بشكل مباشر

ايه هوDapper ؟🧐

ال Dapper هو Micro-ORM يعني هو Library خفيفة بتساعدك تتعامل مع Database بطريقة بسيطة وسريعة

الفرق هنا انه بدل ما تتعامل مع Objects في الكود زي الـEF

لا Dapperبيديك حرية كتابةSQL Queries مباشرة

الفكرة الأساسية إنه يتيحلك تواصل أسرع مع Database بدون تحميل زائد علىSystem

📌 ايه هي مميزاتDapper ؟

أول حاجة Dapperمعروف بسرعته العاليه. لما تيجي تستخدمه هتحس إن Queries بتتنفز بسرعة لأنه مش بياخد وقت كبير في معالجة Data يعني لو عندك Project محتاج سرعة في Performance ف Dapperهو الخيار المناسب.

ثاني حاجة انه خفيف جدًا. مش محتاج تحميل Libraries ضخمة أو إعدادات معقدة كل اللي عليك انك تكتب SQL بشكل مباشر. ده بيخليك مرتاح لإنك بتتحكم فيDetails

ثالث حاجة ممكن تستخدمه مع أي نوع من Databases يعني سواء كنت بتشتغل علىSQL Server ، MySQL، أوPostgreSQL ، ال Dapperهيساعدك في كل ده. وده بيساعدك في توسيع نطاق المشروع بتاعك بسهولة.

📌 طيب إيه هي العيوب؟

لكن زي أي حاجةDapperعنده بعض العيوب

أولاً لو كنت بتحب الـ OOP وعايز تتعامل مع Objects بشكل سلس

Dapperمش هيكون الأفضل ليك لأنه مش بيقدم نفس مستوى السلاسه اللي بتوفرها مكتبات زي.Entity Framework

ثانيًا بما إنك هتكون محتاج تكتب SQL بشكل يدوي، فده هيحتاج منك معرفة جيدة ب Queries وكيفية تنفيذها بشكل صحيح وده ممكن يكون تحدي لبعضDevelopers

📌 طيب ايمته استخدمDapper ؟

لو كنت محتاج سرعة في Performance وبتتعامل مع كميات كبيرة منData

لو بتحب تكون متحكم بشكل كامل فيSQL Queries الخاصة بك.

لومش محتاج تشتغل في بيئة OOP بشكل كبير وبتفضل كتابة كود مباشر

Features	Entity Framework	Dapper
Performance	بيكون بطيء ف بعض الاوقات	سريع جدا طبعا
Ease Of Use	سهل ف الاستخدام	لازم تكون فاهم SQL
Flexibility	بيكون فيه Abstractions بتحكمني	بيكون فيه مرونه ف كتابة ال Queries
Code Complexity	بيكون الكود اللي بكتبه قليل	الكود بيكون اكتر بس بتقدر تعمل Controle كامل عليه
Linq Support	اه بيدعم طبعا ال Linq	مش بيدعم Linq

ايه هو ال ModelBinder And ModelState

ف الاول كدا تخيل انك بتملأ Form معينه ف Website بتدخل Data مثلا زي ..Name , Email , Phone وبعدين بضغط Submit

السؤال هنا بقى إيه اللي بيحصل وراء الكواليس عشان ال Data دي توصل وتتأكد إنها صح ؟

هنا ييجي دور ModelBinder , ModelState.. تعاله نتكلم بقا ايه هما ودورهم ايه

ModelBinder

الModelBinder دا المسئول عن تحويل ال Data اللي جاية من Userزي Form أو Query String ل Objects علشان استخدمها في الكود. ببساطة بياخد القيم اللي User دخلها ويربطها بالموديل اللي انا معرفه في الكود.

إزاي بيشتغلModelBinder ؟

بص ال ModelBinder بيقرأ ال Data اللي جايه من ال HTTP Request يعني بيتعرف على الFildes ويشوف ال Data اللي جايه مع ال Field دا وبعد كدا بيربط ال Filed دي ب ال Model اللي انا معرفه ف الكود عندي لو لقي تطباق بين اسم الFiled ف ال Request واسم ال Properties ف ال Model عندي , بعد كدا يقوم واخد ال Variable اللي جايه من ال Request ويحطها ف ال Property دي

أنواع:ModelBinder

في ASP.NET MVC، فيه نوعين أساسيين من الـ ModelBinder.

الأول هو الـ Default ModelBinder، وده اللي بيشتغل تلقائيًا على معظم الحالات العادية زي Numbers , Dates , Text من غير ما تحتاج تعمل حاجة إضافية. يعني لو Form بسيطه والData اللي User بيدخلها تكون مباشرة الـ Default ModelBinderهيقدر يتعامل معاها تلقائي.

النوع التاني هو Custom ModelBinder، وده بتلجأ له لما يكون عندك حالات خاصة أو Models معقدة بتحتاج طريقة مخصصة للربط. يعني لو عندك موديل مكون من أكتر من نوع Data أو فيه Objects متداخلة أو Method معينة لتمثيل Data، هنا بتقدر تكتب Custom ModelBinderعشان تتحكم في طريقة معالجة وربط Data دي.

أهمية ModelBinder

ModelBinder بيوفر عليك وقت ومجهود كبير، لأنه بيتولى عملية تحويل وربط Data تلقائيا. بدل ما تقعد تستخرج القيم يدوي من الـ HTTP Request وتحطها في الكائنات بنفسك، الـ ModelBinderبيقوم بالمهمة دي بطريقة ذكية. كمان بيخليك تقدر تتعامل مع أنواع Data مختلفة، سواء كانت بسيطة زي النصوص والأرقام، أو مركبة زي Objects المتداخلة. ده بيدي مرونة وسهولة في التعامل مع بيانات Users اللي بتتبعت للAPP بتاعك.

ModelState

ال ModelState دا المسئول عن انه يتحقق من صحة Data اللي جاية من User في الـ Form or Query string قبل ما يتم استخدامها في الكود. يعني هو ببساطة بيعمل فحص سريع للـ Dataاللي بيدخلها User ويتأكد إنها صحيحة ومطابقة للشروط اللي انا معرفها في Model

طيب ازاي بيشتغل ModelState ؟

لما Data توصل للـ Controller الـ ModelState بيبدأ يفحص كل Field من Fields اللي جاية من Request، ويتأكد إنها تمام. لو فيه أي خطأ، زي مثلًا لو User دخل Text في Field مفروض يكون Number، أو لو نسي يدخل Value كانت Requierd

الـ ModelStateبيضيف الأخطاء دي للقائمة الخاصة بيه.

يعني لو كل Data صحيحة الـ ModelStateبيكونValid ولو فيه مشاكل، بيكونInvalid وبيحتوي على تفاصيل الأخطاء.

أنواع Validation فيModelState

Automatic Validation بعض ال Validation بيتم تلقائيًا زي التأكد من توافق الأنواع مثلاً لو مدخل String Data ف Field المفروض انه يستقبيل String Data

Custom Validation ممكن اني اضيف شروط مخصصة في Model باستخدام **Data Annotations** زي [Required] ,[StringLength]

طيب ايه هي أهمية ModelState

ال ModelState مهم جدًا لأنه بيضمن إن Data اللي جاية من Users صح وآمنة قبل ما تستخدمها في الكود. ده بيحميك من الأخطاء المحتملة أو Data اللي مش صحيحه اللي ممكن تسبب مشاكل في APP بتاعي لو Data مش صحيحة، تقدر تبعت رسالة User بالأخطاء، عشان يصلحها قبل ما ترسل Request مرة تانية.

ملخص:

ModelBinder بياخد Data اللي جاية من Form ويربطها بالـ .**Model**

ModelState بيتحقق من صحة Data، يعني مثلا لو فيه أخطاء اما بعمل Login زي إن Username أو Password مش صح ، بيتم إظهار رسالة خطأ لل User ولو Data صحيحة، ال Login بيتم بشكل طبيعي.

HTTP Status Codes Explained

ف البدايه خليني اقولك تخيل انك شغال على بروجيكت وبدأ يظهرلك اكواد HTTP Requests كل كود وليه رساله عايز يقولهالك , تعاله بقا نفهم ايه خكاية كل كود

1xx: Informational Responses

100 Continue

The client should continue with its request.

101 Switching Protocols

The server understands and is willing to comply with the client's request to change protocols.

102 Processing (WebDAV)

The server has received the request and is processing it, but no response is yet available.

2xx Success

200 OK

The request was successfully processed by the server, and the expected data is returned.

201 Created

The request was successful, and a new resource was created as a result.

204 No Content

The request was processed successfully, but there is no content to send back.

3xx Redirection

301 Moved Permanently

The resource has been moved to a new URL permanently.

302 Found

The resource is temporarily located at a different URL, but the original URL should still be used in future requests.

304 Not Modified

The resource has not been modified since the last request, so the cached version can be used.

4xx Client Errors

400 Bad Request
The server could not understand the request due to invalid syntax.

401 Unauthorized
Authentication is required to access the resource, but it was either not provided or is invalid.

403 Forbidden
The server understood the request, but it refuses to fulfill it due to insufficient permissions.

404 Not Found
The server could not find the requested resource.

405 Method Not Allowed
The request method is not supported for the targeted resource.

409 Conflict
The request conflicts with the current state of the server or resource.

429 Too Many Requests
The user has sent too many requests in a given amount of time (rate-limiting).

5xx: Server Errors

500 Internal Server Error
The server encountered an unexpected condition that prevented it from fulfilling the request.

502 Bad Gateway
The server, while acting as a gateway or proxy, received an invalid response from the upstream server.

503 Service Unavailable
The server is temporarily unable to handle the request, usually due to overload or maintenance.

504 Gateway Timeout
The server, while acting as a gateway or proxy, did not receive a timely response from the upstream server.

Value Types & Reference Types

Value Types

Value Types هي الأنواع التي تخزن القيم مباشرة في Memory. يتم تخصيص مكان لل Data في Memory عند إعلان Variable، وعند نسخ قيمة من Variable آخر، بيتم إنشاء نسخة مستقلة.

- Examples

int, float, double, char, bool

- **آلية العمل :**

عند تمرير **Value Type** إلى دالة، يتم تمرير نسخة من القيمة. لذلك، أي تغييرات تتم على النسخة داخل الدالة لا تؤثر على القيمة الأصلية.

Reference Type

Reference Types، على الجانب الآخر، تخزن عنوان Memory الذي يشير إلى Data. وبالتالي، عند نسخ **Reference Type**، يتم نسخ العنوان فقط وليس البيانات نفسها.

- Examples

Classes , Arrays , String

- **آلية العمل :**

عند تمرير **Reference Type** إلى دالة، يتم تمرير عنوان Data. لذا، أي تغييرات تتم على Object من خلال ذلك العنوان تؤثر على Object الأصلي.

باختصار

Value Type	Reference Type
• تُخزن البيانات في .(stack) • كل نسخة من البيانات مستقلة. • أسرع في الأداء عند التعامل مع كميات صغيرة من البيانات.	• تُخزن البيانات في .(heap) • جميع النسخ تشترك في نفس الموقع في الذاكرة. • أكثر مرونة، لكن قد تكون أبطأ من حيث الأداء بسبب إدارة الذاكرة.

Design Patterns

باختصار، الـ Design Patterns هي حلول جاهزة لمشاكل متكررة في التصميم البرمجي. بدل ما تعيد اختراع العجلة في كل مرة تواجه فيها نفس المشكلة، تقدر تعتمد على الـ patterns لتوفير الوقت وضمان حلول نظيفة وكويسة.

ليه الـ Design Patterns مهمة؟

بدل ما تحل المشكلة من الصفر كل مرة، الـ Design Patterns بتوفرلك إطار عمل جاهز. تخيل إنك تبني بيت؛ بدل ما تخترع طريقة جديدة لكل عنصر، بتستخدم تصميمات أثبتت كفاءتها مرارًا. بتساعدك كمان على تحسين جودة الكود، تسهيل الصيانة، وتعزيز التعاون بين فرق العمل.

- خلينا نلقي نظرة على الفئات الثلاثة الرئيسية للـ **Design Patterns**:

Creational Patterns

واحنا شغالين ف Projects بنحتاج نعمل Objects عشان ننفذ مهام معينة. لكن ساعات طريقة إنشاء Objects بتكون معقدة هنا بيجي دور **Creational Patterns** وهي ببساطة طرق ذكية ومبتكرة تخلينا نعمل Objects بطريقة أسهل وأفضل.

خلينا نوضح ال Patterns دي بطريقه سهله ان شاء الله

- Singleton Pattern

الـ Singletonعشان نضمن إن مفيش غير نسخة واحدة من Object معين. يعني كأننا بنقول لل Program هتشتغل بنسخة واحدة بس من Object ده طول الوقت.

Example

عندك مكان فيه مدير واحد بس ومينفعش يبقى فيه أكتر من مدير

- Factory Method Pattern

الـ Factory Method بيساعدك تعمل Objects بدون ما تعرف التفاصيل الدقيقة لإزاي اتعملت. انت بس بتطلب Object وهو بيجي لك جاهز.

Example

عندك مصنع، وكل مرة بيخرج لك منتج جديد حسب الطلب. بس إنت مش محتاج تعرف تفاصيل إزاي المنتج اتعمل، المهم إنك تطلبه والمصنع يسلمك المنتج الجاهز

- Builder Pattern

الـ Builder Pattern بنستخدمه لما يكون Object اللي عايزين نعمله معقد وفيه تفاصيل كتير. بدل ما نعمل Object مرة واحدة، بنبنيه خطوة بخطوة كل مرة بنضيف جزء لحد ما يكتمل.

Example

زي انك بتبني بيت وبتعمله جزء جزء.

- Prototype Pattern

الـ Prototype ببساطة بيخليك تعمل نسخ من Object اللي عندك بطريقة سريعة بدال م تبدأ من الصفر علشان تعمل واحد تاني .

- **Abstract Factory Pattern**

الـ Abstract Factoryبيساعدك تعمل عائلات من Objects المرتبطة ببعض بس من غير ما تقلق من إزاي كل Object اتعمل.

Example

كأنك عندك مصنع بيعمل أكتر من نوع من المنتجات وكل منتج له طريقته الخاصة

Structural Patterns

لما بنبني Systems كبيرة، بنحتاج نعرف إزاي Objects هتشتغل مع بعض وتترابط بشكل سليم قالك **Structural Patterns** .
بتساعدنا نرتب Objects دي بطريقة منظمة بحيث نتجنب أي تعقيد ونخلي الكود بتاعنا أسهل في الفهم

- **Adapter Pattern**

الـ Adapterبيساعدك تعمل Object يتوافق مع نظام مختلف بدون ما تغير في الكود الأساسي. هو ببساطة محول بيخلي Objects تتفاهم مع بعضها

- **Decorator Pattern**

الـ Decoratorبيضيف خصائص جديدة لل Objects بدون ما تغير Object الأصلي. يعني كأنك بتضيف طبقات جديدة لل Object من غير ما تعدل عليه.

- **Facade Pattern**

في الكود لو عندك System معقد فيه Objects كتير، بتعمل Facadeعشان تبسط التعامل معاه وتخفي التعقيدات اللي ف الكود.

Example

زي لما تروح لشركة سياحة مثلاً وهي ترتبلك كل حاجة في رحلتك بدل ما تتعامل مع الفنادق والمطاعم والأنشطة كل واحد لوحده لا

الـ Facadeبيبسط الأمور ويوفرلك وجهة واحدة تتعامل معاها

- **Proxy Pattern**

الـ Proxyزي وسيط أو بوابة بتتحكم مين يقدر يوصل لObject معين. ممكن تستخدمه لو عايز تحدد مين يقدر يشوف أو يعدل في Objects أو حتى تقلل الحمل على Object تاني بانك تحط وسيط بينهم.

Behavioral Patterns

الـ Behavioral Patternsهي Patterns بتحدد سلوك Objects أو طرق تفاعلهم مع بعض عشان تحقق مهام معينة. الPatterns دي بتركز على توزيع المسؤوليات بطريقة تخلي التواصل بينهم سلس وفعال، وبتعالج مشاكل زي طرق تبادل الرسائل بين Objects

- **Observer Pattern**

لو عندك Object بيتغير باستمرار وعايز Objects تانية تتابع التغييرات دي، بتستخدم الـ Observer Patternعشان لما يحصل تغيير كل Observers يتبلغوا أوتوماتيك

- **Strategy Pattern**

الـ Strategy Patternبيسمحلك تغير strategy بتاع Object معين ديناميك من غير ما تعدل في الكود الرئيسي لل Object نفسه

- Command Pattern

لو عندك عملية معينة وعايز تنفذها في وقت لاحق أو تلغيها، الـ **Command Pattern**بييجي هنا عشان ينظم العملية دي.

- Chain of Responsibility Pattern

بيخلي Requests تمر عبر سلسلة من Objects لحد ما حد منهم يتعامل معاها

Example

تخيل عندك فريق دعم فني أول حد في الفريق مش عارف يحل مشكلتك، فيقوم يحولها للي بعده وهكذا لحد ما المشكلة تتحل. الـ Chain of Responsibilityبيعمل نفس الفكرة، بيحول Requests أو العمليات بين مجموعة Objects لحد ما واحد فيهم يحل المشكلة.

- State Pattern

لو عندك Object بيغير حالاته بشكل متكرر (مثلاً، حالة تشغيل أو إيقاف)، بتستخدم الـ **State Pattern**عشان تنظم السلوك ده بناءً على حالته.

- Mediator Pattern

الـ Mediator Patternلما يكون عندك مجموعة Objects بتتواصل مع بعض بشكل مباشر، وعايز تقلل الاعتماد المتبادل بينهم وتنظم عملية التواصل

Example

تخيل لو عندك 10 موظفين في شركة وكل واحد عمال يتواصل مع التاني بشكل مباشر، الدنيا هتبقى زحمة جداً.

فبتجيب مديرMediator اللي هو الشخص اللي بيتواصل الكل معاه عشان ينظم التواصل

Database Keys

استنى ي زعيم متعملش Scroll وخليني اكلمك عن **Database Keys** بطريقة بسيطة ومفسرة عشان تبقى فاهم الدنيا تمام

الـ **Keys** دي زي المفاتيح بتاعت البيت بتفتحلك الباب على Data معينة وتضمنلك إنك ما تغلطش في حاجات زي التكرار أو الترابط بين Data

طيب يلا نبدا شرحهم من غير كلام كتير

اولا Primary Key

ال Primary Key دا حجر الاساس لاي Table لان دا Column اللي بيميز كل Row في Table عن غيره بشكل Unique بيتمثل دوره في ضمان أن كل Record فريد من نوعه ولا يسمح بأن يكون هناك تكرار أو أن يكون Row قيمته NULL

أهمية Primary Key

- **Uniqueness** يضمن أن كل Record في Table يكون مميز وفريد.
- **Non-nullability** لا يمكن أن يحتوي على قيم Null لأنه لازم يكون معرف لكل Record
- **Performance** بيساعد في تحسين أداء Database وقت م بيتم البحث عن Data أو استرجاعها

Example

لو عندك Employees Table ف كدا EmployeeID اقدر استخدمه ك Primary Key

ثانيا Unique Key

ال Unique Key شبه Primary Key من حيث أنه يضمن أن القيم في Column تكون Unique لكنه يسمح بوجود قيم ب Null

بستخدمه اما بكون محتاج ضمان عدم تكرار Values في Column محدد , لكنه مش بيكون ليه نفس ال Constrains اللي ف ال PK , وكمان ممكن يكون عندي اكتر من Unique Column على عكس ال PK

أهمية Unique Key

- **Data Uniqueness** يضمن عدم تكرار Values في Column
- **Flexible Null Handling** يسمح بوجود Null Values على عكس PK

Example

لو فيه Table عندي وفيه مجموعهمن ال Columns منهم PhoneNumber ف كدا ممكن اعملها Unique علشان ميحلصش تكرار

ثالثا Foreign Key

ال Foreign Key هو اللي بيوصل بين اتنين Tables. دوره الأساسي هو إنه ينشئ Relations بين الجداول، وبيكون دايمًا مرتبط ب PK في Table تاني. وده بيساعد في الحفاظ على **Referential Integrity** اللي هو تكامل Data بين Tables

أهمية FK

- **Referential Integrity** يضمن أن Relation بين Tables تكون صحيحة ومتكاملة، بحيث لا يمكن إضافة سجل في الجدول المرتبط لو PK مش موجود في الجدول التاني.
- **Data Integrity** يمنع حدوث تغييرات غير صحيحة أو حذف بيانات مرتبطة بدون تنسيق مسبق.

Example

لو عندي اتنين Tables (Employees , Department) ف طبيعي ان كل موظف يكون ليه قسم تابع ليه ف كدا هحتاج اعمل ربط بين الجدولين دول باستخدام DepartmentID ك Fk

رابعا Composite Key

ال Composite Key عبارة عن مجموعة Columns بتكون مع بعضها Unique Key. بنستخدمه لما مايكونش فيه Column واحد يقدر يميز Records بمفرده، بس لما نجمع بين أكتر من Column مع بعض، يقدروا يكونوا Unique Key , وممكن يكون اكتر من عمودين على حسب الحاجه

أهمية Composite Key

- **Complex Uniqueness** يستخدم في الحالات التي تتطلب مجموعه من Values علشان يكون كل Record فريد
- **Flexibility** يوفر القدرة على التعامل مع Tables أكثر تعقيدا تحتوي على Relations متعددة

Example

لو عندي Table خاص ب حجوزات طيران اسمه FlightBooking ممكن استخدم FlightNumber , FlightDate علشان احدد بشكل فريد

خامسا Candidate Key

الـ Candidate Keyهو عمود أو مجموعة أعمدة ممكن تستخدم كـ Primary Key، لكن مش بالضرورة يكون تم اختياره. يعني، كل Primary Keyهو Candidate Key، لكن مش كل Candidate Keyهو Primary Key

أهمية Candidate Key

- **Multiple Choices** يوفر مرونة في اختيار PK يعني ممكن اختيار واحد من ال Keys المرشحة.

Example

لو عندي Table فيه اتنين Column (NationalID , Email) وكل واحد فيهم Unique ف كدا ممكن استحدم حاجه منهم ك PK

سادسا Super Key

ال Super Keyهو أي مجموعة من Columns اللي ممكن تميز كل Rowفي Table بشكل فريد.
بمعنى تاني أي مجموعة من Columns اللي ممكن استخدمها لتحديد Recordواحد تعتبر Super Key
كل Primary Keyهو Super Key، ولكن ليس كل Super Keyهو Primary Key.

أهمية Super Key

- Uniquenessيضمن أن كل Recordفي الجدول فريد.
- Flexibility يتيح الفرصة لتحديد أكثر من مجموعة من Columns يمكن أن تعمل كUnique Key

Example

لو عندي Employee Table وعندي فيه مجموعه من ال Columns (EmployeeID , NationalID , Email) **ف كدا ممكن اعتبرهم كلهم مع بعض او بشكل منفرد ك Super Key** لأنها تميز كل Record

سابعا Alternate Key

ال Alternate Key هو أي Candidate Keyلم يتم اختياره ليكون Primary Key
بمعنى تاني لو عندي مجموعة Candidate Keys يتم اختيار واحد منه كPK والباقي يصبح Alternate Keys

أهمية:Alternate Key

- **Backup Option** إذا تم تغيير **PK** يمكن استخدام **Alternate Key**كبديل.
- **Uniqueness** زي **PK** يضمن أن القيم في Column فريدة ولكن لم يتم استخدامه كPK

Example

لو عنديUsers Table فيه مجموعه من ال Columns (UserID , Email , NationalID) وانا خدت الUserID كPK

ف كدا باقي Columns هيكونوا Alternate Key

الملخص النهائي:

- **Primary Key**:يضمن التفرد لكل سجل، ولا يسمح بالقيم الفارغة.
- **Unique Key**:يضمن أن القيم في العمود فريدة، مع السماح بقيمة فارغة واحدة.
- **Foreign Key**:يربط بين الجداول، ويحافظ على تكامل العلاقات.
- **Composite Key**:يُستخدم لتحديد السجلات باستخدام أكثر من عمود.
- **Candidate Key**:هو أي عمود أو مجموعة من الأعمدة التي يمكن أن تكون مفتاح أساسي.
- **Super Key**:أي مجموعة من الأعمدة يمكن أن تميز كل سجل بشكل فريد.
- **Alternate Key**:هو أي **Candidate Key**لم يتم اختياره ليكون **Primary Key**.

Cookies, Session, Local Storage

مفكرتش قبل كدا ازاي اما بتسجل دخول ل Website هو ازاي بيفضل فاكرك ؟؟

هنا بقى يظهر دول Cookies , Session , Local Storage لكل حاجه فيهم وظيفة , تعاله نعرف ايه وظيفة كل حاجه فيهم

Cookies

الـ Cookiesعباره عن Files نصية صغيرة جدًا بتتخزن على جهازUser بتساعد Websites في تتبع Users وتخزين Data بسيطة عنهم. مثلاً لما تسجل دخولك على موقع، الـ Cookie بتحتفظ بمعلومات تسجيل الدخول علشان متحتاجش تدخلها تاني في كل مره تزور فيها Website

- **طيب ازاي بيشتغل**

كل مرة تدخل فيها على Website، ال browser بيبعث الـ Cookies اللي مخزنها للموقع. الموقع بعد كده يستخدم Data دي. يعني لو بتزور موقع E-Commerce ، الكوكيز ممكن تحفظ Products اللي كنت بتتصفحها.

- **انواع Cookies**

Session Cookies

بتحفظ مؤقتا وتختفي بمجرد ما تقفل Browser

Persistent Cookies

بتفضل موجودة لفترة محددة حتى لو قفلت Browser زي Language preferences

- **Advantages**

- اقدر اخزن الDataلفتره طويله

- بيحتفظ ب ال Settings اللي انت حفظتها زي Language or Dark Mode

- بتساعد Websites في تحليل User Behavior يعني لو كنت على **video streaming platform** زيYouTube

الCookies بتخزن نوع Videos اللي بتشوفها كتير,وتساعد الWebsite ف انه يعرضلك **personalized suggestions** للفيديوهات الشبيهه للي انت بتشوفها

- **Disadvantage**

- حجمها محدود حوالي 4 KBلكلCookie فمش ممكن تخزن Data كبيرة

Session

الـ Session Storageهي نوع من أنواع التخزين المؤقت للData اللي بتتخزن على جهاز الـ User، بس بتكون مرتبطة بـ sessionوحدة

يعني Data بتفضل موجودة طول ما الـ Userشغال على Website لكن بمجرد ما يقفل الـ Browser، البيانات بتتمسح تلقائي.

- **طيب ازاي بيشتغل؟**

لما تفتح Website، الـ Session Storageبتخزن Dataمرتبطة بالSession الحالية. كل صفحة من Website ممكن توصل للData دي، لكن أول ما تقفل الـ Browserأو حتى tab الData دي بتتمسح تلقائيًا. ده مفيد لو عايز تحفظ Data مؤقتة

- **Advantages**

- الSecurity افضل من الCookies لان Dataمش بتتخزن على Server لا دي بتكون على جهاز User وطول فترة Session فقط

- الData Size اكبر من Cookies يعني ممكن تخزن Data لحد 5MB تقريبا

مفيد ف التعامل مع Data مؤقته مرتبطه ب الSession الحاليه زي تفاصيل Search

- **Disadvantage**

-الData بتتمسح بمجرد م الUser يقفل ال Browser او Tab

-مش بيكون مناسب لتخزين Data هامه لفتره طويلة

Local Storage

الـ Local Storageمشابهة لـ Session Storage، لكن الفرق الرئيسي هو ان Data بتفضل محفوظة على جهاز الـ Userحتى بعد ما يقفل .Browserيعني تقدر تحتفظ بالData لفترة غير محددة لحد ما يتم مسحها يدويا أو عن طريق الكود.

- **طيب ازاي بيشتغل؟**

الـ **Local Storage**بتستخدم لتخزين Data ممكن تحتاجها بشكل دائم أو لفترة طويلة. كل مرة تفتح فيها الـ **Website**، البيانات دي بتفضل موجودة ومتاحة لأي صفحة على Website للوصول إليها، حتى لو قفلت الـ **Browser**وفتحته تاني. ده بيكون مفيد لحفظ تفضيلات الUser زي Language or Mode

- **Advantages**

- تقدر تخزن Data لفتره طويله حتى لو قفلت الBrowser وفتحته تاني

- الStorage اكبر من Cookies and Session يعني حوالي 5-10 MB لكل Website

- سريعه في استرجاع الData لانها بتكون متخزنه على الجهاز الخاص بUser

- **Disadvantages**

- مش بيكون مناسب لتخزين Data مهمه زي Passwords Or Payment information

- الUser ممكن يحذف الData من الاعدادت الخاصه بالBrowser

JWT

تخيل إنك داخل على Website وعايز تفضل عامل Login حتى بعد ما تقفل الصفحة أو تغير Browser. إزاي Website يقدر يتعرف عليك من غير ما يسألك تدخل Data كل مرة؟ الحل هو **JWT (JSON Web Token)**، وهي واحدة من أكثر الطرق شيوعًا لتأمين المصادقة وتبادل Data بشكل آمن بين User وServer

ايه هو JWT

JWT دا اختصار لـ **JSON Web Token**، وهو **Token**بيستخدم لتبادل Data بطريقة آمنة. الفكرة إن الـ Token بيحتوي على Data الخاصه بـUser أو معلومات الصلاحياتPrivileges بشكل مشفر بحيث الـ Client المتصفح مثلاً يقدر يبعت الـ Token مع كل Request يتبعت للServer وبعدين Server يتأكد من صحة الـ Token قبل ما يسمح للUser بالوصول للData أو الخدمات المطلوبة.

طيب ايه بيخلي JWT مميز

اللي بيخلي الـ JWT مميز إنه بيكون Self-contained

يعني إيه؟ يعني كل Data اللي Server بيحتاجها عشان يتأكد من صلاحية الـUser موجودة جوا الـ Token نفسه. Server مش محتاج يخزن User Info في Database أو يدير Sessions وده بيخليها خيار ممتاز للAps اللي عايزة تتعامل بشكل غير معتمد على Stateless

JWT Structure

الـ JWT بيتكون من ثلاثة أجزاء رئيسية مشفرة باستخدام Base64، وكل جزء منهم له وظيفة محددة لتأمين وحماية Data اللي بنبعتها بين Client وServer الأجزاء دي هي

Header , Payload , Signature خلينا نشوف كل جزء بالتفصيل:

Header [1]

الـ Header هو الجزء الأول من الـ JWT وبيشبه عنوان الرسالة اللي بتبعتها. بيحدد نوع الـ Token اللي بنتعامل معاه وطريقة التشفير المستخدمة.

باختصار

تخيل إن الـ Header هو الغلاف الخارجي لرسالة. بيوضح لنا نوع التشفير اللي هنعمله للرسالة نفسها.

Payload [2]

هنا بيتم تخزين البيانات اللي عايز تبعتهاClaims

الـ Claims ممكن تكون Data زي الـUser ID ، name، Roles، أو أي Data تانية محتاج توصلها للServer.

مهم نلاحظ إن Data دي مش مشفرة لكن بس مشفرة باستخدامBase64

يعني لازم تتجنب تخزين أي Data حساسة هنا زي Password

في الغالب الـ Payload بيحتوي على نوعين من الـClaims:

1. **Registered Claims** زي iss اللي هو Issuer أو مصدر الـToken وExpiration أو مدة صلاحية الـToken
2. **Custom Claims** ودي البيانات اللي بنضيفها حسب احتياجاتنا زي **User ID** أو **Roles**

Signature [3]

الـ Signature هو الجزء الأهم اللي بيحمي الـ JWT من التلاعب.

الـ Signature بيتم إنشاؤه عن طريق دمج الـ Header والـ Payload مع Secret Keyباستخدام خوارزمية التشفير اللي اخترناها في الـ Header زيHMAC-SHA256 والهدف منه هو التأكد إن الـ Token ده موثوق ومش متغير.

باختصار

تخيل الـ Signature زي ختم الأمان اللي بتحطه على رسالة أو طرد مهم. بيأكد للطرف اللي هيستلمه إن الرسالة جت من مصدر موثوق وإنها ما اتغيرتش في الطريق.

Conclusion:

- الـ **Header** بيحدد نوع التشفير.
- الـ **Payload** بيحتوي على البيانات (Claims) اللي بتحدد هوية المستخدم.
- الـ **Signature** بيضمن إن البيانات موثوقة ومش متعدّل عليها.

Advantages

Stateless

الـ JWT مش بيتطلب تخزين معلومات Sessions على Server. ده بيخلي APP أخف وأسرع خصوصاً مع APPs اللي بتحتاج تتعامل مع عدد كبير من Users في نفس الوقت

Secure

بفضل JWT Signature، السيرفر يقدر يتأكد إن Data اللي جوا الـ JWT موثوقة ومش متعدل عليها

Cross-platform

تقدر تستخدم الـ JWT في تطبيقات Web، Mobile، وحتى IoT devices.

Disadvantages

Difficult to revoke

لو الـ Token تم تسريبه، مفيش طريقة سهلة تلغيه إلا لو انتهت مدة صلاحيته

Size

الـ JWT ممكن يكون حجمه أكبر مقارنة بالـ Session ID لإنه بيحتوي على بيانات أكثر

AutoMapper

تخيل إنك شغال على مشروع ضخم فيه كمية ضخمة من البيانات، وكل شوية عايز تبعت الـ Data بين الـ Frontend والـ Backend. كل مرة تبعت فيها، محتاج تعمل Manual Mapping بين الـ Entities والـ DTOs. هنا بقى بييجي دور AutoMapper

إيه هو AutoMapper ؟

AutoMapper ببساطة هو مكتبة بتعمل Mapping بين الـ Object بطريقة automated، يعني بتشيل عنك هم كتابة أكواد التحويل يدويا. بدل ما تقعد كل شوية تقول:

هات القيمة دي من الـ Entity وخليها تساوي القيمة دي في الـDTO ،قالك لا AutoMapper بيعمل كل ده لوحده

طيب إزاي بيشتغل؟

1. **Efficient Object Mapping**

بدل ما تكتب كود ضخم لكل Property يدويا عشان تعمل الـMapping ،ال AutoMapper بياخد النوعين اللي انت عايز تحول بينهم، ويعمل **Map** بشكل تلقائي لكل الـ Properties اللي أساميها متشابهة. يعني لو عندك FirstName في الـ Entity و FirstNameفي الـDTO ، هو هيربطهم ببعض أوتوماتيك.

2. **Advanced Mapping Scenarios**

طيب إيه اللي بيحصل لو الـ Properties أساميها مختلفة في الـ Source والـDestination ؟ هنا بييجي دور ميزة التحكم الكامل في عملية الـ Mapping مع AutoMapper، عندك مرونة كبيرة إنك تتحكم في إزاي الـ Data تتحول، حتى لو الـ Properties ليها أسامي مختلفة.

3. **Nested Mapping**

ده واحد من أكتر السيناريوهات اللي بتواجهنا في الـ Applications الكبيرة. تخيل إن عندك Object يحتوي على Object تاني جواه . ال AutoMapper مش بس بيقدر يعمل Mapping للـ Properties العادية، لكن كمان يقدر يتعامل مع Nested Objects تلقائيا.

خلينا نستعرض شوية حاجات مميزة ممكن تستفيد بيها من AutoMapper

- **Custom Mapping Logic**

الميزة دي بتسمح لك إنك تتحكم بشكل كامل في طريقة تحويل Data بين الـ Objects حتى لو الأسامي مختلفة. يعني لو عندك Data في الـ Entity مش متطابقة مع الأسماء اللي في الـ DTO، تقدر تكتب الكود اللي هيعمل التحويل بشكل مخصص، وده بيدي مرونة كبيرة للتعامل مع سيناريوهات معقدة

- **Projection**

AutoMapper كمان بيقدر يعمل Projection، وده معناه إنه يقدر ياخد Data مباشرة من الـ Database ويحطها في شكل مختلف تمامًا يناسب احتياجات الـ UI أو App.

طيب إزاي بيشتغل الـ Projection ؟

بدل ما تجيب كل Data من الـ **Entity** وتعمل **Mapping** بينها وبين الـ **DTO**، ممكن تستخدم **AutoMapper**عشان تجيب بس Data اللي محتاجها من الـ Database وتعملها Projection للـ DTO بشكل مباشر

- **Performance Optimized**

رغم إن AutoMapper بيختصر عليك كتابة كود كتير، إلا إنه مصمم إنه يكون سريع جدًا في التعامل مع Data.

الـ Performance في AutoMapper بيكون ممتاز لأنه بيعتمد على الـ Reflection والكاشينج.

طيب ازاي بيشتغل الكاشينج؟

كل ما تعمل عملية Mapping، AutoMapperبيستخدم الكود اللي انت كتبته مرة واحدة ويحفظه في الكاش، فبالتالي لما تعيد استخدامه، مبيحتاجش يحسب عملية التحويل من جديد، ده بيوفر كتير من الوقت ويخلي الأداء أسرع.

---

3 Tier Architecture

تخيل إنك شغال على APP كبير وعايز تقسمه لأجزاء علشان تبقى كل حاجة منظمة، وتقدر تطور فيها.
هنا بيجي دور 3-Tier Architecture،
طيب يلا نشوف هي ايه 3-Tier Architecture

ايه هي 3-Tier Architecture؟

الـ3-Tier Architecture دي Architectural Design بيعتمد على تقسيم App ل ثلاث Tiers وكل Tier بتقوم بوظيفة محددة ومختلفة. التقسيم ده بيساعد على تنظيم الكود، ويخلي كل جزء مسؤول عن حاجة معينة، فبالتالي لو حصل أي تعديل أو مشكلة في Layer معينة، بتقدر تركز عليها من غير ما تضطر تغير كل System .

الفكرة الأساسية إن كل Layer بتكون مستقلة عن الطبقات Layers، وده بيساعد على تحسين Maintainability وسهولة التعامل مع المشروع بشكل أكبر.

Presentation Tier – User Interface (UI) for interacting with users.

Business Logic Tier – Handles all application logic and rules.

Data Access Tier – Manages communication with the database.

دلوقتي هنعرف إيه دور كل Tier بالتفصيل

- **Presentation Tier**

الـ Presentation Tierدي **Tier**اللي User بيتفاعل معاها مباشرة، ودي اللي بتحتوي على الـ User Interface (UI) ده بيشمل Web, Mobile App أو أي شكل من أشكال UIاللي بيقدر User يشوفه ويستخدمه.

مثلاً، لو انت بتستخدم تطبيق E-Commerce، كل Buttons, Menus والعناصر اللي بتظهر للUser زي Products or Prices، دي كلها تعتبر جزء من الـ Presentation Layer

الهدف الرئيسي:

- عرض Data للUser بشكل جميل وسهل الاستخدام.

- التفاعل مع User وجمع الـ Inputs زي Forms اللي User بيملاها

- **Business Logic Tier**

الـ Business Logic Tier هي المكان اللي بتتم فيه Business Rules للApp. على سبيل المثال بيتم حساب

Discounts Or Taxes

ال Business Logic Tierبتكون في النص بين الـ Presentation Tier والـ Data Access Tier، وده اللي بيخليها مسؤولة عن اتخاذ القرارات بناء على Data اللي بتجي من الـ UI والـ Database

الهدف الرئيسي:

- Business Logic Implementation
الLayer دي بتطبق Business Rules اللي بتدير App زي التحقق من صلاحيات User، Discounts، أو تطبيق أي قواعد تجارية أخرى

- Interacting with the Data Access Tier
بعد م اعمل Data Processing، ببعتها أو تستلمها من الـ Data Access Tierعشان تتخزن أو تتعدل في Database

- **Data Access Tier**

الـ Data Access Tierدي Tier اللي بتتعامل مع الـ Databaseمباشرة.
الTier دي مسؤولة عن تنفيذ العمليات على Database زي الـ CRUD (Create, Read, Update, Delete) أي Process تتعلق بتخزين أو استرجاع Data من Database بتحصل هنا.

في Projects الكبيرة، بنحب إن Tierدي تكون منفصلة علشان نقدر نغير أو نحدث Database بدون ما نضطر نعدل على Tiers التانية.

الهدف الرئيسي:

- Managing the connection to the database
إرسال الاستعلامات (Queries) إلى قاعدة البيانات لاسترجاع أو تحديث البيانات.

- Handling the Data
سواء كان استرجاع Data من Tables معينة أو تعديل Data موجودة بالفعل كل ده بيتم هنا

ليه 3-Tier Architecture ده مهم؟

تقسيم App لـ 3 Tiers بيوفر مزايا كتير:

Separation of Concerns

كل Tierمسؤولة عن مهمة معينة، وده بيسهل عليك تMangment الكود وتتعامل مع التعديلات. لو حصلت مشكلة في جزء معين، بتقدر تصلحها من غير ما تأثر على باقي الSystem

Maintainability

لو محتاج تعدل في UIأو تغير Databaseبتقدر تعمل كده بسهولة من غير ما تضطر تعدل في باقي Tiers

Scalability

بما إن كل Tier بتشتغل بشكل مستقل ده بيخليك قادر توزع كل Tierعلى Server مختلف لزيادة Performance الخاص بـApp

إزاي Layers دي بتتواصل مع بعضها؟

1. ال Presentation Tierهي الـ (UI) بتتعامل مع User بشكل مباشر وبتاخد Inputs
2. الDataبتروح للـ Business Logic Tier اللي بيتعامل مع Logic الخاص بـApp (زي التحقق من User Data)
3. لو في حاجة للتعامل مع Databaseالـ Business Logic Tierبيوجه Data للـ Data Access Tierاللي بينفذ عمليات CRUDعلى Database
4. بعد ما تتنفذ Process دي النتيجة بترجع للـ Presentation Tierعلشان تعرضها للUser

Presentation Tier -

زي الـ Controller بيستقبل Inputs من User وبيبعتها للـ Business Logic Tier

Business Logic Tier -

زي الـ UserService بيتأكد من صحة Data وبيتواصل مع Data Access Tierلاسترجاع Data المطلوبة.

Data Access Tier –

زي الـ UserRepository بيتعامل مع Database سواء لاسترجاع أو تحديث Data

---

Middle Wares

أكيد وإنت شغال على أي Project سمعت عن حاجة اسمها Middlewares
طيب إيه هي الـMiddlewares ؟ وبتشتغل إزاي؟ تعال نفهمها مع بعض

خليني ف الاول كدا احكيلك حكاية علشان نسهل الموضوع

تخيل مثلا أنت دخلت كافيه تطلب قهوتك الحوة كدا وانت رايح الشغل ايه الخطوات اللي بتعملها علشان القهوه تجيلك

بتروح للكاشير تطلب.

الكاشير يبعت الأوردر للشيف.

الشيف يحضر الاوردر ويرجعه للكاشير.

الكاشير يسلمك القهوة.

اهو بالظبط كل خطوه حصلت زيها زي ال Middlewares وكل Middleware ليها دور معين ولازم كل Request يعدي عليه

نتكلم بقا بجد

ازاي ال Middlewares بتشتغل

الـ Middlewares بتشتغل بنظام Pipeline

يعني كل Request بيمر على أكتر من Middleware

وكل Middleware ممكن يكمل ويعمل Calling لل Middleware اللي بعده أو يرد على Request مباشرة ويرجع للـ Client

مثال بسيط على الكلام دا

لما ال User يطلب Data من API

1. أول Middleware يتحقق من Authentication هل المستخدم مسجل دخول
2. التاني يتأكد من Authorizationهل عنده صلاحية
3. التالت يعمل Logging تسجيل الطلب في ملف
4. في النهاية يوصل للـ Controller ويرجع ال Data المطلوبة .

مميزات وعيوب الـ Middleware

المميزات

1. High Flexibility يعني تقدر تعدل او تضيف اي Request جديدة بسهولة.
2. Reusability يعني ال Middleware واحدة ممكن تشتغل في أكتر من Project

العيوب

1. لو عندك عدد كبير من الـMiddlewares ، ممكن ده يؤثر على **أداء التطبيق** ويسبب بطء في معالجة الطلبات.
2. محتاجة تنظيم وإلا ال Requests ممكن تضيع أو تتأخر.

ايمته أستخدم الـ Middleware ؟

تقدر تستخدم Middlewares لما يكون عندك Cross-cutting concerns زي:

1. Authentication التحقق من هوية User
2. Authorization التأكد من صلاحيات الوصول.
3. Logging تسجيل تفاصيل كل Request
4. Exception Handling التعامل مع الأخطاء بشكل مركزي.

أما بقا لو الوظيفة مرتبطه ب جزء معين ف ال App الأفضل تخليها داخل الـ Controller أو Service

ف النهايه بقا

الـ Middlewares مهمه جدا ل اي APP

- لو فهمتها واستخدمتها صح هتقدر تتحكم في كل Request من أول ما يدخل الAPP لحد ما يطلع Request
- استخدامك للـ Middlewares هيخلي الكود بتاعك أسرع، أوضح، وأسهل في التعديل

Serialization

تخيل يا صديقي انك شغال على App وعندك Object بيحتوي على Data مهمة زي name, address, Email

الDataدي بتكون محفوظة في Memory تمام كده؟
لكن ممكن تحتاج تحفظ الDataدي في Fille علشان تقدر ترجع لها بعدين او تنقل الDataدي لAppتاني شغال على جهاز مختلف أو حتى على سيرفر تاني

هنا بتظهر المشكلة
البيانات اللي في الذاكرة مش بتبقى محفوظة بصيغة ممكن تستخدم أو تفهم خارج التطبيق .لازم يكون في طريقة لتحويلها لصيغة تقدر تخزنها أو ترسلها، وده هو دور الـ Serialization

ايه هي الـ Serialization ؟

الـ Serializationببساطة هي عملية تحويل Data أو Objects من شكلها الأصلي في Memory إلى صيغة قابلة للنقل أو التخزين زي JSON ,XAM ,Binary

أنواع الـ Serialization:

JSON Serialization

الـ JSONهو أشهر صيغة لتخزين ونقل البيانات عبر الشبكة. كتير من التطبيقات بتعتمد عليه لأنه بسيط وقابل للقراءة.

XML Serialization

الـ XMLبرضو تعتبر صيغة شائعة، خصوصًا في التطبيقات القديمة، وبتستخدم لترتيب البيانات بشكل هيكلي

Binary Serialization

الـ Binaryتعتبر أسرع وأكثر كفاءة من حيث المساحة في بعض الأحيان، لكن مش قابلة للقراءة زي XML Or JSON

الـ Deserialization

الـ Deserialization هي العملية العكسية لـ Serialization، يعني تحويل Data من صيغة قابلة للنقل زي JSON OR XML مرة تانية إلى Objects في Memory علشان تقدر تستخدمها في APP بتاعك.

ايمته نستخدم الـ Serialization ؟

Data Transfer

لما يكون عندك تطبيق ويب أو API وتحتاج تبعت بيانات بين User And Server

Storage

لما تحتاج تخزن Data في Database أو في Fills

Preserving State

لما تحتاج تحافظ على حالة Objects معينة علشان تستخدمها ف وقت تاني اللي هو ال Caching

Serialization Advantages

Easy Data Transfer

الـ Serializationبتساعدك تنقل Data بين Systems بسهولة

Storage Flexibility

تقدر تخزن البيانات بتاعتك في ملفات أو قواعد بيانات بتنسيقات قياسية زي JSONأوXML

Interoperability

لما تبعت Data بين نظامين مختلفين مثلاً من سيرفر ل عميل الـ Serializationبتخليك تقدر تستخدم Data بسهولة في النظامين

Flexibility
ممكن تحول Data لأكثر من صيغة زي JSON أو XML حسب احتياجاتك

Serialization Disadvantage

Increased Data Size
الـ Serializationأحيانا بتزيد من حجم Data، خصوصًا لو Complex Data

Processing Overhead
الـ Serializationوالـ Deserializationممكن يأخذوا وقت إضافي في بعض الحالات

Encoding
عند التعامل مع بعض أنواع الـ Serializationزي Binary، ممكن يكون من الصعب قراءة Data لو احتجت تتحقق منها

ف النهايه

الـ **Serialization**
هي عملية تحويل Object أو Data من شكلها المعقد في Memory إلى صيغة بسيطة وقابلة للنقل زي **JSON**أو **XML**أو **Binary**
ده بيخليك تقدر تبعت Data بين Systems أو تخزنها في Database أو Fills بسهولة.

أما الـ **Deserialization**
فهي العملية العكسية يعني تحويل Data اللي تم إرسالها أو تخزينها مرة تانية إلى شكلها الأصلي علشان تقدر تستخدمها في App

في النهاية
لو بتشتغل على Project
فهمك للـ Serialization و Deserialization هيساعدك في نقل Data بشكل امن وسهل بين APPs

---

Data Annotation VS Fluent API

تخيل معايا كدا وانت شغال على Project وعندك مجموعة Classes وفيه Relationsبين ال Classes دي
ف انت قدامك طريقتين علشان تعمل Mapping
Data Annotation , Fluent API ايه هما وايه الانسب فيهم دا الليس هنوضحه ان شاء الله
يلا بينا نبدأ

Data Annotation

هي مجموعة من الـ Attributesالجاهزة اللي بنستخدمها لتحديد Validation أو Configuration مباشرة
فوق Properties أو Classes زي [Primary Key – Required Fields]
يعني تقدر تقول انها السهل الممتنع

Advantages

- Simple and Easy to Use
  بسيطة وسريعة في المشاريع الصغيرة
- Code Clarity
  الكود واضح لأن الاعدادات بتكون قريبة من الـ Modelنفسه
- Quick Setup
  اعدادها سريع بدون الحاجة إلى كتابة أكواد اضافية

Disadvantages

Limited Customization
محدودة في التعامل مع Relations المعقدة بين Models

Scalability Issues
الكود ممكن يبقى فوضوي مع زيادة عدد الـ Entities

Separation of Concerns
بتخالف مبدأ Separation of Concerns لان الاعدادات مرتبطة بالـ Modelمباشرة

Fluent API

أسلوب لتكوين الـ Entities والـ Properties في Entity Frameworkباستخدام الكود بدلاً من .Data Annotationsيعتمد على تعريف إعدادات الـ Mappingداخل الـ DbContext باستخدام Method Chaining، مما يوفر مرونة أكبر وتنظيم أفضل للكود.
يطبق داخل الدالة OnModelCreating في الـ DbContext، وهو مفيد بشكل خاص في المشاريع الكبيرة والمعقدة التي تحتاج إلى تخصيص Relations بين Classes زي One-to-Manyو .Many-to-Many

Advantages

- Highly Customizable
  مرن جدًا في التعامل مع جميع أنواع Complex Relations زي One-to-Many و Many-to-Many
- Centralized Configuration
  كل إعدادات الـ Mapping بتكون في مكان واحد داخل الـ DbContext
- Better Separation of Concerns
  بيفصل الاعدادات عن الـ Model ودا بيساعد على ان الكود بيكون منظم

Disadvantages

- More Complex
  بيكون أكتر تعقيدا وبيحتاج وقت أكبر للكتابة مقارنة بـ Data Annotation
- Steeper Learning Curve
  محتاج خبرة أكتر في التعامل مع الـ Entity Framework Core

ايمته تستخدم كل طريقة؟

- **Data Annotation**
  إذا كان المشروع **صغير** أو متوسط، ومحتاج حاجة **بسيطة** وسريعة.

- **Fluent API**
  إذا كان المشروع **كبير** ومعقد، وعايز تتحكم في كل تفاصيل الـ **Mapping**وتفصل الاعدادات عن الكود.

خلاصة الكلام

- **Data Annotation** هو الحل الأمثل إذا كنت محتاج **سرعة** وبساطة في مشروع صغير.
- **Fluent API** هو الحل الأمثل إذا كنت محتاج **مرونة** وتحكم كامل في مشروع كبير ومعقد.

وفي النهاية
ممكن تستخدم الطريقتين مع بعض في نفس المشروع حسب احتياجاتك المختلفة

Comparison: Data Annotation VS Fluent API

	Data Annotation	Fluent API
Ease Of Use	Simple And Quick	Requires More Setup
Customization	Limited	Highly Flexible
Separations Of Concerns	Mixed With Model	Centralized In DBContext
Scalability	Suitable For Small Projects	Suitable For Large Abd Complex Projects

---

Value Type & Reference Type

تخيل إنك بتشتغل على ملف مهم

أول حاجة لو كريت الملف على اللاب بتاعك فأي تعديل تعمله هيبقى مرتبط بنفس النسخة اللي محفوظة على جهازك.
ده شبه **Value Types** كل حاجة محفوظة في مكانها وبتعدل فيها مباشرة.

لكن لو حبيت تكمل شغلك على الملف ده من جهاز تاني في البيت بدل ما تنقل الملف نفسه هتعمل Access عليه هنا أنت بتتعامل مع **Reference Types** لأنك بتستخدم رابط أو Pointer يوصلك للملف الأصلي علشان تعدل عليه من جهازك التاني.

فهمنا الدنيا الى حدا ما
يلا ندخل ف الموضوع

Value Type

الـ Value Type هي نوع Data بيتم تخزينه بشكل مباشر في Memory داخل الـStack يعني لما تعمل نسخة من Variable من النوع ده، القيمة نفسها هي اللي بتتنسخ، وكل نسخة بتكون مستقلة عن التانية
الـ Value Type بتشمل الأنواع الأساسية زي int, float, char, structs

Reference Type

الـ Reference Type هو نوع Data بيتم تخزينه في Heap لكن Reference(اللي هو العنوان بتاع المكان في الذاكرة) بيتم تخزينه في الـ Heap يعني لما تعمل Copy من Variable من النوع ده، النسخة بتكون مجرد referenceلمكان Data في memory، بمعنى إن النسختين بيشتركوا في نفس Data. وده معناه إنه لو غيرت Data في النسخة التانية، هتأثر على النسخة الأولى لأنها بتشير لنفس المكان في Memory

ايمته استخدم Value Type وايمته استخدم Reference Type

Value Types

- لما تحتاج قيم بسيطة ومحددة زي Numbers أو Flags
- لما تكون Data مش هتتغير كتير، أو مش محتاجة تشاركها بين أجزاء مختلفة من App

Reference Types

- لما Data تكون Large أو Complex زي Lists أو Objects
- لما تحتاج تشارك نفس Data بين أكتر من جزء في App

Advantages And Disadvantages

Value Type

✅ High Performance
بسبب التخزين في الـ Stack، التعامل مع الـ Value Typesبيكون سريع جدًا
لأن الـ Stack بيشتغل بنظام (Last In, First Out)، وده بيخليه خفيف على System

✅ Independent Copies
كل نسخة من الـ Value Typeبتكون منفصلة تماما. التعديلات اللي بتعملها على نسخة مش بتأثر على النسخ التاني، وده بيقلل احتمالية الأخطاء في الكود.

❌ Limited Use
مش مثالي للتعامل مع large data structures أو Complex Data، لأنه بيتعامل بشكل أساسي مع simple data types زي int or float

Reference Type

✅ **Flexibility**
مناسب جدًا للتعامل مع complex data زي objects وcollections الـ Reference Typesبيديك القدرة على تخزين Larg Data في الـ Heap بدل ما تثقل الـ Stack

✅ **Shared Data**
ممكن أكتر من variableيشير لنفس Data في الـ .Heapده مفيد لما تحتاج تعمل تحديثات على نفس Data من أماكن مختلفة في الكود.

❌ **Lower Performance**
التعامل مع Data في الـ Heapبيكون أبطأ مقارنةً بالـ .Stackكمان، الـ Garbage Collectorممكن يستهلك وقت لتنظيف البيانات غير المستخدمة.

❌ **Data Overlap**
بما إن أكتر من referenceممكن يشير لنفس البيانات، التعديلات اللي بتعملها على نسخة واحدة هتأثر على باقي النسخ. لو مش واخد بالك، ده ممكن يسبب unexpected behaviorأو أخطاء صعبة في التصحيح

الفكرة هنا إن الفرق الأساسي هو طريقة التخزين والتعامل مع البيانات:

- Value Types النسخة بتبقى مستقلة، وكل نسخة ليها مكان خاص.
- Reference Types بتتعامل مع نسخة أصلية باستخدام رابط والتعديلات بتتم على نفس النسخة الأصلية.

Redis

تخيل معايا كده... عندكApplication ، وكل مرة الUser بيطلب حاجة، الـ Application بيروح يدور عليها في Databaseاللي مليانة بيانات كتير جدًا. النتيجة؟ بطء رهيب. 😢
هنا ييجي دورRedis

ايه هو الRedis

Redis دا **In-Memory Data Store**، يعني بيخزن Data في **RAM** مش على الهارد، والنتيجة؟ **أداء أسرع** بكتير مقارنة بأي طريقة تخزين تقليدية.

هو مش بس Database عادي لا ده:

Key-Value Store

Redis بيشتغل كنظام تخزين بسيط جدًا، فكر فيه كأنه Dictionary كبيرة:

- عندك **Key** بتحدد Data
- وعندك **Value** اللي بتتخزن تحت المفتاح ده.

مثال:

Key: "UserName"

Value: "Ahmed"

ده معناه إنك لو طلبت "UserName"، هتاخد القيمة."Ahmed"

Cache

Redisبيتميز بسرعة رهيبة لأنه بيشتغل بالكامل في الـ **Memory (RAM)** بدل الـDisk.

- فايدته ؟ تخزين البيانات اللي بيتم طلبها كتير (Frequently Accessed Data) علشان التطبيق يجيبها أسرع.
- زي لما تستخدمه في **Caching** نتائج Queries أو web pages

Message Broker

Redisممكن تشتغل كـ Message Queue System.

- يعني لو عندك أكتر من Services بتتكلم مع بعضها، Redisبيساعدهم يتبادلوا الرسائل بطريقة منظمة.
- بيشتغل بنظام الـ Publish/Subscribe (Pub/Sub)، بحيث ان Service تبعت Publish والخدمات التانية تستقبل Subscribe

طيب ايه هي مميزاته

✅ سرعته عالية
لأنه بيعتمد على **In-Memory Storage**، بيكون أسرع 10 xمن أي DB تاني.

✅ **Multiple Data Structures**
بيشتغل مع .**Strings, Lists, Sets, Hashes, Sorted Sets, Geospatial**

✅ **High Availability**
مع **Replication**، ممكن تنسخ البيانات بسهولة لو حصل Fail في الـMaster Node.

✅ **Scalability**
مع **Clustering**، Redisيقدر يتعامل مع ملايين الطلبات في الثانية

وطبعا مفيش ميزات من غير عيوب

❌ **Loss of Data**
لو السيرفرRestart ، البيانات ممكن تضيع (ممكن تقلل المشكلة باستخدام .(**Persistence**

❌ **Memory Intensive**
لأنه بيعتمد على الـRAM ، ممكن يكون مكلف في التطبيقات اللي بتحتاج بيانات كتير جدًا.

❌ **Limited Querying**
مش بنفس مرونة الـSQL Queries.

طيب ايمته اسخدمه

1. Caching لما User يطلب حاجة لأول مرة، تخزنها في Redis علشان لو طلبها تاني، نوفرها على طول.
   - زي
     User Profile، Web Pages، أو نتائج Queries المعقدة.
2. Session Store بدل ما تخزن Session في Cookies أوDatabase ، خزنها فيRedis.
   - زي
     تخزين حالة الUser في مواقع E-commerce
3. Leaderboard الألعاب اللي فيها Scores كتير تعتمد عليه.
   - زي
     Fortnite أو أي لعبة أرقام ضخمة.
4. Pub/Sub لو عايز تبعت رسائل أو Notifications بين Apps
   - زي
     إرسال إشعارات فورية لكل الUsers لما يحصل تحديث.

الملخص ✍

Redisهو الحل المثالي لما تحتاج:

- **سرعة** في التعامل مع البيانات.
- **مرونة** في إدارة الـCache.
- **سهولة** في تخزين Sessions أو التعامل مع الرسائل.

Abstract Class & Interface

لما تكون شغال ف Project غالبا هتواجه سؤال أستخدم Abstract Classولا Interface؟ القرار ده مش عشوائي، وبيعتمد على فهمك للفرق بينهم واختيار الأنسب

وان شاء ف البوست دا هنوضح الفرق بينهم

Interface

- الـ Interface ببساطة هو Contract لما Class يقرر إنه ينفذ Interface (Implement) بيكون ملتزم يحقق كل الـ Methods
  والـ Properties اللي الـ Interface عرفها لكن من غير ما يكون في تفاصيل للتنفيذ
- قبل C# 8.0 كان الـ Interface يحتوي فقط على Method Signatures
  لكن دلوقتي بيدعم .Default Implementations

Abstract Class

الـ Abstract Classهو خليط بين Class عادي وInterface بيقدر يحتوي على:

- Abstract Methods اللي لازم الـ Class اللي يورّثها يحققها.
- Concrete Methods اللي ممكن تكون متعرفة وجاهزة للاستخدام.

طيب ايه الفرق بينهم

Purpose

- Interface لو عايز تحدد Contractبدون فرض طريقة تنفيذ.
- Abstract Class لو محتاج توفر Base Implementation مع إمكانية وجود قواعد أو منطق مشترك يتم توريثه.

Inheritance

- Interface الكلاس يقدر يطبق أكتر من Interfaceيعني يدعم Multiple Inheritance
- Abstract Class الكلاس يقدر يورث Abstract Classواحد بس Single Inheritance

Flexibility

- Interface مثالي لو عايز تحدد General Constraintsأو مواصفات عامة مشتركة بين الكلاسات المختلفة.
- Abstract Class أفضل لو فيه Shared Logicأو Logic مشترك محتاج توريثه.

Fields

- Interface لا يدعم Fieldsأو أي بيانات يتم تخزينها.
- Abstract Class ممكن يحتوي على Fieldsتُستخدم كبيانات مشتركة بين الكلاسات الفرعية.

Concrete Methods

- Interface يدعم Concrete Methodsفقط في الإصدارات الحديثة من #Cابتداءً من C# 8.0
- Abstract Class دائمًا يدعم Concrete Methods

Access Modifiers

- Interface كل شيء داخل Interfaceبيكون Publicفقط
- Abstract Class يدعم كل أنواع Access Modifiersزي Public, Private, Protected, Internal

Performance

- Interface أبطأ نسبيا عند استخدام Multiple Inheritance
- Abstract Class أسرع نسبيا لأنه يعتمد على Single Inheritance

متى تستخدم كل واحد؟(When to Use Each)

Abstract Class

لما يكون عندك سلوك مشترك (Shared Behavior) بين كذا كلاس مختلف بيحتاج توريث.
لو عندك Fieldsأو Properties عايز تورثها لكل الكلاسات المشتقة.
مناسب للعلاقات اللي تكون is-a Relationship
لما تحتاج تعرف Base Functionalityوتسمح للكلاسات المشتقة إنها تعدل عليها أو تضيف لها.

Interface

ما تحتاج تحدد Contractمشترك بين كائنات (Objects) غير مرتبطة وراثيًا.
مثالي للتطبيقات اللي بتعتمد على Polymorphismأو .Dependency Injection
بيشتغل كويس جدًا في المشاريع الكبيرة اللي محتاجة مرونة لتطبيق نفس الـ Interfaceعلى كائنات مختلفة.
مناسب أكتر لتحديد العلاقات السلوكية (Behavioral Relationships) بدل الهيكلية (Structural)

Advantages And Disadvantages

Abstract Class

✅ المميزات

- بيدعم Abstract Methodsو Concrete Methods، فبيوفر مرونة أكبر
- بيسمح باستخدام Fieldsلتخزين بيانات مشتركة
- بيدعم Access Modifiersزي(private, protected, public)

❌ العيوب

- بيدعم فقط Single Inheritance، وده ممكن يكون قيد في بعض السيناريوهات
- أقل مرونة مقارنة بـ Interfacesفي المشاريع الكبيرة

Interface

✅ المميزات

- بيدعم Multiple Inheritance، وده بيتيح تصميم أكتر مرونة
- بيبسط فكرة الـ Abstractionبإنه يركز على تعريف السلوكيات بدون أي تنفيذ
- بيعزز من استخدام Abstractionفي تصميم البرامج

❌ العيوب

- ما بيسمحش بوجود Fieldsأو Concrete Methods إلا في الإصدارات الحديثة من C#
- كل Members بيكونوا Publicافتراضيًا، وده بيقلل من التحكم في الـAccess Modifiers.
- الأداء أبطأ شوية في حالة الـ Multiple Inheritanceبسبب زيادة عمليات البحث

Summary

- Use Abstract Class
  لما تحتاج Shared Behaviorأو Fieldsمشتركة بين Classes
  أو لما Relation بين Classes تكون is-a Relationship
- Use Interface
  لما تحتاج Contractمشترك بدون تنفيذ، أو لما التصميم يتطلب Multiple Inheritance ومرونة في تطبيق نفس Behaviors على Objects مختلفة

ببساطة

- Abstract Class بيجمع بين Abstractionو Shared Implementation
- Interfaceبيديك مرونة في التصميم وتركيز على العقود المشتركة فقط

Delete, Drop وTruncate

لو إنت شغال مع Database، أكيد هتحتاج في مرحلة ما انك تحذف Data معينه سواء ال من ال Tables او Tablesنفسها وهنا بيظهر سؤال مهم
ايه الفرق بين DELETE, DROP وTRUNCATE ؟

الاختلاف بين التلاتة مش مجرد فرق في الOrders لا ده فرق في التأثير، الأداء، والسيناريوهات اللي ممكن تستخدمهم فيها.
في البوست ده هنتكلم بالتفصيل عن كل واحد فيهم وامتى تختار تستخدمه عشان تضمن أحسن أداء وتتجنب مشاكل ممكن تحصل بعد كدا

DELETE

ايه اللي بيحصل؟
الـ **Delete** بيحذف Data من Table بناءا على Conditions معينه ولو مكتبتش ال Conditions الDataكلها بتتحذف من الTable لكن Table نفسه بيبقى موجود في الـ **Database** يعني لو عندك جدول **Users** وحذفت Data معينة Table هيفضل موجود لكن من غير Data

Advantages

- ✅ الDataاللي اتحذفت ممكن تعملها Rollback لو كتبتها في Transaction Log
- ✅ يقدر يمسح Data معينة بناء على الConditionsاللي انت حاططها

Disadvantages

- ❌ بطيء مقارنة بالـ **Truncate** ودا ليه لانه بيمسح Row Row وبيتعامل مع الـ Logging System
- ❌ ممكن تاثر ع ال Performance بشكل عام

TRUNCATE

إيه اللي بيحصل؟
الـ **Truncate**بتحذف كل Data في Table مره واحده بدون Conditions ولكن Table نفسه بيبقى موجود!
العملية دي أسرع من الـ **Delete**لأنها مش بتسجل العمليات في الـ **Transaction Log**

Advantages

- ✅ أسرع بكتير من **Delete** ودا ليه لأنه ما بيتعاملش مع كل Row لوحده
- ✅ بيقلل الـ Logging والـ Locking على Table
- ✅ بتحذف ال Data من غير م تاثر ع ال Performance

Disadvantages

- ❌ مش ممكن تعمل Rollback في معظم Databases لو اتنفذت
- ❌ ما ينفعش تستخدمها لو Table مربوط بـ Foreign Key
- ❌ مش ممكن تحط Conditions يعني هتحذف كل الDataاللي ف الTable مره واحده

Drop

إيه اللي بيحصل؟

ال Dropمش بس بتحذف Data لا ده بيشيل الـ Table كله من الـ Database يعني لو استخدمت Dropعلى Table الجدول يختفي تماما ب الData اللي فيه

Advantages

✅ بيحرر مساحة كبيرة على Database

✅ سريع جدًا لأنه بيشيل Table بالكامل

Disadvantages

❌ مستحيل ترجع ال Table بعد م تخذفه Irreversible

❌ أي Objects مرتبطة بالجدول زي الـ Views أو الـ Indexes بتتشال معاه.

❌ لو فيه اي Relations ممكن تسبب مشاكل لو ال Table دا مربوط ب جداول تانيه

الملخص ✍

ال Delete لوعايز تمسح Data معينة وتفضل محتفظ بالTable

ال Truncate لو عايز تمسح كل Data بسرعة وتسيب Table موجود

ال Drop لو مش محتاج Table تاني نهائي وعايز تشيله من Database

أمتى تستخدم كل واحد؟

ال Delete لو محتاج تحذف Data معينة فقط وعايز القدرة على انك تعملها Rollbackتاني

الTruncate لو محتاج تفضي Table بالكامل بسرعة وما يهمكش انك تعمل Rollback

ال Drop لما Table خلاص انتهى استخدامه وعايز تشيله نهائي

.NET Framework.و .NET Core

لو أنت شغال بـ .NET. أو حتى لسه بتبدأ أكيد وقفت قدام السؤال دا

أشتغل بـ .NET Framework. ولا.NET Core ؟

تعاله ف البوست دا اقولك

.Net Framework

الNET Framework. دا الأساس القديم لتطوير تطبيقات .NET. بدأ في 2002 وكان الحل المثالي وقتها لتطوير الAPPs على نظام Windows

Advantages

✅ **Stability**
مستقر جدا ومناسب لأي Apps قديمةLegacy Systems

✅ **Rich Library**
بيوفر Libraries ضخمة جاهزة لدعم كل حاجة ممكن تحتاجها

✅ **Windows-Specific**
مثالي لو الAppبتاعك هيشتغل على Windows فقط

Disadvantages

❌ **Not Cross-Platform**
ما ينفعش تشتغل بيه على Linux أوMac ودي نقطة سلبية كبيرة في الوقت دا
اللي معظم الAPPsفيه بتحتاج تشتغل على أكتر من Platform

❌ **Lower Performance**
الـ .NET Framework. مش مصمم عشان يتعامل مع Modern Apps Needs
زي الكلاود أو الـMicroservices فالأداء مش هيكون الأفضل في الحالات دي

❌ **Outdated for Modern Needs**
الـ .NET Framework. ابتدى يواجه Challenges عشان يواكب التطور التكنولوجي
والتطوير فيه بقى أقل لصالح .NET Core. والنسخ الموحدة من .NET. اللي بتدعم الـ Cross-platform

.Net Core

.NET Core.هو الجيل الجديد من.NET ، واللي بيقدم تجربة تطوير حديثة وسريعة بتشتغل على أي نظام تشغيل.

Advantages

✅ **Cross-Platform**
الـ .NET Core. هو منصة متعددة الأنظمة(Cross-Platform) بمعنى إنه يشتغل على أكثر من نظام تشغيل زي **Windows**، **Mac**، و **Linux**.ده بيديك حرية كبيرة في تطوير التطبيقات، لأنك مش مُقيد بنظام تشغيل معين. زي لو كنت شغال على Windows ممكن توزع تطبيقك على Mac أو Linux بسهولة

✅ **High Performance**
الـ .NET Core. مُصمم علشان يكون خفيف وسريع في التعامل مع البيانات، وبالتالي هيعطيك أداء أفضل في التطبيقات التي تحتاج سرعات استجابة عالية زي تطبيقات الـ APIsأو التطبيقات الكبيرة

✅ **Cloud-Ready**
مناسب جدًا لتطبيقات الكلاود زيMicroservices ، وبيشتغل بسهولة مع Docker و.Kubernetes

✅ **Open Source**
الـ Source Code مفتوح، وده بيتيح لأي حد المساهمة في تطويره.

Disadvantages

❌ **Limited APIs (In Early Versions)**
في الإصدارات القديمه من الـ.NET Core ، كان فيه بعض القيود من حيث عدد الـ APIsالمتاحة مقارنة بـ .NET Framework.كان بينقصه بعض المميزات الأساسية للتعامل مع بعض أنواع Data أو الوظائف التي كانت مدعومة في.NET Framework.

❌ **Learning Curve**
لو شغال على Framework لفترة طويلة، هتحتاج وقت عشان تتعود على.Core

What's the Main Difference

1 Supported Environment

- **.NET Framework** مخصص لـ **Windows**فقط
- **.NET Core** بيشتغل على .**Windows, Mac, Linux**

2 Performance

- **.NET Core** أسرع بشكل **ملحوظ**، خصوصًا في الـ .**Web APIs**

3 Support & Development

- **.NET Framework** بيحصل على **تحديثات maintenance** فقط.
- **.NET Core** يتم تطويره بشكل **مستمر** مع ميزات جديدة.

4 Target Applications

- **.NET Framework** مناسب لـ التطبيقات القديمة و .Windows Forms
- **.NET Core** مثالي للتطبيقات الحديثة، الكلاود، و التطبيقات القابلة للتحديث.

When to Use Each

- **.NET Framework**
  لو بتشتغل على نظام قديم أو تطبيقات مخصصة لـ Windowsفقط.
- **.NET Core**
  لو بتبدأ مشروع جديد أو محتاج تطبيق حديث يدعم الـ Cloudو يشتغل على أي نظام تشغيل

متى تستخدمه؟

- لو عندك مشاريع Legacy شغالة بـ.NET Framework.
- لو بتطور تطبيق Windows Desktop فقط.
- لو شغلك كله داخل بيئة.Windows Server

) .NET Core 2️⃣البطل العصري 🚀 (

.NET Core هو الجيل الجديد اللي أطلقته Microsoft سنة 2016، وبيعتبر خطوة نحو المرونة والأداء العالي:

- **Cross-Platform**: 🌍 بيشتغل على Windows, Mac, و Linux.
- **أداء خرافي (High Performance)**: 🔥 أسرع بكتير في التعامل مع الطلبات الكبيرة.(Requests)
- **Modular Design**: 🎯 ممكن تختار وتضيف بس اللي تحتاجه بدل تحميل Framework كامل.
- **Cloud-Ready**: ☁️ مثالي لتطبيقات Cloud والـ Microservices.
- **مفتوح المصدر (Open Source)**: 🔓 المجتمع كله بيشارك في تطويره وتحسينه.

متى تستخدمه؟

- لو بتبدأ مشروع جديد وعايز تضمن إنه يشتغل على أكتر من نظام تشغيل.
- لو بتعمل تطبيقات Cloud Native أو Microservices.
- لو محتاج أداء أعلى مع مرونة في التطوير.

Trigger

مساء الخير عليكم جميعا
اكيد طالما اشتغلت Database سمعت عن الTrigger

في الأول كده... تخيل معايا
عندك Database كبيرة جدا كل مرة بتعملInsert ، Updateأو حتى Delete على Table معين بيبقى فيه شوية Operations إضافية لازم تحصل بشكل تلقائي.
النتيجة؟ شغل Manual كتير وممكن يحصل أخطاء

هنا ييجي دور الـ **Trigger**
ايه هو وبيعمل ايه دا اللي هنفهمه ان شاء الله

ايه هو الـTrigger ؟

ببساطة الـ Trigger ده زي "رد فعل أوتوماتيكي" بيحصل جوه الـ Database لما يحصل حدث معين على Table أوView
هو مش Code عادي، ده Code بيتنفذ تلقائيًا من الـ Database نفسها لما يحصل Event معين زيInsert ، Update، أوDelete

أنواع الـTriggers

Before Trigger

- ده بيشتغل قبل ما يحصل الـ Event الأساسي.
- مثال: لو عايز تتأكد إن قيمة معينة Valid قبل ما تضيفها في الـTable.

After Trigger

- ده بيشتغل بعد ما يحصل الـEvent.
- مثال: لو عايز تسجل Log أو تعمل Backup بعد ما يتم إضافة Data جديدة

Instead Of Trigger

- النوع ده مش بيكمل الـEvent لكنه بيبدله بعملية تانية
- مثال: لو عايز تمنع Delete مباشر على Table معين وتستبدله بعملية Archiving للData بدل ما تمسحها.

استخدامات الـTrigger

1 Data Validation
تضمن إن الData الصحيحة بس هي اللي تدخل الـDatabase زي منع ادخال مرتب أقل من الحد الأدنى

2 Auditing
تسجل أي تعديل على Data زي (مين عدل؟ وايمته؟ وايه اللي اتغير؟) في Table مخصص

3 Auto Update
تحديث الData المرتبطة تلقائيا زي تقليل المخزون لما يتم تسجيل Request جديد

4 Enforcing Business Rules
تفرض قواعد العمل زي منع حذف عميل عنده طلبات مفتوحة

5 Maintaining Historical Data
تخزين نسخة قديمة من الData قبل التعديل للاحتفاظ بالسجل التاريخي

6 Data Cleaning
تصحيح البيانات تلقائيا زي تحويل البريد الإلكتروني لحروف صغيرة

7 Security Enforcement
منع التعديلات غير المصرح بها على الData الحساسة

8 Notifications
ارسال إشعارات تلقائيا عند حدوث أحداث مهمة زي سحب مبلغ كبير من الحساب

مميزات الـTrigger

✅ **Automation**
العمليات بتحصل تلقائيا بدون تدخل.

✅ **Consistency**
بتساعد على الحفاظ على قواعد الـDatabase

✅ **Auditing**
تسجل كل حاجة بتحصل وده بيساعد في التحليل لاحقا

عيوب الـTrigger

❌ **Difficulty in Debugging**
لو الـ Trigger معقد بيكون صعب تلاقي المشاكل فيه.

❌ **Impact on Performance**
الـ Triggers ممكن تبطئ الأداء لو مش معمول حسابها كويس.

❌ **Hidden Logic**
بعض الناس بتشوف إن وجود Logic جوه الـ Database بيخلي الكود صعب المتابعة

فين تستخدمه؟

Audit Trail
زي لما تحتاج تسجل كل تعديل على بيانات العملاء
Auto Calculations
تحديث القيم تلقائيًا زي حساب الـ Total بعد إضافةOrder
Data Cleaning
تمنع القيم الغلط قبل ما تدخل الـDatabase
Enforcing Security
تمنع التعديلات غير المصرح بها على بيانات حساسة

Finally

الـ Triggers بتساعدك تضمن جودة البيانات، الأمان، وتحسين العمليات تلقائيًا في الـ Database

---

Delegates

أكيد لو كنت شغال مع#C لازم تكون سمعت عن الـ Delegatesبس خليني في البداية أشرحلك بإيجاز
ايه هو الـ Delegate ده وأزاي ممكن يفيدك في شغلك.

في الأول كده..
تخيل معايا وانت شغال على مشروع في #C وبيزيد معاك حجم الكود كل يوم وبتواجه مشكلة انك بتكرر نفس الأكواد في أماكن مختلفة
عشان تنفذ نفس الـOperations في الوقت ده بتحتاج حاجة تكون flexible وقوية عشان تسهل عليك العملية

النتيجة!!
التكرار ده ممكن يسببلك مشاكل في الـMaintaining وتنسيق الكود هنا بقى ييجي دور الـ Delegate
اللي هيساعدك في اختصار الوقت والحفاظ على الكود من الفوضى

الـ Delegate ده زي Sender للـ Methodsاللي بتستخدمها في كودك بحيث تقدر تمررها كـ parameter لطرق تانية أو حتى
تستخدمها لتنفيذ عمليات معينة بدون ما تكرر الكود بنفسك

ايه هو الـDelegate ؟

ببساطة كدا الـ Delegate دا objectبيمثل methodمعينة
بيسمحلك بتمرير الـ methodكـ parameterلطرق تانية أو حتى تحديد طريقة التنفيذ بشكل ديناميكي في أكتر من مكان كأنك بتخلي
الكود يبقى مرن وقابل لإعادة الاستخدام.

هو مش مجرد كود عادي لا دا وسيلة للتفاعل مع الـ methodsمن خلال objectsمعينة بتمثلها وبالتالي تقدر تنفذ الأكواد بشكل أكثر
تنظيما ومرونة

أنواع الـDelegates

Single Delegate
ده Delegate بيمثل methodواحدة بس. يعني لو انت عايز methodواحدة تتنفذ في وقت معين، تستخدم الـ Delegate ده.
Example
لو عندك methodهتنفذ عملية معينة في وقت معين، زي حساب تكلفة منتج أو تنفيذ عملية حسابية.

Multicast Delegate
ده Delegate بيقدر ينفذ أكتر من methodفي نفس الوقت.
Example
لو عندك مجموعة من الـ methodsعايزهم ينفذوا مع بعض، زي تنبيه وإرسال رسالة بعد إضافة طلب جديد.

Generic Delegate
الـ Generic Delegateبيخليك تستخدمه مع أنواع بيانات مختلفة. يعني بدل ما تكتب أكواد متشابهة للـ typesالمختلفة، تقدر تستخدم الـ generic delegateلتبسيط العملية.

Action, Func, Predicate
في#C ، فيه delegatesجاهزة زي Action, Funcو Predicate، دول بيوفروا عليك كتابة delegateمخصص.

- Delegate: **Action** بيمثل **method**ما بترجعش قيمة.(**void**)
- Delegate: **Func**بيمثل **method**بترجع قيمة.
- Delegate: **Predicate**بيتم استخدامه لتصفية البيانات.

استخدامات الـDelegate

Event Handling
الـ Delegatesبتعتبر من أهم الأساسيات في التعامل مع الـ eventsفي .#C لما يحدث eventمعين، الـ delegate بيحدد طريقة التعامل مع هذا الحدث.

Callback Functions
الـ delegateبتستخدم بشكل كبير لما عايز تمرر methodكـ callbackعشان تنفذها في وقت لاحق.

Passing Methods as Parameters
لو عندك methodعايز تمررها لغيرها عشان تتنفذ في وقت لاحق، الـ delegateبيخليك تعمل ده بكل سهولة.

Event-driven Programming
الـ delegateبيساعد في تنظيم التعامل مع eventsبشكل فعال، خصوصًا في التطبيقات المعتمدة على الـ UIأو الويب.

Asynchronous Programming
ممكن تستخدم الـ delegateمع الـ asyncو awaitعشان تدير العمليات غير المتزامنة بشكل مرن، وتفصل عملية التنفيذ عن الـ UI

مميزات الـDelegate

Flexibility
الـ delegateبيوفر لك مرونة كبيرة في التعامل مع الـ methodsبكل سهولة.

Code Reusability
ممكن تعيد استخدام نفس الـ methodفي أكتر من مكان بدون تكرار الكود.

Event Handling
الـ delegateبيخليك تدير الـ eventsبكفاءة، وتنفذ الأكواد المطلوبة في الوقت المناسب.

Decoupling
بتقدر تفصل بين الـ logicوالـ **events**، وبالتالي الكود بيكون أكثر نظافة وسهولة في الصيانة.

عيوب الـDelegate

Complexity in Debugging
لو الكود اللي بيستخدم الـ delegateمعقد، هتلاقي صعوبة في تتبع الأخطاء والـ flowبتاع الـ program.

Performance Issues
استخدام الـ delegatesبشكل مفرط ممكن يأثر على الأداء، خصوصًا لو كنت بتنادي heavy operationsأو عمليات معقدة.

Hidden Logic
الـ delegateبيخلي الـ logicمش واضح في بعض الحالات، لأنه ممكن بيتم تنفيذه في وقت لاحق أو من مكان تاني في الكود.

فين تستخدمه؟

Event Handling
لو عندك eventsمعينة في التطبيق زي ضغطة زر أو تغيير في بيانات الـ UI، تقدر تستخدم الـ delegateللتعامل مع eventsدي.

Asynchronous Operations
في حالة التعامل مع عمليات غير متزامنة، الـ delegateبيساعدك تنفذ الأكواد بطريقة مرنة ومتوازية.

Passing Methods
لو بتحتاج تمرر methodكـ parameterلغيرها، الـ delegateبيسهل عليك العملية دي بشكل كبير.

Multicast
لو عندك مجموعة من الـ methodsعايزهم ينفذوا مع بعض، استخدم multicast delegateعشان تديرهم بسهولة.

في النهاية

الـ delegatesبتوفر لك طريقة مرنة وسهلة لتنفيذ الأكواد في C#, وتساعدك على اختصار التكرار وتنظيم الكود بشكل ممتاز

Constructor

لو عندك Class وعايز تبدأ تشتغل عليه
إزاي بتضمن إن كل Object جديد بيتم إنشاؤه من الـ Class ده هيبدأ بالطريقة الصح؟
يعني هل الData اللي فيه هتكون مضبوطة؟
هل Settings مظبوطة؟ معتقدش
طيب ايه الحل ؟
الحل هنا هو ال Constructor ايه هو وبيعمل ايه دا اللي هنوضحه ف البوست دا

ايه هو الـConstructor ؟

ببساطة كدا الـ Constructor هو Special Method بتتكتب جوا الـ Class وهدفها إنها تجهز أي Object جديد أول ما يتم إنشاؤه
يعني هو اللي بيحدد إزاي الـ Object ده هيشتغل

هل هيكون فيه Data افتراضية؟
هل هيحتاج إعدادات معينة؟
هل هتحتاج تمنع إنشاء الـ Object أصلاً؟

يعني تقدر تقول إنه بوابة الدخول لأي Object عندك ف الCode
الـ Constructor بيتنفذ تلقائي وقت إنشاء Object جديد من غير ما تحتاج تستدعيه بنفسك

أنواع الـConstructor

عندي خمس انواع لل Constructor
[1] Default Constructor

ده أبسط أنواع الـConstructor وبيتم إنشاؤه تلقائيًا لو ما كتبتش أي Constructor في الـClass

- **فكرته :** بيجهز Object بدون ما يحتاج منك أي بيانات.
- **الاستخدام :** لو الـ Object مش محتاج أي إعدادات أولية أو قيم معينة

[2] Parameterized Constructor

هنا بنضيف Parameters للـ Constructor عشان نمرر بيانات أثناء إنشاء الـObject.

- فكرته : إعداد الـ Object بناءً على البيانات اللي بتديهاله
- الاستخدام : لو محتاج تخصيص الـ Object عند إنشائه

[3] Copy Constructor

ده Constructor بياخد Object من نفس النوع كمعطى (Parameter) وينسخ بياناته.

- فكرته : عمل نسخة مطابقة من Object موجود بالفعل.
- الاستخدام : لما تحتاج تكرر نفس البيانات في Object جديد مع الحفاظ على الأصل

[4] Static Constructor

ده نوع خاص جدًا من الـConstructor ، بيتنفذ **مرة واحدة بس** لكلClass ، وبيشتغل مع البيانات أو الخصائص اللي بتكون.Static

- **فكرته :**تهيئة القيم أو الإعدادات المشتركة لكل Objects من الـ.Class
- **الاستخدام :**لما يكون عندك بيانات أو Configurations عامة لكل الـObjects

[5] Private Constructor

ده نوع مميز لأنه بيمنعك من إنشاء Object من الـ Class مباشرة.

- **فكرته :** التحكم الكامل في طريقة إنشاء الـ.Objects
- **الاستخدام :** لما يكون الـ Class Utility (للوظائف العامة فقط) أو لما تطبق Design Patterns زيSingleton

[6] Chained Constructor
ده مش نوع منفصل لكنه طريقة لتنظيم الكود لما يكون عندك أكتر من Constructor في نفس الـClass

- **فكرته :** استدعاء Constructor معين من داخل Constructor تاني لتقليل تكرار الكود
- **الاستخدام :** تحسين قراءة الكود وتقليل الكتابة المكررة

[7] Default Values Constructor

ده بيجمع بين Default Constructor و Parameterized Constructor، حيث يتم إعداد بعض القيم افتراضيًا مع السماح بتمرير معطيات إضافية.

- **فكرته :** المرونة بين استخدام بيانات افتراضية أو تخصيص القيم.
- **الاستخدام :** توفير الإعدادات الافتراضية بدون إجبار المستخدم على تمرير كل القيم

طيب ليه الـ Constructor مهم؟

- **بيهيئ الData :** بيساعدك تجهز الـ Object بالبيانات المطلوبة وقت إنشائه.
- **بينظم الCode :** بيسهل عليك كتابة الكود بطريقة واضحة ومنظمة.
- **بيحسن الPerformance :** يقلل الحاجة لاستدعاء إعدادات إضافية بعد إنشاء الـObject

Advantages

Code Initialization بيساعدك تجهز الـ Object بسهولة
Clean Code بيقلل الحاجة لكتابة Methods إضافية
Encapsulation بيحافظ على البيانات جوا الـClass

طيب ايمته ممكن يعملك مشاكل

لو Constructor معقد جدا ممكن يسبب مشاكل في الأداء
استخدام Constructors كتير بنفس الـ Signature ممكن يعمل Confusion
لو فيه Error جواه هيوقف إنشاء الـ Object بالكامل

ايمته استخدم كل Type من ال Constructors

- لما يكون الـ Class بسيط، استخدم.Default Constructor
- لو محتاج إعدادات أولية، روح لـ.Parameterized Constructor
- لو بتشتغل في Patterns زي Singleton الـ Private Constructor هو صديقك
- ال Static Constructor بيساعد جدا في الـ Configurations المشتركة

Asynchronous & Synchronous

الفرق بين Sync و Async
ف الاول كدا خلينا نحكي حكايه صغيره نسهل فيها الموضوع
فاكر لما كنت بتحجز عند الميكانيكي عشان تصلح عربيتك وتقوله هاقعد جنبك لحد ما تخلصها ده بقا اسمه Sync يعني انت موقف حياتك لحد م يخلص العربيه
إنما لما تقوله خلصها براحتك وانا هروح اخلص اللي ورايا وارجعلك تاني ده Async اعتقد كدا المعنى وصل تعاله بقا نشوف الموضوع دا بالنسبه للكود اللي انت بتكتبه

Synchronous

في Synchronous Programming، كل Task لازم تخلص بالكامل قبل ما العملية اللي بعدها تبدأ
بمعنى إن الكود بيتنفذ Line by Line وده بيخلي أي Delay في العملية الحالية يوقف البرنامج كله

🖊️ إزاي بيشتغل؟

كل Operation بتحجز Thread كامل في الـ Processor لحد ما تخلص وبالتالي Main Thread بيبقى مشغول ومش هيقدر ينفذ أي حاجة تانية

مثال بسيط : زي لما تستنى دورك في طابور، مش هتتحرك غير لما الشخص اللي قدامك يخلص.

Advantages

1. Code Simplicity

الكود بيكون سهل القراءة والفهم Readable and Predictable
مفيش تعقيد من ناحية ترتيب العمليات No Context Switching

2. Easier Debugging

تتبع الأخطاء (Errors) بيكون بسيط لأن العمليات بتتنفذ بترتيب واضح.

3. Low Overhead

مفيش إدارة إضافية Extra Management من الـ Runtime

Disadvantages

1. Blocking Behavior

لو العملية الحالية بطيئة زي File I/O أو HTTP Request هتوقف كل حاجة وده هيأثر على Performance

2. Limited Scalability

غير مناسب للتطبيقات اللي فيها طلبات كثيرة أو عمليات طويلة.(Long-Running Tasks)

Asynchronous

في Asynchronous Programming العمليات مش لازم تخلص بالكامل عشان يبدأ اللي بعدها الكود بيشتغل بأسلوب الـ Non-Blocking بمعنى إن العملية بتبدأ و Thread الرئيسي بيروح يشتغل على عمليات تانية
ولما العملية تخلص بيتم إبلاغ الـ App بتاعي Callback or Promise

🖊️ إزاي بيشتغل؟

الـ Async بيستخدم **Event Loop** أو **Task Scheduler** لإدارة العمليات.

لما تبدأ Task معينة زي طلب HTTP أو قراءة File الكود ما بيستناش بيكمل تنفيذ بقية ال Operations
ووقت انتهاء الـ Task بيتم تنفيذ الـ Callback المربوط بيها

Advantages

1. Non-Blocking Execution

الكود مش بيوقف الOperations التانية ودا بيحسن الأداء العامBetter Throughput

2. Scalability

مناسب للAPPs اللي بتحتاج التعامل مع عدد كبير من الـ Requestsفي نفس الوقت زي الـWeb Servers

3. Efficient Resource Utilization

بدل ما الـ Threadيبقى مستني عملية تخلص بيشتغل على عمليات تانية

Disadvantages

1. Increased Complexity

الكود الـ Asyncبيبقى أصعب في الفهمHard to Read والتنفيذHard to Write

2. Overhead

تنفيذ الOperations بشكل Async بيحتاج إدارة إضافيةExtra CPU Cycles عشان يتابع الحالة بتاعت كل عملية

3. Debugging Challenges

عمل Debugging بيكون أصعب مع استخدام Callbacksأو Awaiting Tasks

طيب ايمته تستخدم Sync وAsync ؟

تستخدم Sync لما

1. اما تكون Fast Operations ومفيهاش انتظار كبير
2. الكود بسيط ومش محتاج تنفيذ عمليات كتيرة Concurrent Tasks
3. لو الApp بيخدم عدد قليل من الUsers ومفيش ضغط كبير على الPerformance

تستخدم Async لما

1. عندك operations بطيئة أو طويلة زيHTTP Requests, File Uploading, Database Queries
2. لو بتشتغل على تطبيق Web أو Mobile وعايزعدد كبير من Users
3. لما تكون محتاج Optimize Performanceعن طريق استغلال الResources بشكل أفضل

هل الـ Asyncدايمًا أسرع؟

مش شرط السرعة بتعتمد على نوع Operations

I/O Bound Tasks

زي: قراءة الFilles، HTTP Requests، أو التعامل مع DB

بيكون أسرع هنا لأنه مش بيوقف التنفيذ أثناء انتظار الOperation

CPU Bound Tasks

زي: Images processing، حسابات معقدة زي ML Models، أوEncryption

مش هيبقى أسرع هنا لأن الProcessor شغال بأقصى طاقته

الأفضل استخدام Parallel Processingلتوزيع المهام على أكتر من Core

ليه أحيانًا Async بيبقى بطيء؟

Context Switching Overhead

الContext Switching بياخد وقت وResources إضافية

Poor Implementation

لو استخدمت **Await**بشكل خاطئ أو متسلسل بدل ما تستفيد من .**Concurrency**

Concurrent Executionالعمليات تشتغل في نفس الوقت.

Sequential Executionالعمليات تشتغل واحدة ورا التانية

الخلاصة

- **Async** أسرع في عمليات.I/O
- **Parallel Processing** أفضل معCPU Bound

Clustered Index & Non-Clustered Index

لما تيجي تتكلم عن Performance Optimization في SQL Server لازم أول حاجة تفكر فيها هي Indexes
ليه؟ ببساطة لأنهم بيحددوا إزاي الـ Database Engine بيبحث عن Data

يعني لوعندك كتاب مفيهوش فهرس كل مرة تدور على حاجه انت محتاجها من الكتاب لازم تقرأ كل الصفحات واحدة واحدة!
لكن لو فيه فهرس منظم هتوصل للي بتدور عليه بسرعه.
وده بالضبط اللي بيعمله Indexingفي Database

طيب السؤال المهم:
إيه الفرق بين Clustered Index وNon-Clustered Index ؟
وإزاي نستخدم كل واحد فيهم لتحسين أداء Queries

ف الاول إيه هو الـ Index أصلاً؟

الـ Indexفي SQL هو Data Structure بيحسن سرعة SELECT queriesعن طريق إنشاء طرق سريعة للوصول للData من غير م احتاج اعمل Full Table Scan الـ Index بيشتغل زي الفهرس في كتاب:

Clustered Index

بيرتب الصفحات فعليا حسب العمود يعني Data نفسها بتتخزن بنفس ترتيب الـIndex

Non-Clustered Index

بيعمل Lookup Tableمنفصلة فيها الValues ومعاها Pointersبتوصلك للصفوف الأصلية في Table

Clustered Index

الـ Clustered Index بيحدد ترتيب تخزين الصفوف نفسها على القرص .بمعنى، البيانات في الجدول مترتبة فعليًا بناءً على الـ Clustered Index

طيب ازاي بيشتغل

[1] لما تعمل Clustered Indexعلى Column معين الـ SQL Server بيرتب البيانات في الTable بناء على Values في Column ده

[2] كل Table يقدر يكون عنده Clustered Indexواحد بس لأنه بيتحكم في ترتيب الData نفسها

Example

لو عندك جدول فيه مجموعةCustomers والـ Primary Keyهو CustomerID لما تعمل Clustered IndexعلىCustomerID الData نفسها هتكون مرتبة بناء عليه

```
CREATE CLUSTERED INDEX IX_CustomerID
ON Customers (CustomerID);
```

Advantages

- ✅ أفضل Performance عند البحث عن Data باستخدامPrimary Key
- ✅ مثالي للQueries اللي بتطلب Sorting for Data أو تعمل Range Queries زي BETWEEN وORDER BY
- ✅ بيقلل الحاجة إلىFull Table Scan وبالتالي يحسن الـ Performance بشكل كبير

Disadvantages

- ❌ أي Update في Values اللي عليه الـ Index ممكن يسبب إعادة ترتيب للData وده يأثر على Performance
- ❌ بما إنه بيتحكم في ترتيب الData الTable يقدر يكون عنده Clustered Index واحد بس.

Non-Clustered Index

الـ Non-Clustered Indexهو Logical Indexبيتم تخزينه في Separate Data Structure عن الData الفعلية
بمعنى إن الData مش Physically Orderedبناء على Non-Clustered Index
لكنه بيحتفظ بPointers تشير ل Rows الحقيقية داخل الTable وده بيساعد في تسريع Query Optimization بدون تغيير ترتيب الData الأساسي

طيب ازاي بيشتغل

الData مش مترتبة فعليا بناء على Non-Clustered Index لكنه بيحتوي على References Pointersبتوصلك للRows الأصلية بسرعة

الTable يقدر يكون عنده أكتر منNon-Clustered Index لأن كل واحد بيشتغل على Column مختلف.

Example

لو عندك Products Table وبتبحث دايما بProductName بدل الـID ممكن تضيف Non-Clustered Index على ProductNameعشان البحث يكون أسرع

```
CREATE NONCLUSTERED INDEX IX_ProductName
ON Products (ProductName);
```

Advantages

✅ **يسمح بوجود أكتر من Index على نفس الTable ودا بيحسن أداء الQueries**
✅ **بيحسن سرعة البحث في Columns غير المرتبة**
✅ **بيقلل التحمل على الـClustered Index**

Disadvantages

❌ **أبطأ من Clustered Index وقت استرجاع Data الكاملة، لأنه يحتاج إلى Lookup للRows الحقيقية**
❌ **بيستهلك Storage إضافية في التخزين**

طيب ايمته استخدم كل حاجه منهم

Clustered Index

- لو بتبحث عن Rangeمن الData أو عايز تحسن Sorting
- لو الTable بيتم قراءته بكثرة Read-Heavy Table
- لو بتتعامل مع Primary Keyبشكل متكرر

Non-Clustered Index

- لو بتبحث عن Value محددة Single Value Lookup
- لو بتبحث باستخدام Columns مختلفة كل مرة
- لو Table بيحتوي على Complex Queries
- لو بتحتاج أكثر من Index على نفس Table

Entity FrameWork

لما تيجي تشتغل على .NET Applications.وتحتاج تتعامل مع Database، عندك حل من الاتنين:

تكتب Raw SQL Queries بنفسك وده ممكن يكون متعب ومعقد.

تستخدم Entity Framework (EF) واللي بيعتبر ORM (Object-Relational Mapper)يسهل عليك التعامل مع Database بدل ما تكتب SQLمباشرة.

السؤال هنا :هل Entity Framework هو الحل الأمثل لكل حاجة؟ 🤔
ولا في حالات مينفعش تستخدمها فيها؟
دا اللي هنتكلم فيه ان شاء الله

ايه هي ال Entity Framework

الEntity Framework (EF) هو ORM Frameworkبيسهل التعامل مع Relational Databasesعن طريق تحويل Tablesفي Database إلى C# Objects، بحيث تقدر تنفذ CRUD Operations (Create, Read, Update, Delete)
من غير م تحتاج تكتب SQL Queriesيدويا

- **الفكرة الأساسية:**

بدل ما تكتب:

```
SELECT * FROM Products WHERE Id = 1;
```

ممكن تعمل كده:

```
var product = context.Products.Find(1);
```

وده بيخلي الكود cleanوأسهل في Maintenance and development

طيب ايمته نستخدم ال Entity Framework

- When the project requires faster development

ال EF بيساعدك تكتب كود أقل وتتعامل مع Databaseبطريقة OOP وده بيقلل وقت الDevelopment

- When the application is Read-Mostly

لو الApp بيقرأ Data أكتر من إنه يعمل Edit عليها EFبيكون اختيار ممتاز مع **AsNoTracking()**لتحسين الPerformance

- When the application requires continuous maintenance and updates

بما إن EF بيشتغل بطريقة Abstraction Layer تقدر تغير أو تطور Database Structureبدون تأثير كبير على الCode

- When working on a .NET Core or .NET 6+ project

ال EF Core متكامل مع .NET Core.وبيوفر أداء عالي وميزات متطورة زي Cross-Platform Supportو.NoSQL Support

إمتى منستخدمشEntity Framework ؟

- When you need maximum performance

لو عندك App بيعمل ملايين الـ Database Transactionsفي الثانية EFممكن يكون أبطأ من Dapperأو .Raw SQL

- When queries are highly complex and require advanced optimization

في بعض الحالات EF LINQ Queriesممكن تنتج Inefficient SQL Queries
وفي الحالة دي ممكن يكون أفضل تكتب Stored Procedureبنفسك.

- When you need full control over the database

لو بتحتاج تعمل Custom Indexing, Partitionsأو Tuning يكون Advanced ممكن EF يكون محدود بالنسبة لك.

Advantages

✅ Reduces the amount of code you write
تقدر تعمل CRUD Operations بدون م تحتاج تكتب SQL Queriesيدويا

✅ Supports Migrations easily
تقدر تعدل Database Schemaبدون الحاجة لكتابة ALTER TABLEيدويًا.

✅ Supports Change Tracking
بيعرف التعديلات اللي حصلت على Entitiesوبيعمل Updateتلقائيا.

✅ Supports Lazy Loading & Eager Loading
بيساعدك تتحكم في طريقة تحميل Related Dataلتحسين الأداء.

✅ Cross-Platform Support in EF Core
يشتغل على Windows, macOS, Linux مع .NET Core.

Disadvantages

❌ Slower performance compared to raw SQL or Dapper
لأن EF بيضيف Abstraction Layer، ممكن يكون أبطأ في بعض السيناريوهات.

❌ Some queries may not be optimized
لو مكتبتش LINQ Queries بشكل صحيح، ممكن EF يولد SQL Query ضعيفة الأداء.

❌ Higher memory consumption
بسبب Change Tracking & Caching ممكن يستهلك Memory أكتر من اللازم في التطبيقات الكبيرة.

أنواع Entity Framework

Entity Framework 6 (EF6)

- الإصدار القديم من EF
- بيشتغل على .NET Framework فقط.
- مناسب لل Projects القديمة اللي شغالة على .NET Framework.

Entity Framework Core (EF Core)

- الإصدار الحديث من EF
- بيشتغل على .NET 6+ & .NET Core.
- أداء أعلى ودعم للـ Cross-Platform
- بيدعم NoSQL & In-Memory Databases

String & StringBuilder

اما بتيجي تتعامل مع Strings بيكون عندك اختيارين أساسيين:

– string وهو Immutable ودا بيكون غير قابل للتغير بمعني ان أي تعديل عليه بينشئ Object جديد في الMemory
– StringBuilder وهو Mutable ودا بيكون قابل للتغير بمعني انه بيسمح لك بالتعديل على نفس الObject من غير م تحتاج تنشئ Object جديد في كل مرة

لكن السؤال المهم إمتى نستخدم كل واحد فيهم؟ 🤔
الإجابة بتعتمد على Performance, Memory Consumption, and Use Case,
لأن كل واحد فيهم ليه Pros & Cons والاختيار الصحيح بيأثر بشكل كبير على Application Efficiency.

String

هو Reference Typeلكنه بيتصرف زي Value Type وده لأنه Immutable object بمعنى إن أي تعديل عليه بينشئ Object جديدة في الMemory بدل ما يغير النص القديم التعديل ده بيتم في Heap Memory وعلشان كده كل مرة بتعدل فيها على نص بيتم إنشاء Object جديد وحذف القديم وده ممكن يسبب استهلاك زائد للMemory

✅ ايمته استخدمString

✓ لما يكون عندك Fixed Texts زي Messages, Labels, Configurations لأنك مش هتعدل عليهم كتير

✓ لو Readability & Simplicity أهم منPerformance لأن stringبيخلي الكود واضح وسهل فـmaintenance

✓ في String Comparison لأن الـ string valuesبيتم تخزينها في Intern Poolداخل .NET Runtime. وده بيقلل Memory Allocationفي بعض السيناريوهات

❌ **طيب وايمته مستخدموش**

✗ لما تحتاجFrequent Modifications زي Concatenation inside Loops لأن كل تعديل بيعمل New Allocation in Memoryوده ممكن يؤدي ل High Memory Consumption

✗ لو Performanceمهم لأن كل مرة بتعدل على string بيتم إنشاء New Object ودا بيخلي Garbage Collection Overhead تزيد

✗ لو بتتعامل مع Large Text Data وبتعدل عليه كتير الأفضل تستخدم StringBuilderلانه بيعدل على نفس Memory بدل ما يعمل New Instancesكل مرة

StringBuilder

هو Reference Typeوبيتصرف كـ Mutable Object بمعنى إن أي تعديل عليه بيتم على نفس الـ Object في الMemory بدل ما ينشئ Objectجديدة زي string التعديلات بتتم داخل Heap Memoryلكن بدون Multiple Allocations وده بيحسن Performance وبيقلل Memory Consumption خصوصا ف الScenarios اللي فيها تعديلات متكررة عStrings

✅ **ايمته استخدم StringBuilder**

✓ لما تحتاج تعديلات متكررة ع الStrings زي Concatenation داخلLoops لأن StringBuilder بيعدل على نفس الـ Memory Allocationبدل ما ينشئ New Objectsكل مرة

✓ لما تشتغل علىLarge Text Data زي تكوين Text Files، Reports

✓ لو عاوز تحسن Performance لأن StringBuilder بيقلل من Garbage Collection Overhead عن طريق In-Place Modification

❌ **طيب وايمته مستخدموش**

✗لو عندك Fixed Text زي Labels, Messages, Configurations، ودا لان stringبيكون أوضح وأسهل في القراءة

✗لو بتتعامل معString Comparison لأن StringBuilderمش بيتم تخزينه في Intern Poolزي string وده ممكن يأثر على Memory Efficiency

✗لما تكون بتعدل عدد قليل من المرات لأن StringBuilderفيه Extra Overheadوممكن يكون أبطأ في السيناريوهات البسيطة

First, FirstOrDefault, Single, SingleOrDefault, Where, Find, ToList

طبعا وانت شغال على بروجيكت ومحتاج تعمل Retrieveل Data من ال Database فأكيد هتقابل أكتر من طريقة تجيب بيها الData دي بس السؤال هنا:

أستخدم FirstOrDefault()ولا Single()؟ طب إمتى أستخدم Find()؟ وإيه الفرق بينهم؟

First

بص دا بيرجع أول Record يحقق الConditionاللي انت باعته لكن لو مفيش Data هيكسر الدنيا وهيرجعلي Exception

طيب تستخدمه ايمته؟

- لو متأكد 100% ان فيه على الأقل Record واحد مطابق للCondition
- لو عاوز أول Result بس حتى لو فيه أكتر من Result

❌ **ما تستخدموش ايمته؟**

- لو مش متأكد ان فيه Data لأنه هيعمل Exceptionلو مفيش ولاRecord

```
var user = _context.Users.First(u => u.Age > 25);
```

FirstOrDefault

نفس فكرة First() بس الفرق انه لو مفيش Data بيرجع null بدل ما يرجعلي Exception

طيب تستخدمه ايمته؟

- لو الData بتاعتك ممكن ترجع ب Null
- لما تبقى عاوز تتجنب Exceptions وتحافظ على الكود شغال بدون مشاكل

❌ **ما تستخدموش ايمته؟**

- لو محتاج تتأكد إن فيه Data لأن nullممكن يسبب مشاكل لو مش مهندله كويس

```
var user = _context.Users.FirstOrDefault(u => u.Age > 25);
```

Single

بيرجع Record واحد بس لكن لو لقى أكتر من Record أو مفيش أصلا هيعملException

طيب تستخدمه ايمته؟

- لما تكون متأكد ان فيه Record واحد بس يحقق الCondition
- في الحالات اللي الـ Business Logic بيقول إن مفيش غير Value واحدة مطابقة للCondition

❌ **ما تستخدموش ايمته؟**

- لو ممكن يكون فيه **أكتر من Record** لأنه هيعمل مشاكل ف الكود
- لو الData ممكن تكون Null لأنه هيعمل Exception لو مفيش أي Record

```
var user = _context.Users.Single(u => u.Email == "test@example.com");
```

SingleOrDefault

نفس فكرة ()Single لكن الفرق انه بدل ما يعمل Exception لو مفيش Data راجعه بيرجع null

طيب تستخدمه ايمته؟

- لما تكون متأكد ان هيرجع Record واحد بس طبقا للCondition او ميرجعش خالص
- لما تحتاج تتجنب Exception لو مفيش Data لكن لو أكتر من واحد برضه هيعملException

❌ **ما تستخدموش ايمته ؟**

- لو ممكن يكون فيه أكتر من Record ليه نفس الValue لأنه برضه هيعملException

```
var user = _context.Users.SingleOrDefault(u => u.Email == "test@example.com");
```

Find

بيستخدم الـ Primary Key عشان يجيب الData ولو الحاجة اللي بتدور عليها موجودة في الـCache مش هيروح للـ Database أصلا

طيب تستخدمه ايمته؟

- لو الSearch هيكون بس باستخدام الـPrimary Key
- لما تحتاج performance سريع لأنه لو الـ Record موجود في الـ Cache مش هيعمل Query على الـDatabase

❌ **ما تستخدموش ايمته؟**

- لو بتبحث بCondition غير الـPrimary Key لأن كده مش هيشتغل

```
var user = _context.Users.Find(1);
```

Where

بترجع كل الـ Records اللي بتحقق الCondition اللي انا بعته

طيب تستخدمه ايمته؟

- لما تحتاج تجيب Listفيها أكتر من Record بناء على Condition معين
- لما تعمل فلترة للData قبل ما تحولها لـList

❌ ما تستخدموش **ايمته؟**

- لو عاوز Recordواحد بس استخدم ()FirstOrDefault

```
var users = _context.Users.Where(u => u.Age > 25).ToList();
```

ToList

بيحول الResult لـList في الMemory وده معناه انه بينفذ ال Quarry فورا

طيب تستخدمه ايمته؟

- لما تكون عاوز ترجع الData كـList مباشرتا
- لما تحتاج تستخدم الResults أكتر من مرة من غير ما تعمل Query كل شوية

❌ **ما تستخدموش ايمته؟**

- لو الكود مش محتاج List كاملة عشان ما تحملش الDataكلها في الMemory بدون داعي

```
var users = _context.Users.Where(u => u.Age > 25).ToList();
```

ملخص سريع كدا

📌 **لو عاوز ترجع أول Result بس**

✓استخدم First() لو متأكد إن فيه Data

✓استخدم FirstOrDefault() لو مش متأكد وعاوز تتجنب الـException

📌 **لو عاوز نتيجة واحدة بس ولازم تتأكد إنها وحيدة:**

✓استخدم Single() لو متأكد إن فيه Recordواحد بس

✓استخدم SingleOrDefault() لو ممكن مايكونش موجود بس لازم يكون يا اما واحد يا اما مفيش

📌 **لو بتبحث باستخدام الـPrimary Key**

✓استخدم Find()لأنه أسرع في البحث بالـ Primary Key

📌 **لو عاوز List من الData**

✓استخدم Where()مع ToList() لو بتحتاج مجموعة من الResults

✓ استخدم ToList() الاول قبل ما تخلص الفلترة عشان ما تحملش كل الداتا في الMemory

---

DTOs

لما تيجي تبني اي App غالبا هتحتاج تبعت Data بين الـ Layers المختلفة أو حتى بين Microservices المشكلة هنا انك لو استخدمت الـ Entitiesبتاعة الـ Database بشكل مباشر هتقع في مشاكل

زي Overposting، Performance Issues، Security Risks، و Tight Couplingبين الـ Layers.

قالك هنا بقا بييجي دور الـ DTOs

ايه هو وبيعمل ايه دا االلي هنشرحه ان شاء الله ف البوست دا

ايه هو الـDTO ؟

الـ DTO دا اختصار ل Data Transfer Objects وهو Plain Object بيحتوي على الDataفقط

بدون أي Business Logicأو Behaviors

الهدف الأساسي منه هو نقل الDataبين الـ Layers المختلفة في الAppبشكل More Security and Performance

Example

تخيل عندك كلاس **UserEntity**

```
public class UserEntity

}

public int Id { get; set; }

public string Name { get; set; }

public string Email { get; set; }

public string Password { get; set; }
```

```
public string Role { get; set; }
```

```
{
```

في الـ API Response مش منطقي إنك تبعت الـ Password أو حتى الـ Role لكل Users فبدل ما تبعت الـ Entity كاملة تقدر تستخدم DTO بالشكل ده

```
public class UserDto
```

```
}
```

```
public int Id { get; set; }
```

```
public string Name { get; set; }
```

```
public string Email { get; set; }
```

```
{
```

بكدا لما تعمل GET Request علشان ترجع Users Data هيرجع الـData المطلوبه بس بدون أي Data مهمه او حساسه

ليه نستخدم الـDTOs ؟

Security

الـ DTOs بتمنع تعريض البيانات الحساسة زي الـ Passwords، Tokens، أو Internal Fields

Performance Optimization

بدل ما تبعت كل الـData اللي ف الـEntity بتبعت المطلوب بس وده بيقلل حجم Data المبعوته ودا هيحسن الـ Response Time

Encapsulation & Abstraction

بيخلق فصل بين الـ Domain Models والـAPI Response وده بيسهل Development وكمان Maintainability

API Versioning

تقدر تدعم أكثر من اصدار للـ API بدون الحاجة لتعديل الـDatabase Schema

Prevents Over posting Attacks

لو بتبعت Object مباشرة للـ API الـUser ممكن يضيف Values غير مسموح بيها لكن مع DTOs تقدر تحدد الـData المسموح للـUser

طيب هو فيه اوقات ممكن مستخدمش فيها Dtos ؟؟

قالك اه

طيب ايمته؟

❌ لو الـApp صغير جدا ومفيش Layers كثيرة بين الـ Database والـAPI

❌ لو الـData اللي بتبعتها هي نفسها اللي في الـEntity فمفيش داعي لإنشاء DTO منفصل لانك هتكرر نفس الكود بدون اي داعي

❌ لو الـPerformance هو الأولوية لأن تحويل الـ Entities لـ DTOs ممكن يستهلك CPU Cycles اضافية

شوية حاجات خلي بالك منها وانت بتستخدم DTOs

Keep DTOs Simple

الـ DTO مفروض يحتوي على الـData فقط بدون أي Logic أو حسابات

Use AutoMapper

لو الـApp كبير وفيه Multiple DTOs استخدم AutoMapper عشان تقلل الـ Boilerplate Code

Avoid Overuse

مش كل API يحتاج DTO لو الـApp بسيط ممكن تستخدم الـ Entities مباشرة

Flatten Complex Structures
بدال ما ترجع Nested Objects خليك دايما مع Flat DTOsلتحسين الPerformance
Separate DTOs from Entities
ممنوع تخلط بين الـ DTOsو الـ Domain Models عشان متعملش Tight Couplingبين الـLayers

طيب ايه الطرق اللي احول بيها بين ال Entities وDTOs
عندي 4 طرق

Manual Mapping
الطريقة دي تعتمد على كتابة كود تحويل يدوي لكل Entityو DTOودا بيديك تحكم كامل في الOperations

✅ المميزات

- الPerformanceعالي بدون أي Overheadاضافي
- تحكم كامل في التحويل وامكانية إضافة Custom Logicبسهولة

❌ العيوب

- ممكن يؤدي إلى Boilerplate Codeبسبب التعامل مع عدد كبير من DTOs
- صعوبة في الMaintenance مع زيادة حجم الApp
- هتلقى نفسك بتكرر ف نفس الكود كتير

AutoMapper
لو مش عايز تكتب الكود بايدك لكلEntity يبقى تستخدم AutoMapper وهي مكتبة بتعمل الMapping لوحدهاAutomatic

✅ المميزات

✔ الCode بيكون قليل وأسرع في الDevelopment
✔ بيدعم تحويل معقد بين الEntities

❌ العيوب

❌ ممكن يسبب Performance Overhead لأنه بيعملReflection و Dynamic Mapping
❌ لما يحصل Error الDebugging مش بيكون سهل

LINQ Projection
بدل ما تعمل Mappingبعد ما تجيب الData من DB ممكن تستخدم LINQ Projection وتحول الـ Entityل DTO مباشرة جوه الـQuery وده بيكون أسرع

✅ المميزات

✔ ال Performance عالي لأنه بيحمل الData المطلوبة بس
✔ بيقلل Memory Usage لأنه مابيحملش Entitiesكاملة

❌ العيوب

❌ مش دايما يكون مرن لو عندك تحويل معقد
❌ لو عندك Business Logicداخل الـDTO مش هينفع تستخدمه بسهولة

Reflection

دي طريقة تستخدم Reflectionعشان تحول أي Entity لـ DTO بشكل عام بدون ما تكتب Mapping لكل واحد

✅ **المميزات**

✔ بتكون Flexible جدا وتقدر تحول أي Entity لأي DTO بسهولة
✔ مش محتاج تكتب Mapping لكل Classيدوي

❌ **العيوب**

❌ أبطأ طريقة لانها بتستخدم Runtime Type Inspection
❌ ال Debugging صعب وممكن يحصل Unexpected Errors

طيب أي طريقة هي الأفضل

✅ **لو عايز أعلىPerformance** ← استخدم **Manual Mapping**
✅ **لو عايز توفر وقت وتكتب كود أقل** ← استخدم **AutoMapper**
✅ **لو عايز سرعة فيQueries** ← استخدم **LINQ Projection**
✅ **لو عايز حاجةDynamic** ← استخدم **Reflection**بس مش دايما أنصح بيها

CQRS

وانت شغال على اي APP وبدا يكبر معاك والCode يكبر فجاه بدات تواجه Problems ف الApp زي

-ال Performance بقا بطيء ودا بسبب ان الQueries بقت More Complex ف دا خلاها تضرب ف الDatabase
-وكمان مشكلة Scalability لما الترافيك يعلى والSystem يبدا يهنج
-الComplexity بسبب ان عمليات الRead وWrite متداخله مع بعض وبتأثر ع الPerformance

طيب وبعدين مفيش حل ولا ايه 🙄
لا طبعا اكيد فيه الCQRS واد فكرته ببساطه انه بيفصلك بين عمليات القراءه والكتابه بدل م يكون عندي Model واحد بيعمل كل حاجه

يعني ايه CQRS

الCQRS اختصار ل Command Query Responsibility Segregationودي Architectural Pattern
بتقولك:

❌ ماينفعش القراءة Queries تشتغل بنفس طريقة الكتابة Commands
✅ خلي كل واحدة ليها Model منفصل علشان تحقق Performance & Scalabilityأعلى

يعني بدل ما يكون عندك Single Modelبيتعامل مع Read & Write Operations لا الCQRS بيقسم الـ Application لجزئين:

1️⃣ Write Side
اللي هو Command Model ودا مسؤول عن اضافة، تعديل، وحذف الData

Read Side2
اللي هو Query Model ودا مسؤول عن انه يعمل Retrieve للData فقط

يعني بدل ما نفس الـ Entity تستخدم في كل العمليات CQRS بيفصلهم علشان يحسن الPerformance
يقلل الـBottlenecks ويسهل الـ Scaling

طيب ايه هي الBenefit من CQRS

1 Performance Boost
الـ Read & Write Models منفصلين فده بيسمح لك تستخدم Data Structures مختلفة لكل واحدة

- ممكن تستخدم SQL للـWrites ، و NoSQLللـ Reads علشان تقرأ أسرع
- تقدر تعمل Caching للـ Read Data بدون ما تأثر على عمليات الWrite

2 Read Scaling
لو عندك High Read Traffic زي E-commerce Systems أو Social Media Platforms
تقدر تعمل Database Replication للـ Read Side بس وتحسن الPerformance

3 Security & Data Consistency

- ممكن تمنع الـ Read Models من تعديل الData وبالتالي تحافظ على الـData Integrity
- مع Event Sourcing تقدر تحتفظ بكل الUpdates اللي حصلت على الData

4 Flexibility رهيبة في الـData Models

- الـ Write Side بيحتفظ بالـ Business Logic والـValidations
- الـ Read Side ممكن يكون بسيط وسريع ويستخدم Views أو Projections

5 CQRS + Event Sourcing
لو جمعت CQRS مع Event Sourcing هتقدر تحتفظ بكل الUpdates اللي حصلت على الData
وده بيكون مهم جدا في الApps المالية الـAuditing Systems والـBlockchain

ايمته تستخدم CQRS

✅ لو عندك High Read Traffic والApp بيقرأ Data أكتر بكتير من الWrite
✅ لو محتاج Scalability عالي ومحتاج Database Replication علشان تقسم الحمل
✅ لو عندك Complex Business Rules بتحتاج Validation وتقسيم واضح للـLogic
✅ لو شغال على Microservices Architecture وعايز تفصل الـRead & Write Services
✅ لو بتستخدم Event Sourcing وعايز تحافظ على سجل كامل لكل الOperations اللي حصلت

ايمته CQRS يكون Overkill

❌ لو الApp بسيط وكل الـ CRUD Operations عادية
❌ لو عدد الـ Reads والـ Writes متقارب جدا ومش محتاج فصل واضح
❌ لو الـ Development Team صغيرة لأن CQRS هيزود الـ Complexity وممكن يعمل مشاكل في الـMaintenance
❌ لو مش محتاج Scaling عالي أو Read Optimization

قبل م تستخدم الCQRS اسال نفسك السؤالين دول

✓هل عندي High Read/Write Operations؟
✓هل الApp محتاج Scalability عالي؟

✓هل Queries معقدة جدا؟ وبتحتاجPerformance Optimization ؟
✓هل محتاج Event Sourcing علشان أوثق كل العمليات؟

❌لو إجابتك لأ على معظم الأسئلة دي فـ CQRS غالبا هيكونOverkill وهيزود الـ Complexity بدون فايدة حقيقية

LINQ

عارف لما تفضل تكتب for loopعشان تلف على Listكبيرة وتعمل filtering, sorting, groupingوتيجي بعد فترة ترجع للكود تحس إنك تايه وسط الـLoops ؟

أو جربت مثلا قبل كدا انك تقعد تعمل Debug لكود فيه بـ Loops و If Conditions وتحس انك مش فاهم حاجه او الدنيا رخمه

طيب ليه تعمل ف نفسك كدا وانت ممكن تستخدم Linq

ايه هوLINQ ؟

بص ي سيدي الLinq دا اختصار ل Language Integrated Query ودا مكتبة
بتوفر طريقة موحدة وسهلة لكتابة Queriesعلى Collections, Databases, XML, JSON وغيرها بنفس الBehavior
من غير م احتاج استخدم Loopsو Conditions كتير

ليه تستخدمLINQ ؟

Code Readability

بدل ما تكتب كود طويل ومعقد باستخدام For Loopsو If Statementsعشان تفلتر أو ترتب Data لا LINQبيختصرلك الموضوع بطريقة واضحة وسهلة القراءة

Less Boilerplate Code

سطر واحد يكفي بدل Loops و If Conditionsكتير
وكلمة Boilerplate Code معناها الCode المكرر اللي بتكتبه في كل مرة لما تتعامل مع Data

Unified Querying

بيكون نفس الـ Syntax علىLists, Arrays, Databases, XML, JSON
يعني بدل ما تتعلم Methods مختلفة علشان تتعامل مع Collectionsأو Databasesأو XML
لا LINQبيوفر أسلوب موحدUnified Querying تقدر تستخدمه مع كل دول!

Performance Optimized

أحيانًا LINQ أسرع منTraditional Loops

يعني ف بعض السيناريوهات LINQبيكون أكفأ وأسرع من الـ For Loops
لأنه بيستخدم Deferred Executionو Query Optimization

- **Deferred Execution**
  يعني الQuery ما بيتنفذش فورا لكنه بيتأجل لحد ما تحتاج الResult وده بيحسن الPerformance
- **Query Optimization**
  لما تستخدم LINQ مع Entity Frameworkأو Databases الCode بيتحول SQL Queryمحسن بدل ما يتنفذ في الMemory

Linq Syntax

– **Query Syntax** ودي شبهSQL
– **Method Syntax** باستخدامLambda Expressions

LINQ Methods

عندي مجموعة Methods زي

Filtering

بستخدمها علشان اصفي Data بناءا على Condition معين

Sorting

الترتيب باستخدام OrderBy()و OrderByDescending()

OrderBy()

بترتب البيانات تصاعدي

OrderByDescending()

بترتب البيانات تنازلي

Grouping

بجمع الData باستخدام GroupBy()

وبتستخدمه لتقسيم الData ل Groups بناء على Value معينة

Selecting

تحويل الData باستخدام Select()

بتستخدم لتحويل الData لاستخراج Values معينة أو تعديل هيكل الData

Aggregating

حساب الValues باستخدام Sum(), Average(), Count(), Max(), Min()

بتستخدم الـ Aggregation Methods لحساب الValues الكلية داخل الـCollection

Joining

دمج الData باستخدام Join()

وبتستخدم لربط الData بين جدولين مختلفتين بناء على Key مشترك

ايمته استخدم Linq

- محتاج كود نظيف وسهل القراءةCode Readability
- بتتعامل مع Collections معقدة وتحتاج Filtering, Sorting, Aggregationبسرعة
- محتاج Code Maintainability بدون تكرار زائدBoilerplate Code
- شغال ب Entity Framework معDatabase Queries

ايمته مستخدمش Linq

- اما يكون الPerformance عندك اهمHigh Performance Scenarios
  بعض LINQ Queriesأبطأ من For Loopsالعادية
- ال Queries اللي بتكتبها بتكون More Complex في بعض الحالات SQL Queriesبتكون أسرع وأوضح
- بتشتغل على كود بسيط جدا لو مجرد For Loopكافية مش لازم تستخدمLINQ

Code First & Database First

لما تيجي تشتغل على Project جديد هتقابلك طريقتين علشان تعمل Create لـDatabase

– Code First Approach
لما تبدأ بالكود الأول و EF Coreيتولى إنشاء الـ Database Schemaتلقائي
– Database First Approach
لما تبدأ بانشاء الـ Database Schemaالأول وبعدها تولد الـ Modelsتلقائي من الـ DB

وهنا بييجي السؤال المهم تستخدم Code First ولاDatabase First ؟ وإيه الأفضل منهم ؟

ايه هو Code First

ال Code First هوالأسلوب اللي بيخليك تبدأ بتصميم الكود أولا وبعدين Entity Framework
بيتكفل ببناء Database Schemaتلقائي بناء على الـ Classes اللي بتكتبها
الفكرة الأساسية هنا انك مش بتتعامل مع SQL Tablesمباشرة لا بتعتمد على OOP لانشاء الEntities والRelations بينها
وبعدها EF بياخد الCode ده ويحوله لهيكل الDatabase

ليه ممكن تختار Code First ؟

Full Control Over Code
مفيش حاجة اسمها Manually Creating Tables كل حاجة بتكتبها بالكود والـ EF Coreبيتولى التنفيذ

Best for New Projects
لما تبدأ Project من الصفر Code First بيكون حل مرن وسهل للتعديل والتطوير بدون ما تحتاج تتعامل مع SQL بشكل مباشر

Migrations Support
أي تعديل على الكود تقدر تطبقه بسهولة على الـ Databaseبدون ما تحتاج تحذف أو تعيد بناء الـ Tablesيدوي

Faster Development
بدل ما تضيع وقت في تصميم Database تبدأ بالكود اللي انت فاهمه والـ EF Coreبينيها لك بشكل Automated

طيب ايمته تكون Code First غير مناسبة

If You Have an Existing Database
لو بتشتغل على Database جاهزة تعديل الـ Schemaمن خلال Code Firstممكن يكون رخم شوية وغير عملي
If the Project Requires Frequent Schema Changes
في بعض الأحيان الManual Controle في الTables بيكون أفضل لو الـ Database Structureبيتغير كتير
If a Dedicated DBA Team Manages the Database
في بعض الشركات فيه فريق متخصص في تصميم وإدارة الـ Database وساعتها Database Firstبيكون الأنسب

Advantages

– **Full Control Over the Code**
أي تعديل على الـ Modelsهيتم تحديثه تلقائيا في الـ Database
– **Migrations Support**
تقدر تضيف وتعدل أي حاجة بسهولة بدون التعامل المباشر مع SQL Scripts
– **Best for New Projects**
لو بتبدأ Frome Scratch يبقى Code Firstهو الحل الأمثل

Disadvantages

- لو الـ Database Schemaمعقد جدا ممكن إدارة الـ Migrationsتبقى صعبة
-لازم تكون فاهم كويس EF Core Migrationsعشان تتجنب Breaking Changes

ايه هوDatabase First

الDatabase First هو النهج اللي بيخليك تبدأ بتصميم Database أولا وبعدها Entity Frameworkبيحولها ل Classesاللي بتمثل الـ Entitiesبناء على Database Schema اللي تم إنشاؤه

بمعنى تاني الـ Database هي المصدر الرئيسي وبعدها بتستخدم EF Coreعشان ينشئ لك Modelsتلقائي من خلال Scaffoldingبدل ما تكتبها يدوي

ليه ممكن تختارDatabase First ؟

Perfect for Existing Databases

لو عندك Databaseجاهزة بالفعل فالـ Database Firstهو الحل المثالي عشان تستورد الـ Schemaبدل ما تعيد بناؤها من الصفر

Full Database Control

هتقدر تتحكم يدوي في Schema, Indexing, Stored Proceduresوكل التفاصيل الخاصة بالـ Database من خلال الـ SQL Scriptsمباشرة

Works Well for Large & Complex Databases

لو شغال على Enterprise-Level Databaseفيها Relations معقدة Stored Proceduresأو،Views، فالتحكم المباشر هيكون أسهل بكتير من الاعتماد على EF Core Migrations

Ideal When DBAs Manage the Database

زي م قولنا فوق ان ف بعض الشركات فيه Database Administratorsمسؤولين عن ادارة وانشاء Database وساعتها Database Firstبيكون الخيار الطبيعي

متى تكون Database First غير مناسبة؟

If You Need High Flexibility in Development

لما تستخدم Database First التعديلات في الكود بتتطلب تحديثات يدوية في الـ Database Schema وده ممكن يكون غير عملي للProjects اللي بتحتاج تغييرات مستمرة

If You Want to Avoid Manual Database Management

بما انك بتبدأ من الـ Database فأي تعديل لازم يتم يدوي وده ممكن يكون صعب في الProjects اللي عاوزة Agile Developmentوسرعة في التعديلات

If You Prefer Working with Object-Oriented Models First

لو انت متعود تكتب الكود الأول باستخدام Classesوبعدين تبني الـ Database تلقائي فالـ Database Firstممكن يكون غير مريح بالنسبالك

Advantages

Best for Existing Databases –

لو عندك DB شغالة بالفعل مش هتحتاج تعيد بناء كل حاجة من الأول

Manual Schema Control –

هتقدر تتحكم في الـ Tables, Keys, Indexes, Constraintsبدل ما تعتمد على EF Migrations

Better for Enterprise Applications –
مناسب للProjects الكبيرة اللي فيها Databaseمعقدة وبيتم إدارتها من قبل DBAs
Supports Stored Procedures & Views –
لو شغلك بيعتمد على Stored Proceduresأو Views، فالـ Database Firstبيكون الحل المثالي.

Disadvantages

Slower Development
أي تعديل في الـ Modelsمحتاج تعديله يدوي في الـ Database وده ممكن يعطل الدنيا

مش مرن زي Code First
التعديلات لازم تتم من خلال SQL Scriptsبدل ما تكون Automatically Generatedزي Code First

محتاج خبرة في SQL
لازم يكون عندك معرفة قوية بـ SQL Schema Designعشان تManage التعديلات

ايمته تستخدم كل واحدة؟

✅ **استخدم Code First لما:**
✓بتبدأ Project جديد ومفيش Databaseلسه
✓عايز تحافظ على Version Controlللـ Database باستخدام Migrations
✓التيم بتاعك مرتاح أكتر في العمل بالـ Modelsبدل الـ SQL Scripts

✅ **استخدم Database First لما:**
✓عندك Databaseجاهزة فيها جداول وRelations مش عايز تغيّر فيها كتير
✓شغال مع Legacy Systemsأو Projects موجودة بالفعل.
✓ال DBAsبيتحكموا في الـ Database Schemaومش عايزين حد يعدل عليها من الكود

Unit Testing

تخيل انك شغال على Project ضخم وكل شوية فيه Updates جديدة مرة تضيف Feature ومرة تصلح Bug
وفجأة تلقى حاجة كانت شغالة تمام وقعت بسبب الUpdate دا طيب ايه السبب انت مش عارف
هنا بقى بييجي دور الـ Unit Testing اللي بيضمنلك ان الكود بتاعك شغال زي الفل حتى بعد أي تغيير

ايه هو Unit Testing

ال Unit Testingهو Automated Testبيتم على مستوى أصغر وحدة في الكودUnit وغالبا بتكون Methodأو Class
والهدف منه هو التأكد ان الUnit دي بتشتغل بشكل صحيح في جميع الScenarios الممكنة

ليه ممكن تختار Unit Testing ؟

Bug Detection Early
بتكتشف الـ Bugs بدري جدا قبل ما تدخل في مراحل متقدمة من الDevelopment وبالتالي بتوفر Time وEffort كبير

Code Confidence
لما يكون عندك Tests لكل جزء من الكود تقدر تعدل براحتك وأنت مطمن ان أي تغيير مش هيسبب مشاكل ف حاجة تانية

Better Code Design
لما تبدأ تكتبUnit Tests هتلاقي نفسك بتقسّم الكود بطريقة Modularأكتر وده بيحسن من Code Quality

Faster Debugging
بدل ما تفضل تدور على المشكلة في الProject كله الاختبارات هتحددلك بالظبط فين الError

طيب ايمته تكون Unit Testing غير مناسبة؟

❌ **لو الProject صغير جدا**

لو عندك Project بسيط جدا ومش هيكون فيه اي Updates اوDevelopments كتير ممكن يكون الـ Unit Testing Overkill ومش ضروري

❌ **لو بتتعامل معUI-Heavy Applications**

اختبار الـ UI Components باستخدام Unit Tests مش دايما سهل وساعات بتحتاج Integration Tests أو End-to-End Testing عشان تتأكد من كل حاجة

❌ **لو الProject بيحصل عليه Updates باستمرار**

لو الProject بيتغير كل شوية فكتابة Unit Tests ممكن تكون مضيعة للوقت لأنك هتفضل تعدل فيها بشكل مستمر

إيه الTools اللي ممكن تستخدمها في Unit Testing معNET. ؟

– **xUnit** الأكثر استخدامًا في .NET Core. خفيف، سريع، وبيوفرfeatures قوية زي Theory وFixtures

– **NUnit** فريم وورك مرن جدا، مناسب للمشاريع الكبيرة، وبيقدم Assert قوية وتخصيص عالي للاختبارات

– **MSTest** الحل الرسمي من مايكروسوفت، متكامل مع Visual Studio، لكنه أقل شيوعًا مقارنة بـ xUnit وNUnit

عشان تكتب Unit Tests بشكل احترافي هتحتاج Test Framework يساعدك في تنفيذ الاختبارات وإدارة النتائج في .NET.
عندك 3 فريم ووركس أساسيين كل واحد ليه مميزاته وبيستخدم في سيناريوهات معينة

الxUnit الأكثر استخداما مع .NET Core.

يعتبر الأكثر شهرة في بيئة .NET Core.لأنه خفيف وسريع جدا

بيعتمد على مفهوم Dependency Injection بشكل طبيعي، وده بيخليه مرن أكتر عند كتابة الاختبارات

بيقدم Features قوية زي Theory Attribute اللي بتخليك تمرر بيانات مختلفة لنفس الاختبار بسهولة بدل ما تكتب اختبارات مكررة

بيستخدم Fixtures لادارة Shared Context بين الاختبارات بدون الحاجة لاستخدام SetUp وTearDown زي باقي Frameworks

NUnit الأكثر مرونة وقوة في المشاريع الكبيرة

واحد من أقدم وأقوى الفريم ووركس فيNET. وبيتم استخدامه في الProjects الكبيرة اللي بتحتاج تحكم أكبر في الاختبارات

بيقدم Assertions قوية ومتنوعة بتساعدك في كتابة اختبارات دقيقة جدا

بيدعم Parameterized Tests بشكل ممتاز، وده بيخليك تمرر بيانات مختلفة لنفس الاختبار بسهولة

بيدعم Parallel Test Execution وده بيخلي الاختبارات تشتغل بشكل أسرع على الأنظمة الكبيرة

MSTest الحل الرسمي من مايكروسوفت

مدعوم رسميا من مايكروسوفت وبيجي بشكل مدمج مع Visual Studio
مما يخليه اختيار مناسب للProjects اللي بتعتمد على Microsoft Stack بالكامل

مناسب لو شغال في بيئة بتستخدم Azure DevOps أو Microsoft Ecosystem بشكل عام

أقل مرونة من xUnit وNUnit لكنه بيقوم بالغرض لو مش عايز تضيف Dependencies خارجية

بيدعم بعض الـ Attributes زي TestInitialize و TestCleanup لادارة Test Setup وTest Teardown

MSTestالحل الرسمي من مايكروسوفت

مدعوم رسميا من مايكروسوفت وبيجي بشكل مدمج مع Visual Studio
مما يخليه اختيار مناسب للمشاريع اللي بتعتمد على Microsoft Stackبالكامل

مناسب لو شغال في بيئة بتستخدم Azure DevOpsأوMicrosoft Ecosystem بشكل عام

أقل مرونة من xUnit وNUnit لكنه بيقوم بالغرض لو مش عايز تضيف Dependenciesخارجية

بيدعم بعض الـ Attributesالجيدة زي TestInitializeو TestCleanup لادارة Test Setupو Test Teardown

ايمته يكون Unit Testing هو الاختيار المثالي؟

لما تكون شغال على Project كبير وعايز تتأكد ان كل جزء فيه شغال بشكل مستقل
لما يكون عندك Business Logicمعقد ومحتاج تعمل عليه Testing كويس
لما يكون Development Team كبير وعايز تتأكد إن أي تعديل جديد مش هيسبب مشاكل ف الكود القديم
لما تكون عايز Code Maintainabilityأعلى بحيث تقدر ترجع بعد فترة وتكون متأكد ان الكود شغال تمام.

إزاي تكتب Unit Test يكون فعال؟

🟢 **خلي كل Test يكون Independent**

يعني كل Test يشتغل لوحده ومايعتمدش على أي Test تاني عشان لو واحدFail مايبهدلش الدنيا كلها كده هتعرف بسهولة إذا كان العيب في الكود ولا في الـ Test نفسه

🟢 **اعمل test كل Scenarios الممكنة**

ماينفعش تكتب Test للحالات العادية بس!
لازم تجرب Edge Cases زي الInputs الغلط، او Null Values، وExceptions علشان تتأكد إن الكود بيتعامل صح مع كل حاجة

🟢 **استخدم Mocks و Fakes بدل الحاجات الحقيقية**

لو الكود بيتعامل مع Database أو API خارجي ماينفعش في كل Test تروح فعلا تتصل بيهم
هتاخد وقت وهتعمل مشاكل استخدم Moqأو أي Mocking Framework تاني علشان تحاكي الـ Dependencies وتخلي الـ Test سريع وStabe

---

Class & Struct

طبعا كلنا نعرف ال Class وStruct
بس هل كلنا نعرف الفرق بينهم
هل نعرف ايمته نستخدم كل حاجه فيهم ايمته
هل ممكن الفرق بينهم بياثر ع الPerformance معتقدش
ان شاء الله هنشرح كل دا ف البوست انهارده

ايه هو Class

دا Reference Type يستخدم لإنشاء Objects وبيتم تخزينه في Heap Memory ودا معناه أنه بيSend الReferences بدلا من الValues

خصائص الـClass

- بيتم تخزينه في Heap Memory وده بيخليه مناسب للObjects الكبيرة والمعقدة
- دا بيعمل Send By Reference يعني أي Update عليه ينعكس في جميع الأماكن اللي بتستخدم نفس الـReference
- بي Support الInheritance وكمانPolymorphism
- يعتمد على Garbage Collector لادارة الMemory لكنه ممكن يسبب بعض التأخير
- يستخدم للComplex Objects التي تحتاج لUpdates باستمرار

ايه هو Struct

هو Value Type يستخدم لإنشاءObjects يتم تخزينها في Stack Memory وده معناه إنه بينقل القيمBy Value بدل من تمرير References وده بيخليه أسرع في بعض السيناريوهات وأكفأ في استهلاك الMemory

خصائص الـStruct

- بيتم تخزينه في Stack Memory وده بيخليه أسرع ف Retrieve الData
- يتم تمريره By Value يعني كل مرة يتم فيها تمرير Struct بيتم إنشاء نسخة جديدة
- أسرع في الPerformance عند التعامل مع Objects صغيرة
- لا يدعم الInheritance ولاPolymorphism
- بيستخدم للObjects الصغيرة اللي مش بتحتاج تغيير كتير
- مش بيعتمد علىGarbage Collector ودا لان الData بتتحذف تلقائي بمجرد خروجها من الـScope

الفرق في طريقة الارسال Passing Mechanism

الـ Classيتم إرساله By Reference

لما تبعت Object من Class ل Method أوFunction انت بتبعت Reference لنفس الذاكرة
يعني أي تعديل عليه هيعدل الـ Original Object

الـ Structيتم إرساله By Value

لما تبعت Object من Struct بيتم نسخه بالكامل يعني أي Update عليه مش هيأثر على الأصل

ايه فيهم الافضل ف الPerformance

Struct

أسرع في الأداء لأنه بيتخزن في Stackومباشرة بيتم تحرير الMemory بمجرد انتهاء الـScope

Class

أبطأ لأنه بيتخزن فيHeap وبيعتمد على Garbage Collectorلمسح الData وده ممكن يسبب Lag مؤقت

طيب ايمته استخدم كل حاجه فيهم

✅ **استخدم Struct لو**

- حجم الData صغير
- مش بتعدل على Data بعد انشائها
- عايز Performance أسرع ومحتاج نسخة مستقلة لكل Variable

✅ **استخدم Class لو**

- الData معقدة وكبيرة
- محتاج Inheritance أوPolymorphism
- الData بتتغير باستمرار وعايز تمررها By Reference

خلاصة الكلام

Class

هو Reference Type، بيستخدم Heap Memoryوبيتم تمريره By Reference.

Struct

هو Value Type، بيستخدم Stack Memoryوبيتم تمريره By Value.

Class

مناسب ل Large Objects والمعقدة لكن Structمناسب ل Small Objects والخفيفة

Class

بيدعم Inheritance أما Struct لا يدعمه لكنه يقدر يستخدم Interfaces

Struct

أسرع في الPerformance لأنه مش بيحتاج Garbage Collector
