أكاديمية المبرمجين العرب تقدم لكم

الشرح الميسور في تقنية ASP.NET Core

خالد السعداني

دليل مختصر يجمع بين الشرح النظري والعملي

﴿ يَا أَيُّهَا الَّذِينَ آمَنُوا اتَّقُوا اللهَّ وَقُولُوا قَوْلاً سَدِيدًا

(70)

يُصْلِحْ لَكُمْ أَعْمَالَكُمْ وَيَغْفِرْ لَكُمْ ذُنُوبَكُمْ وَمَنْ يُطِعِ اللهَّ

وَرَسُولَهُ فَقَدْ فَازَ فَوْزًا عَظِيمًا (71) ﴾

سورة الأحزاب

همسة في أذنك قبل الانطلاق

هذا الكتاب الذي بين يديك الآن هو مجهود بشري قد تتخلله مجموعة من الأخطاء، أو أقول لك بصراحة: حتما ستكون فيه أخطاء نتيجة للسهو، النسيان، ضعف التركيز، أخطاء في الكتابة، أو نوع آخر من الأخطاء نحن لا نعلمه، أنت قد تعلمه أثناء سفرك مع هذا السفر!

ولأن هذا السفر الصغير أعد خصيصا لك، من أجل نفعك، فلا تتردد على صاحبه المغمور بالتماس الأعذار إن صادفت فيه أي نقص، ولو أردت أن تسدي إليه معروفا فراسله على الإيميل التالي بتفاصيل الخطأ:

```
Khalid ESSAADANI Email:
                    Khalid_essaadani@hotmail.fr
```

وسيكون لك من الشاكرين!

وإن شعرت أن هذا الكتاب قد أضاف إلى رصيدك المعرفي شيئا، فلا تبخل على صاحب الكتاب ووالديه وسائر المسلمين أجمعين بدعوة بظهر الغيب، رفع الله قدرك في الدارين.

همسة أخيرة قبل فتح باب الكتاب:

أسأل الله عز وجل أن يجعل هذا الكتاب خالصا لوجهه الكريم، وأن ينفعك به نفعا يختصر عليك الوقت والجهد.

دام لكم البشر والفرح!

خالد السعداني، في 15/أغسطس/2019

فهرس الكتاب

فهرس الصور

مقدمة في فن الإغراء

السلام عليكم ورحمة الله وبركاته،

سعيد جدا بأن أعود إلى قرائي الكرام بعد غياب طويل من خلال بوابة هذا الكتاب، الذي سأحاول من خلاله أن أمهد لكم طريق البرمجة بتقنية ASP.NET Core، وحتى لا أكون ضيفا ثقيلا على قلوبكم، وقبل ذلك على أفهامكم، سألجأ كما جرت العادة إلى أسلوبي الكلاسيكي في جعل خطابي البرمجي مرصعا بنكهة خاصة تشد القارئ وتجعله مستمتعا ومستفيدا في الوقت عينه، لذلك اسمح لي أن أستهل كتابي هذا بعرض خبراتي التي قل نظيرها في فن التحفيز والجذب :)، وذلك من خلال طرحي للأسئلة التالية التي أتوقع أنك مثلي أيضا تقوم بطرحها:

هل فكرت في بناء مشاريع ويب بلغة سي شارب تعمل بشكل مثالي على عدة أنظمة تشغيل سواء ويندوز، ماك أو لينكس؟

هل أنت في حاجة إلى بناء تطبيقات ويب سريعة تنفذ بأداء عال؟

هل احتجت في مشاريعك إلى إنشاء كلاسات غير مترابطة فيما بينها من أجل جعل هذه المشاريع قابلة للاختبار Testable وقابلة للصيانة Maintainable فاضطررت إلى استعمال مكتبات خارجية مثل Autofac لتدبير أمر هذه الارتباطات؟

أو اضطررت إلى بناء نظام يعتمد مبدأ Dependency Injection بنفسك، فوجدت أن هذا الجانب يستغرق منك وقتا طويلا وجهدا كبيرا، وصرت تفكر في تقنية تحتوي على آلية تلقائية لتدبير الارتباطات أو ما يعرف ب Dependency Injection Container؟

هل وجدت صعوبة في فهم السؤال السابق :) انتقل للسؤال التالي ولا تبالي، سأستر عليك!

هل تجد صعوبة في إدارة المكتبات التي تستعملها في مشروعك من قبيل jQuery و Bootstrap وتضطر إلى تحميلها وإعدادها يدويا، وتفكر في تقنية تسمح لك بإدارة هذه المكتبات وغيرها بأسلوب متقدم وتلقائي؟

هل تحتاج إلى مشاركة الكلاسات التي تكتبها بين عدة مشاريع، بحيث يمكنك استعمال نفس الكلاس في مشروع ويب، سطح مكتب أو حتى موبايل؟

عند الانتهاء من تطوير المشروع، هل تجد صعوبة بالغة في نشره على الانترنت أو على سيرفر محلي، وصرت تبحث عن تقنية توفر عليك كل هذا العناء وتضمن لك نشر الموقع بأريحية على سيرفر محلي أو على الانترنت؟

إن كنت تبحث عن هذه المزايا وغيرها فإن هذا الكتاب يناسبك بشدة، لأننا سنتعرف من خلاله على تقنية قوية جدا تتميز بكل ما ذكرنا وزيادة، عن تقنية ASP.NET Core أحدثك.

قبل ذلك دعنا نتعارف، اسمي خالد السعداني، ميكروسوفت MVP، وخبير برمجي في تقنيات الدوت نيت، سأرافقك في هذا الكتاب من بدايته حتى نهايته إن شاء الله، على أمل أن تحصل الفائدة المرغوبة، فكن على أتم الاستعداد.

أعلم أنني كنت فاشلا في فن الجذب، لكن لا عليك من هذا، استر علي الله يستر عليك (ولا تنس أيضا أنني سترت عليك في موضوع Dependency Injection)، وبالمقابل أعدك بوجبة شروحات دسمة.

دام لك البشر والفرح!

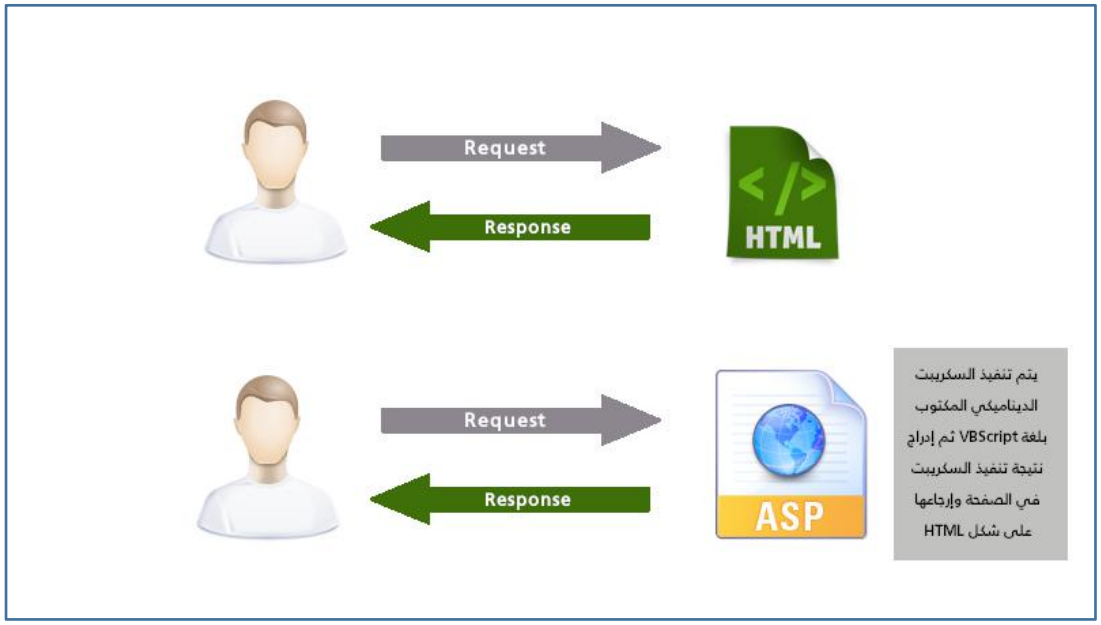
الحياة في ميكروسوفت قبل ASP.NET Core

قبل أن تصدر ميكروسوفت إطار العمل ASP.NET Core الذي يسمح للمطورين ببناء تطبيقات ويب حديثة، كانت قد أصدرت عدة حزم قبل ذلك، كان أولها هو Classic ASP وتحديدا عام 1996، ثم بعد ذلك ومع ظهور إطار العمل دوت نيت فريموورك أصدرت ASP.NET Web Forms عام 2002، ثم في عام 2009 قامت بإصدار إطار العمل ASP.NET MVC، ثم أصدرت إطار العمل ASP.NET Web API، ثم بعد ذلك اضطرت ميكروسوفت إلى إعادة كتابة إطار العمل دوت نيت فريموورك من الصفر لأسباب سنبينها لاحقا، مما دفعها إلى بناء تقنية جديدة لتطوير تطبيقات الويب تتبنى نفس الأسلوب الحديث، فقامت بإصدار ASP.NET Core عام 2016.

في الفصول المقبلة سنتعرف على كل تقنية من التقنيات التي ذكرنا على حدة، من أجل أن نفهم كيف وصلنا إلى ASP.NET Core وما الذي يميز هذا الإطار عن غيره من أطر العمل السابقة.

بدأت الحفلة مع Classic ASP

تعد Classic ASP أو Classic Active Server Pages أول إطار عمل من ميكروسوفت يسمح للمطورين ببناء تطبيقات الويب، وظهر عام 1996، وكان يسمح للمطورين ببناء صفحات الويب بأسلوب يجمع بين HTML Markup و Dynamic Server-Side Scripting، أما عن طريقة اشتغال هذا الإطار فكانت على الشكل التالي:



الصورة 1 - خطاطة تنفيذ الصفحات فيClassic ASP

حينما يرسل المستخدم طلبا إلى صفحة ثابتة من نوع HTML، يقوم السيرفر بسهولة بإرجاع الملف المطلوب وعرضه في المتصفح.

وحينما يرسل المستخدم طلبا إلى صفحة من نوع ASP File تحتوي على أوامر HTML بالإضافة إلى أوامر السيرفر، فإن محرك ASP أو ما يعرف ب ASP Engine، يقوم بمعالجة الطلب، ويقرأ ملف ASP المطلوب، ثم يقوم بتنفيذ السكريبت الديناميكي المكتوب بلغة VBScript ويقوم بإدراج نتيجة تنفيذ السكريبت في الصفحة، ويتم بعد ذلك إرجاعها إلى المستخدم على شكل HTML فقط ليستعرضها على متصفحه.

ونظرا لغياب التنظيم في هذا الإطار الذي يجعل الأوامر البرمجية مكتوبة بشكل غير مرتب، وبالتالي من الصعب اختبارها، أو صيانتها، بالإضافة إلى رغبة ميكروسوفت في توسيع مجال نفوذها في تخصص تطوير الويب، قررت أن تأتي بمقاربة جديدة لجذب مجموعة جديدة من المطورين، هذه المقاربة جاءت على هيئة إطار عمل جديد اقترحت له اسم ASP.NET Web Forms.

غريزة حب النفوذ مع ASP.NET Web Forms

لأن شركة ميكروسوفت تسعى للحصول على الجزء الأكبر من مجال البرمجيات في العالم، فإن همها الشاغل هو كيفية إغراء المطورين ليستعملوا أدواتها، ومن ثم بسط نفوذها وتوسيع دائرة مستعمليها، فكان لزاما عليها أن تأتي بإطار عمل جديد يستهوي المطورين الذين لا يشتغلون في مجال الويب وذلك عبر بناء إطار عمل قريب جدا من أطر العمل التي يستعملها هؤلاء المطورون لبناء تطبيقات سطح المكتب Desktop Apps.

فجاءت فكرة إصدار إطار العمل ASP.NET Web Forms كأداة تسمح للمطورين الذين يشتغلون في مجال برمجة سطح المكتب بالانتقال السهل والسلس إلى مجال تطوير برمجيات الويب، وارتأت ميكروسوفت في هذا الانتقال أن تحافظ على أبرز ملامح مجال تطوير تطبيقات سطح المكتب، فكان هذا الإطار الجديد ASP.NET Web Forms الذي خرج إلى الوجود عام 2002 مع حزمة الدوت نيت فريموورك، يجمع بين قدرته الرهيبة في تسهيل عملية بناء تطبيقات ويب حديثة، وبين حفاظه على معالم تطوير تطبيقات سطح المكتب، حيث بإمكان المطورين أن يصمموا صفحات الويب بأسلوب مماثل للأسلوب المعتمد في بناء تطبيقات سطح المكتب، إذ بإمكانهم إضافة الصفحات بنفس طريقة إضافة النوافذ Forms، وكذلك بناء الصفحات عبر سحب الأدوات من Toolbox بنفس الطريقة المتبعة في سطح المكتب، وكذلك اقتناص الأحداث والتعامل معها، بالإضافة إلى أمور أخرى.

لكن مع مرور الزمن، وبعد أن تمرس هؤلاء المطورون في مجال تطوير تطبيقات الويب، وجدوا أن هذا الإطار يحد إمكانياتهم، ولا يسمح لهم بالتحكم أكثر في صفحات HTML، ولأن الويب تطور أيضا مع ظهور تقنيات قوية، ومع كثرة مكتبات الجافاسكريبت التي أصبحت تهدد عرش ASP.NET، هذا بالإضافة إلى مشاكل أخرى من قبيل أن هذا الإطار مرتبط بشدة ببيئة التطوير فيجوال ستوديو، وكذلك مرتبط بالدوت نيت فريموورك، والأكثر من ذلك أنه من غير الممكن تحديث هذا الإطار لأنه مرتبط ب IIS، وهذا الأخير مرتبط بنظام التشغيل، مما يعني أنه إذا أرادت ميكروسوفت أن تطور إطار العمل ASP.NET Web Forms فعليها أن تطور أيضا نظام IIS، وإذا طورت نظام IIS عليها أن تطور نظام تشغيل جديد مع كل إصدار أو أن توفر تحديثا للنظام، وهو أمر مكلف وصعب.

للتغلب على هذه المشاكل، قررت ميكروسوفت أن تصدر إطار عمل جديد عام 2009 اسمه ASP.NET MVC.

التأثر بالمحيط الخارجي وميلاد ASP.NET MVC

عام 2009 أصدرت ميكروسوفت إطارها الجديد الذي اختارت له اسم ASP.NET MVC، وأتى هذا الإسم كتوصيف دقيق لنموذج التصميم Design Pattern الذي يعتمده هذا الإطار، والمعروف ب MVC اختصارا ل Model, View and Controller ، وهو نموذج يسمح بفصل المهام Separation of concerns عبر تفويض كل وحدة للقيام بعمل محدد، وكذلك للسماح بإجراء الاختبارات وجعل المشروع قابلا للصيانة والتحديث، ويرتكز نمط MVC بالأساس على المكونات التالية:

- **Model:** عبارة عن كلاس يحتوي على البيانات التي يتعامل معها التطبيق
- **Controller:** عبارة عن كلاس يستقبل الطلبات القادمة من المستخدمين، ثم يقوم بتنفيذها وإرجاع النتيجة إلى View
- **View:** عبارة عن HTML template خاصة بعرض البيانات القادمة من Controller

إجمالا يمكننا القول، أن بنية MVC تعمل كما يلي:

يقوم المستخدم بإرسال HTTP Request إلى Controller، ولأن هذا الأخير عبارة عن كلاس تحتوي على مجموعة من الوظائف التي تسمى Actions، فإنه يقوم بتحديد Action المناسبة لل HTTP Request القادم، ثم يقوم بتنفيذ هذه Action التي يمكنها أن تجلب البيانات من Model أو ترسل إليه البيانات، بعد ذلك تقوم هذه Action الموجودة في Controller بتحديد View اللازمة من أجل عرض البيانات المرجعة على هيئة HTML.

على الرغم من أن إطار العمل ASP.NET MVC مثل قفزة نوعية لميكروسوفت في مجال تطوير تطبيقات الويب الحديثة، إلا أن مشكل الارتباط مع نظام الويندوز، ومع الدوت نيت فريموورك، جعله حكرا على مطوري الويندوز، مما حدا بميكروسوفت إلى تطوير إطار عمل جديد يعمل على عدة أنظمة وغير مرتبط بالدوت نيت فريموورك أسمته ASP.NET Core وهو موضوع هذه الكتاب.

التكيف مع التطور التقني مع ASP.NET Web API

لكي تواكب ميكروسوفت التطورات التقنية التي عرفها العالم في الآونة الأخيرة، حيث صار بإمكان المستخدمين الوصول إلى البيانات عبر أجهزتهم اللوحية، وهواتفهم الذكية، بل تجاوز الأمر ذلك لنصير في عالم يعج بالأجهزة الذكية التي تحتاج إلى الاتصال ب Remote API من أجل قراءة البيانات أو من أجل إرسال البيانات، فصرنا نجد أجهزة تلفاز ذكية، ألات ذكية للغسيل، سيارات ذكية، وغيرها، فيما صار يعرف بأنترنت الأشياء Internet of Things.

أمام هذا التطور المهول كان لزاما على ميكروسوفت أن تستثمر في مجال بناء خدمات ويب قابلة للوصول من مختلف الأجهزة، عبر الاستفادة من أسلوب REST الذي يسمح بالاستفادة من مزايا HTTP من أجل القيام بعمليات القراءة والكتابة بواسطة خدمات الويب، وتسمى هذه الخدمات التي تتبنى أسلوب REST ب RESTful services، ومن أجل بناء هذا النوع من الخدمات قدمت ميكروسوفت إطار العمل الجديد ASP.NET Web API الذي يتبنى أسلوب REST والذي يسمح ببناء تطبيقات يمكن لكافة الأجهزة الذكية أن تتواصل معها وتستفيد منها.

أسلوب Razor ورفاهية كتابة الأوامر

من أجل تسهيل عملية بناء الصفحات، وجعل دمج أوامر السيرفر مع أوامر HTML ممكنة بطريقة سلسة، سمحت ميكروسوفت للمطورين باستعمال أسلوب Razor والذي يمثل إضافة قوية حيث صار بإمكاننا أن نكتب أوامر بلغة سي شارب في الصفحات، وعند استدعاء هذه الصفحات يتم تنفيذ هذه الأوامر وإدراج النتيجة في صفحة HTML ومن ثم إرجاعها إلى المستخدم.

لتمييز أوامر Razor عن HTML Markup، فإننا نستعمل الرمز @ قبل التعليمات، حيث بإمكاننا على سبيل المثال عرض التاريخ في صفحة ويب باستعمال Razor كما يلي:

```
Razor Snippets:
<p>@DateTime.Now</p>
```

حينما يكون عندنا بلوك من الكود نحتاج إلى كتابته بأسلوب Razor فعلينا أن نستعمل المعقوفات كما يبين المثال التالي:

```
Razor Snippets:
@{
    var name = "Khalid ESSAADANI";
}
<p>Hello, @name</p>
@{
    name = "Said HAMRI";
}
<p>@name</p>
```

ستكون لنا صولات وجولات مع هذا الأسلوب الشيق الذي يسهل بناء صفحات الويب في ASP.NET.

لم الشتات مع OWIN و Katana

نظرا لكثرة أطر العمل التي تقدمها ميكروسوفت من أجل بناء تطبيقات الويب، ولأن كل إطار عمل له خصوصياته التي تميزه عن غيره، فمثلا أطر العمل ASP.NET Web Forms و ASP.NET MVC مرتبطة بالدوت نيت فريموورك وب System.web assembly، بينما ASP.NET Web API يأتي بشكل مستقل وغير مرتبط بالدوت نيت فريموورك و ب System.web assembly، وذلك

راجع بالأساس إلى أن هذا الإطار تم تطويره من قبل فريق آخر وليس من قبل نفس الفريق الذي طور ASP.NET MVC.

ومن المعلوم أن تطبيقات الويب الحديثة ممكن أن تجمع بين عدة مكونات مختلفة من عدة أطر عمل، فممكن للتطبيق أن يتعامل مع الصفحات، أن يتعامل مع Web API، أن يتعامل مع Real Time Notifications كما هو الحال مع إطار العمل SignalR، وغيرها، ولأن كل ميزة من المزايا المذكورة تنتمي إلى إطار عمل معين، وكل إطار عمل له خصوصياته كما ذكرنا، فهذا يعني أن كل مكون سيعمل بشكل مستقل، لذلك قررت ميكروسوفت أن تنشئ طبقة فاصلة Abstraction بين الويب سيرفر وبين أطر العمل، وذلك بغرض تسهيل عملية إنشاء مكونات جديدة بمواصفات هذه الطبقة الفاصلة، وذلك بغرض تسهيل عملية نقل التطبيقات بين الخوادم بل وبين عدة أنظمة تشغيل، حيث يصير تطبيق الويب غير مرتبط بويب سيرفر معين أو نظام تشغيل معين، وتمت تسمية هذه الطبقة الفاصلة ب OWIN اختصارا ل Open Web Interface، وهي عبارة عن توصيف Specification يبين كيف ينبغي أن يكون الويب سيرفر وكيف يكون التطبيق من أجل ضمان أن يشتغل التطبيق دون مشاكل ومن أجل إمكانية نقله إلى بيئات جديدة لم تكن مدعومة من قبل.

ولا يحتوي OWIN على أية أدوات، فهو مجرد توصيف كما ذكرنا، وللاستفادة من هذا التوصيف، قام فريق من مطوري ميكروسوفت ببناء إطار عمل جديد أسموه Katana، هذا الأخير الذي يمثل جسرا بين أطر العمل الخاصة بميكروسوفت وبين OWIN Specification، ونجحت Katana في جعل كل من Web API و SignalR موافقين لمواصفات OWIN، لكن لأن ASP.NET MVC و ASP.NET Web Forms مرتبطان ب System.web.dll فعملية جعلهما موافقين لمواصفات OWIN صعبة جدا، مما جعل ميكروسوفت تقوم بإدراج مميزات Katana في ASP.NET 5 والتي كانت تسمى أيضا vNext ، والتي سميت في وقت لاحق ب ASP.NET Core.

عبر هذا التدرج وصلنا إلى إطار العمل ASP.NET Core الذي سنتعرف عليه أكثر في غضون هذا الكتاب، فكونوا على أتم الاستعداد.

ثم كانت ASP.NET Core

تمت كتابة ASP.NET من الصفر لكي تلبي متطلبات مبرمجي الويب، حيث وضعت ميكروسوفت نصب عينيها أن هذا الفريموورك الجديد ينبغي أن يراعي الأمور التالية:

1. **أن يكون متعدد البيئة Cross-platform:** بحيث يمكن الاشتغال على مشاريع من هذا النوع على مختلف أنظمة التشغيل: ويندوز، ماك، لينكس.
2. **أن يكون مفتوح المصدر بالكامل Open source:** لفتح الباب أمام مجتمعات التطوير للإسهام في هذا الإطار، بحيث يستطيع الجميع تحميل الشفرة المصدرية والاشتغال عليها.

3. **أن يكون جزئيا Modular:** بحيث يتضمن المشروع فقط المكتبات التي يستعملها، والتي تأتي على شكل حزم يمكننا الوصول إليها عبر Nuget Packages، على خلاف الدوت نيت فريموورك الذي يعد وجوده إلزاميا حتى وإن كان التطبيق صغيرا جدا.
4. **لا يفرض قيود بين الإصدارات:** بمعنى يمكن للتطبيق أن يشتغل على إصدارات أحدث أو أقدم دون مشاكل، لأن المكتبات المستعملة تأتي على شكل Nuget Packages.
5. **تحتوي مسبقا وبشكل جاهز على نظام لإدارة الارتباطات Dependency Injection System:** مما يسمح بإضعاف الترابط بين مكونات المشروع وجعلها غير مرتبطة ببعضها البعض مما يضمن سهولة الاختبارات والقدرة على التحديث والصيانة.
6. تتوفر تقنية ASP.NET Core على إطار عمل جديد يدمج بين ASP.NET MVC و ASPNET Web API، بحيث يمكننا في المشروع الواحد أن نشتغل على هذين المكونين دون مشاكل لأنه تم دمجهما معا.
7. **الدعم الكامل لل Cloud أو ما يعرف ب Cloud-Optimized:** تم تصميم ASP.NET Core لتتلاءم مع Cloud بسهولة، حيث يستطيع المطورون بناء Cloud-Based Web Applications، وتوفير مكتبات API لاستهلاكها من قبل تطبيقات أخرى، مع إمكانية الاشتغال على أي نظام تشغيل سواء كان ويندوز، ماك أو لينكس.

ظهرت أول نسخة من ASP.NET Core في صيف 2016 وتحديدا في شهر يونيو، والإصدار الحالي هو الإصدار ASP.NET Core 3.0.

وللإطار العام كلمة: عن الدوت نيت كور نتكلم

الدوت نيت كور هو إطار العمل الجديد الذي أصدرته ميكروسوفت لحل المشاكل التي كان يعرفها الفريموورك الذي كان قبله والمعروف بالدوت نيت فريموورك، ويتميز الدوت نيت كور بكونه متعدد البيئة، بمعنى نستطيع استعماله في تطوير المشاريع على مختلف الأنظمة سواء ويندوز، ماك أو لينكس، بالإضافة إلى أنه Modular مما يعني أنه يسمح لك باستعمال المكتبات التي ستحتاجها فقط، بالإضافة إلى أنه مصمم ليتوافق مع بناء تطبيقات متناغمة مع Cloud لذلك يعرف دوت نيت كور بأنه Cloud-Optimized، ويتميز أيضا بكونه مفتوح المصدر، إذ يمكنك الاطلاع على Repositories الخاصة بهذا الإطار وعمل Clone لما تحتاجه ويمكنك الإسهام في التطوير أيضا.

هنالك من يخلط بين الدوت نيت كور وبين ASP.NET Core، مع العلم أنه لا وجه للمقارنة بينهما، لأن ASP.NET Core تعد جزء من إطار العمل دوت نيت كور، وعلاقتها به مثل علاقة ASP.NET MVC مع الدوت نيت فريموورك، ويشتمل الدوت نيت كور بالإضافة إلى ASP.NET

Core على مكتبة الفئات والتي تسمى CoreFX ويحتوي كذلك على فضاء للتنفيذ يسمى CoreCLR، بالإضافة إلى مترجم Compiler يسمى Roslyn.

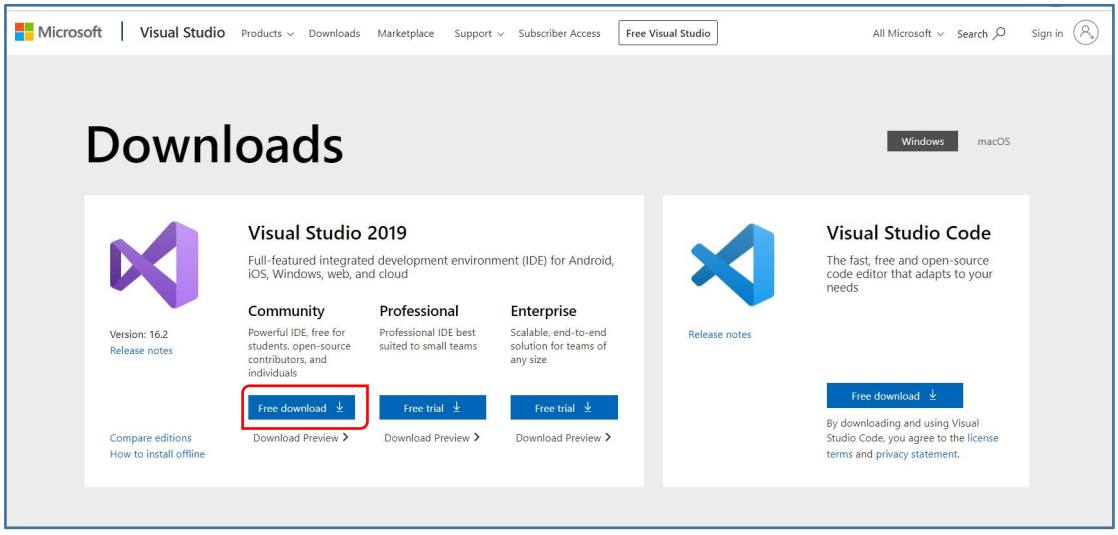
عند تثبيت الدوت نيت كور يقوم تلقائيا بتنصيب هذه المكونات التي ذكرناها بما فيها ASP.NET Core، يتبقى فقط أن ننصب بيئة للتطوير، ويستحسن أن نستعمل الفيجوال ستوديو.

تنزيل وتثبيت الفيجوال ستوديو

بعد أن تعرفنا في الفقرات الماضية على مختلف أطر العمل التي أصدرتها ميكروسوفت للسماح للمطورين ببناء مشاريع تعمل على الويب، وتعرفنا على الفرق بين الدوت نيت كور وبين ASP.NET Core، وصلنا إلى مرحلة تنزيل وتثبيت الأدوات اللازمة، سنحتاج إلى برنامج فيجوال ستوديو، والذي يمكننا الوصول إليه لتنزيله من الرابط التالي:

```
Download Visual Studio:

https://visualstudio.microsoft.com/downloads/
```

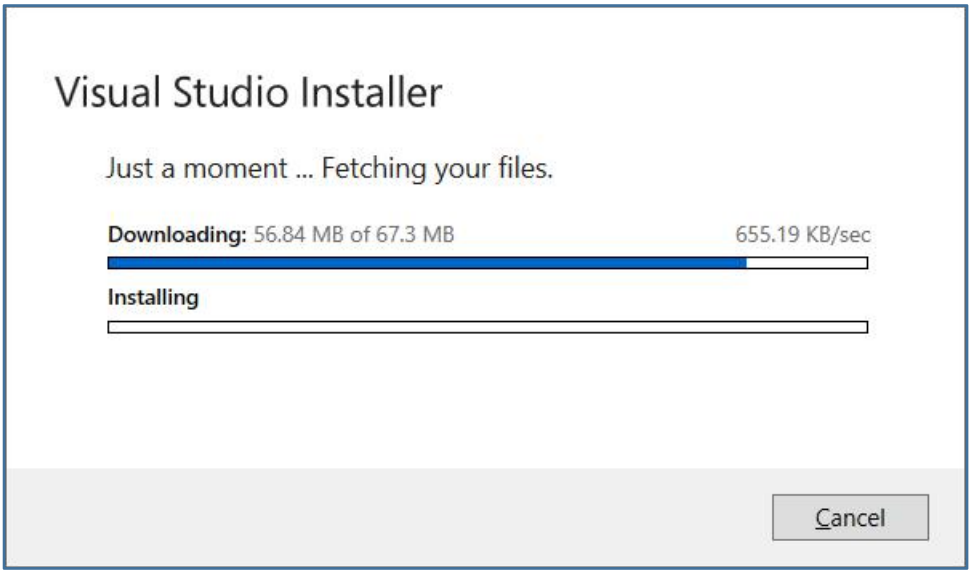
عند الدخول إلى الرابط أعلاه، ستطالعنا الصورة أسفله:



الصورة 2 - واجهة تحميل برنامج فيجوال ستوديو 2019

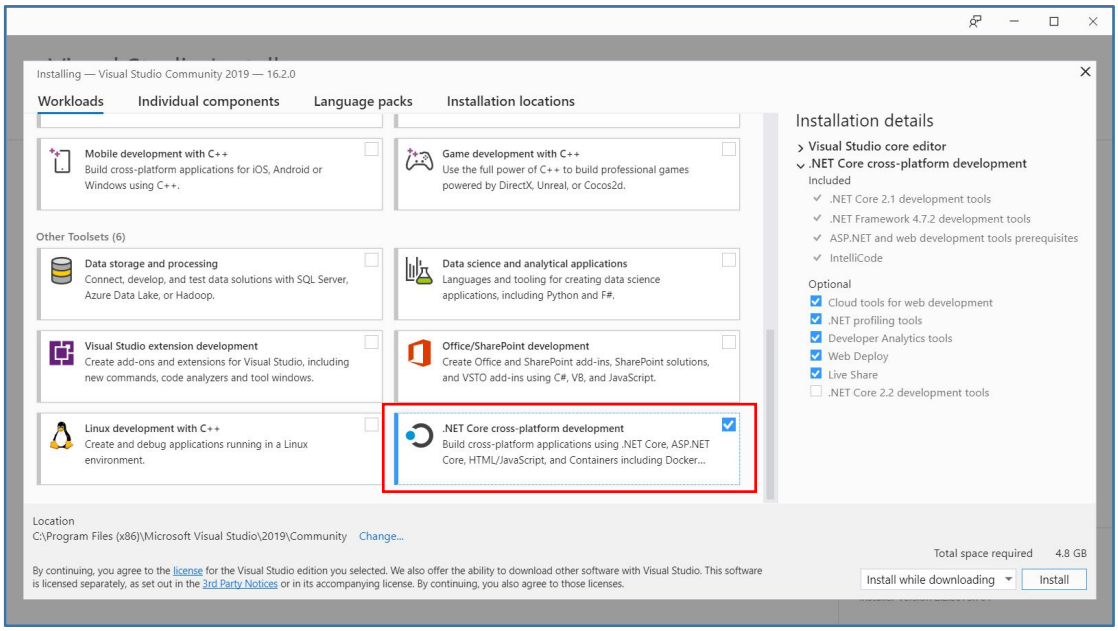
كما تلاحظون يوجد لدينا ثلاث إصدارات من الفيجوال ستوديو، سنقوم بتنزيل النسخة Community باعتبارها النسخة المجانية، للتنزيل سنضغط على الزر الأزرق Free download، ليتم مباشرة بعد ذلك تنزيل ملف تنفيذي باسم vs_community، نضغط مرتين على هذا الملف لبدأ عملية التنصيب:

سيتم تنزيل بعض الملفات كما تبين الصورة التالية:



الصورة 3 - تنزيل وتثبيت ملفات فيجوال ستوديو

بعد ذلك سيعطيك مختلف Work Loads التي قد تحتاجها، قم بتحديد ما تشاء، لكن للاشتغال على الدوت نيت كور، يكفيك أن تحدد .NET Core cross-platform development. كما تبين الصورة التالية:

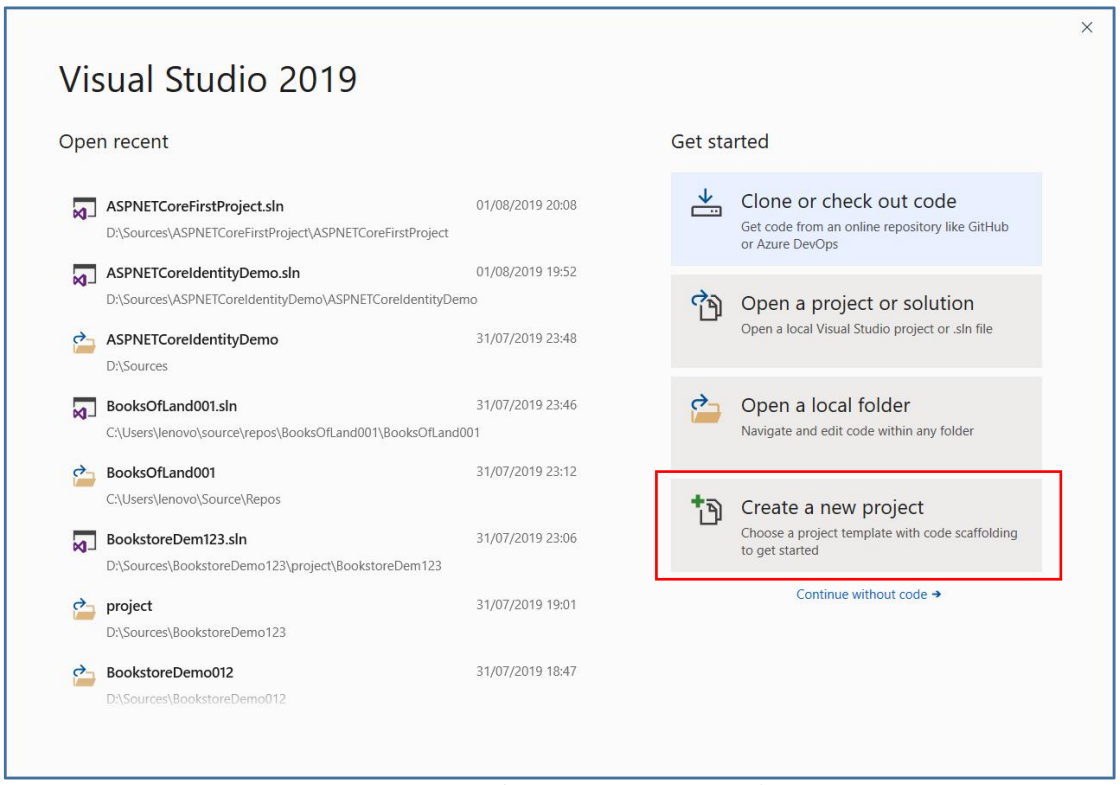


الصورة 4 - مكونات فيجوال ستوديو 2019

ثم اضغط على الزر Install لبدأ عملية التثبيت، بعد الانتهاء قم بتشغيل الفيجوال ستوديو، لأننا في الفقرة المقبلة سوف نقوم ببناء أول تطبيق لنا مع تقنية ASP.NET Core، فكن على أتم الاستعداد.

مسافة الألف تطبيق تبدأ بإنشاء أول مشروع

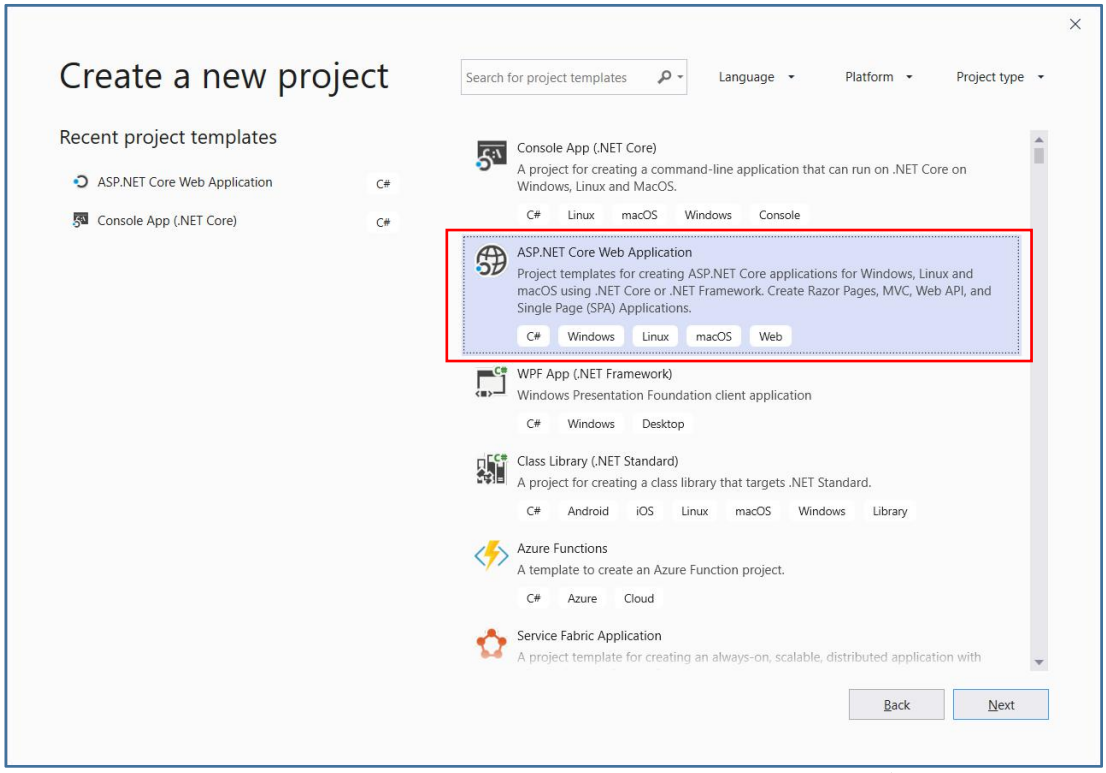
عندما تدخل إلى الفيجوال ستوديو ستطالعك الواجهة التالية:



الصورة 5 - شاشة بداية فيجوال ستوديو 2019

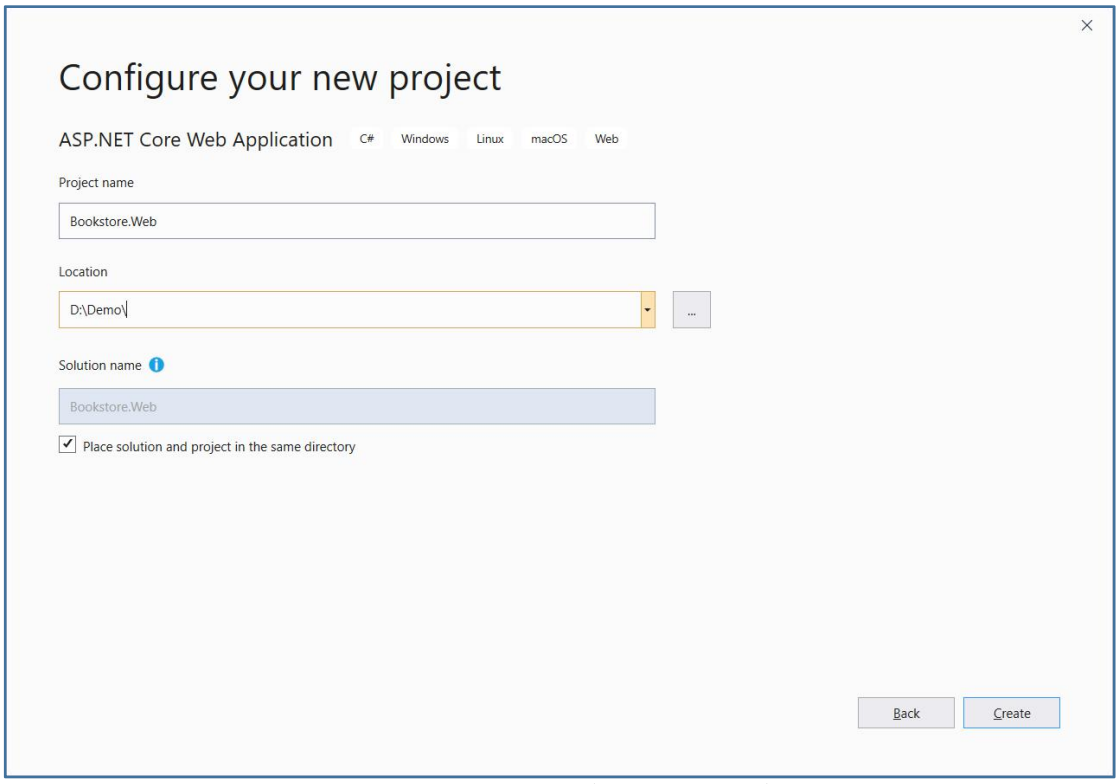
الشاشة كما تلاحظ مقسمة إلى جزأين، الجزأ الأول يعرض آخر المشاريع التي قمت بالاشتغال عليها، والجزء الثاني يحتوي على الاختيارات التالية:

- Clone or check out code: وتسمح لك بعمل clone ل repository معين مرفوع على منصة GitHub مثلا أو على منصة Azure DevOps الخاصة بميكروسوفت.
- Open a project or solution: تسمح لك بفتح مشروع موجود على حاسوبك.
- Open a local folder: تسمح لك بفتح مجلد معين والتعديل على الملفات المصدرية الموجودة عليه.
- Create a new project: وتسمح لنا بإنشاء مشروع جديد من الصفر، وهو الخيار الذي سنتبعه في هذا الفصل، لذلك لنقم بالضغط على هذا الأمر لتطالعنا الشاشة التالية:



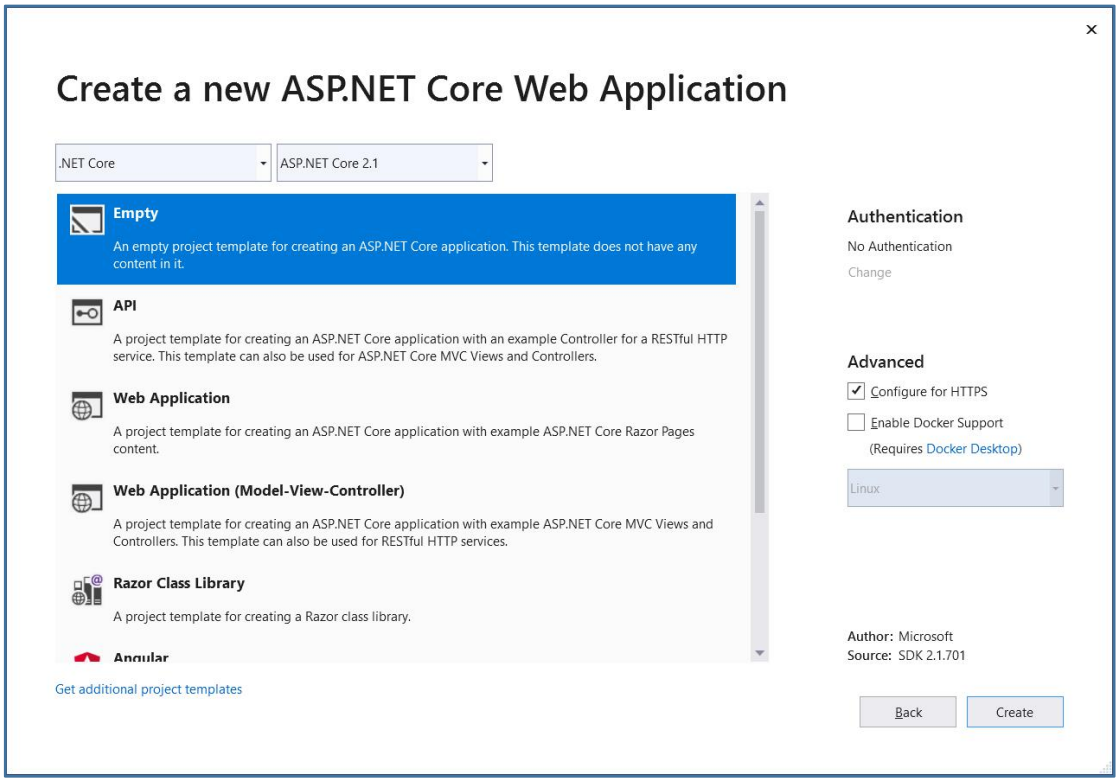
الصورة 6 - إنشاء مشروع جديد من نوع ASP.NET Core Web App

من الشاشة أعلاه نستطيع أن ننشئ عدة أنواع من المشاريع التي يوفرها الفيجوال ستوديو، لبناء مشروع ويب بتقنية ASP.NET Core لنقم باحتيار ASP.NET Core Web Application، ثم نضغط على الزر Next لتطالعنا الشاشة الآتية:



الصورة 7 - تسمية المشروع وتحديد مسار حفظه

لنقم بإدخال اسم المشروع واختيار مسار حفظه، ثم نضغط على الزر Create، بعد ذلك ستظهر لنا الواجهة التالية:



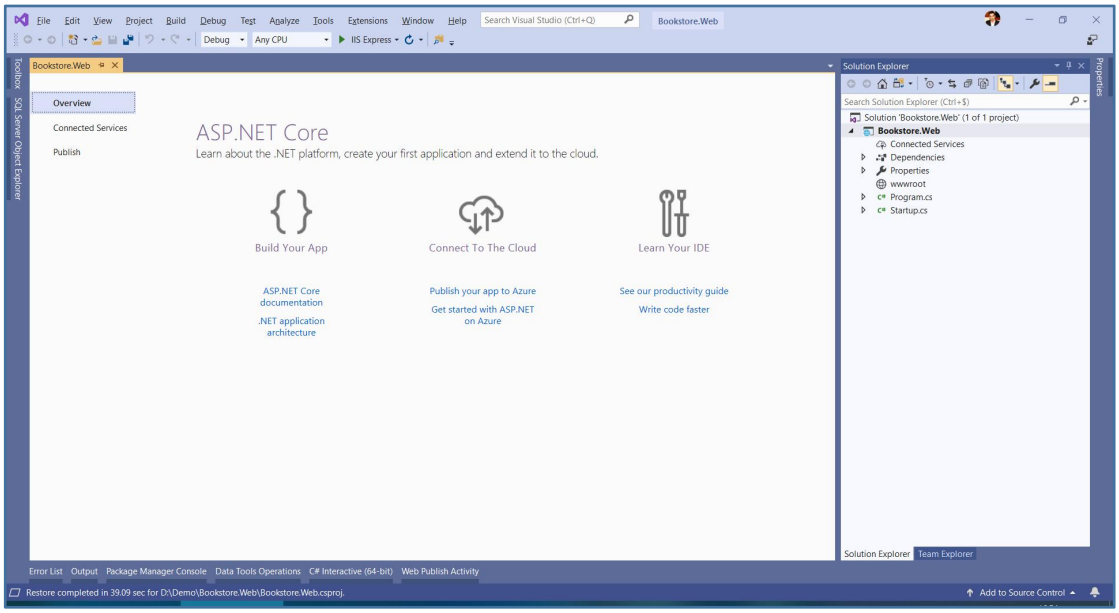
الصورة 8 - اختيار شكل المشروع

الواجهة أعلاه تسمح لنا باختيار نوع المشروع والاختيارات الواردة كما يلي:

1. **Empty:** لإنشاء مشروع ويب من نوع ASP.NET Core فارغ
2. **API:** لإنشاء مشروع يسمح ببناء خدمات ويب من نوع RESTful HTTP.
3. **Web Application:** تسمح لنا ببناء مشروع ويب قائم على صفحات Razor.
4. **Web Application (MVC):** يسمح لنا ببناء مشروع ويب من نوع ASP.NET Core MVC
5. **Razor Class Library:** من أجل بناء مكتبة خاصة ب Razor لاستعمالها في مشاريع الويب.
6. **Angular:** لبناء تطبيق ويب متكامل يجمع بين ASP.NET Core و Angular
7. **React.js:** يسمح لنا ببناء تطبيق ويب متكامل يجمع بين ASP.NET Core و React.js
8. **React.js and Redux:** لبناء تطبيق ويب يجمع بين ASP.NET Core و React.js & Redux.

لاحظ في القائمة المنسدلة في أعلى الواجهة أننا نستطيع اختيار إطار العمل إما الدوت نيت كور أو الدوت نيت فريموورك، بالإضافة إلى الإصدار الذي نود استعماله، سنترك الاختيارات على ماهي عليه، ثم سنقوم باختيار القالب الفارغ Empty وهو الأول في القائمة، ثم نضغط على الزر Create، ليتم إنشاء المشروع.

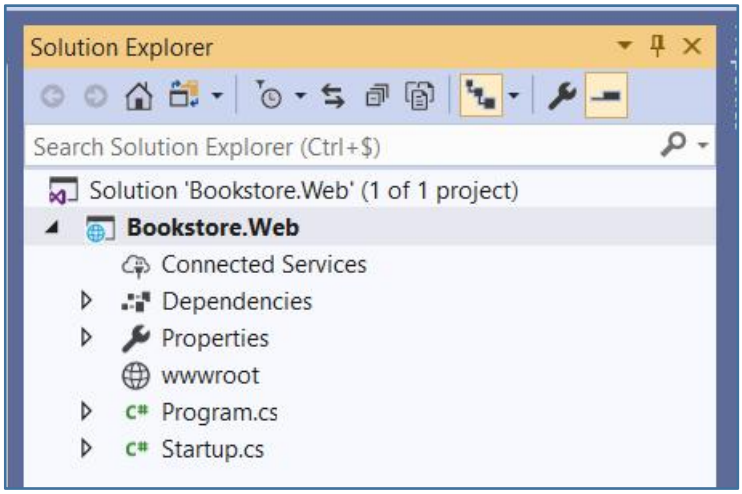
عند الانتهاء من إعداد بيئة التطوير فيجوال ستوديو، ستطالعك الواجهة الرئيسية للبرنامج وهي كما يلي:



الصورة 9 - شاشة بداية المشروع

متصفح الملفات: عكازة الأعمى

على يمين الشاشة ستجد متصفح الملفات واسمه Solution Explorer، إذا لم تجده، فاذهب إلى القائمة العليا View، ثم قم باختيار Solution Explorer لتظهر لك واجهته وهي كما يلي:



الصورة 10 - واجهة متصفح الملفات

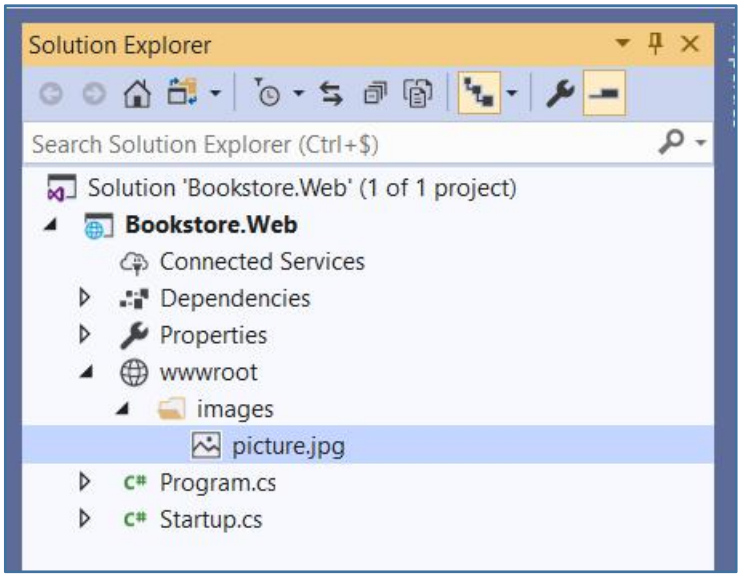
ستجد أسفل ملف Solution اسم المشروع الذي قمت بإنشائه، وأسفل المشروع ستجد عدة عناصر وهي كما يلي:

1. Connected Services: من أجل الاتصال بالخدمات الجاهزة من قبيل Azure، أو خدمات WCF وغيرها.

2. Dependencies: تحتوي على المكتبات التي تستعملها في مشروعك.
3. Properties: تعرض خصائص المشروع مع إمكانية التعديل عليها.
4. Wwwroot: خاص بالملفات الثابتة كالصور وملفات css و javascript وغيرها.
5. Program.cs: سنتعرف عليه بالتفصيل لاحقا، لكن مبدئيا يكفي أن نقول أنه يمثل نقطة بداية التنفيذ للمشروع.
6. Startup.cs: هذا الكلاس هو نقطة بداية تنفيذ ASP.NET Core pipeline وسيأتي شرحه بالتفصيل لاحقا.

مجلد wwwroot خزانتك التي عليك ترتيبها

هذا هو المجلد الأساسي في ASP.NET Core والذي نستطيع أن ننشئ مجلدات فرعية داخله من أجل تخزين الملفات الثابتة كالصور، ملفات css، ملفات javascript، وغيرها، بما أن هذا المجلد هو root folder فهو يمثل المسار الأساسي للتطبيق، فلو افترضنا عندنا مجلد اسمه images يوجد داخل wwwroot، ويوجد بداخل مجلد images ملف صورة باسم picture.jpg كما تبين الصورة أسفله:



الصورة 11 - مجلد wwwroot

فيمكننا أن نصل إلى هذه الصورة عبر المسار التالي:

```
Image Path Example:

http://localhost:portNumber/images/picture.jpg
```

حيث portNumber هو رقم البورت الذي يشتغل التطبيق عليه.

ملف Program.cs باب منزل التطبيق

بالنسبة لمن له سابق خبرة مع لغة سي شارب، سيجد أن هذا الاسم ليس بغريب عليه، وكذلك محتواه، فلو دخلنا إليه سنجده يحتوي على الوظيفة main التي من المعلوم أنها تمثل نقطة بداية تنفيذ التطبيق، وهو الأمر نفسه في ASP.NET Core، فملف Program.cs يمثل نقطة بداية تنفيذ التطبيق، حيث أن تطبيق الويب في ASP.NET Core هو في الحقيقة عبارة عن تطبيق من نوع Console، وبالتالي فإن ملف Program يسمح لنا بإنشاء Host لتنفيذ تطبيق الويب عليه، ولو دخلنا إلى محتواه سنجده كما يلي:

Program.cs:

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

الوظيفة Main تقوم باستدعاء الوظيفة CreateWebHostBuilder والتي بدورها تقوم بإنشاء Web Host بإعدادات افتراضية لتشغيل تطبيق الويب عليه من خلال الوظيفة CreateDefaultBuilder.

الوظيفة CreateDefaultBuilder تقوم بإعداد ويب سيرفر يسمى Kestrel والذي يستعمل بشكل افتراضي لتنفيذ تطبيقات ASP.NET Core، كما تقوم هذه الوظيفة أيضا بالإعدادات الخاصة بالويب سيرفر IIS.

في الختام ستلاحظ أن Web Host الذي نقوم بإنشائه يستعمل الكلاس Startup والذي بدوره يقوم بتسجيل الارتباطات Dependencies المستعملة في التطبيق، ومن خلال هذا الكلاس يبدأ تنفيذ تطبيق ASP.NET Core ومن خلاله تمر Request إلى ASP.NET Core Pipeline، ولنا عودة مع هذا الملف في الفقرة المقبلة فكونوا على أتم الاستعداد.

ملف Startup.cs أو قاعة الاستقبال

في الإصدارات السابقة لل ASP.NET كانت نقطة بداية التطبيق هي ملف global.asax، مع ASP.NET Core صار الوضع مختلفا وتم الاستعاضة عن هذا الملف بملف Startup.cs والذي

بدوره يقوم بمعالجة Request Pipeline، كما يقوم أيضا بتسجيل الارتباطات Dependencies المستعملة في التطبيق، ويحتوي ملف Startup.cs على وظيفتين وهما كما يلي:

Startup.cs:

```
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
        }
    }
```

الوظيفة الأولى ConfigureService تستقبل برامتر من نوع IServiceCollection، والذي يقوم بتسجيل الخدمات services المستعملة في المشروع عبر آلية Dependency Injection والتي توجد مسبقا في ASP.NET Core، ولنا عودة مع آلية Dependency Injection في ASP.NET Core في درس مستقل لاحقا، لكن حاليا يكفيك أن تعرف أن الوظيفة ConfigureService تقوم بتسجيل Services المستعملة في المشروع، حيث أنها أول ما ينفذ عند الدخول إلى Startup class، فتقوم بتسجيل كافة services في Dependency Injection Container ليتم بعد ذلك استعمالها في التطبيق.

وعلى فكرة حينما نقول service فنحن نتحدث عن كلاس، لكن بلغة Dependency Injection نسميها service.

الوظيفة الثانية هي Configure، وهي تسمح لنا بإعداد كيفية التعامل مع الطلبات Requests في تطبيقنا، ويمكننا هنا استحضار مفهوم Middleware والتي تسمح لنا بالقيام ببعض العمليات على Request Pipeline، ويمكننا إضافة Middlewares هنا داخل الوظيفة Configure.

وستكون لنا عودة مع مفهوم Middleware في فصل لاحق إن شاء الله، فكونوا على أتم الاستعداد.

كيف يجعل Razor حياتك سعيدة

كنا قد ذكرنا سابقا أن Razor هو عبارة عن أسلوب لبناء الصفحات في ASP.NET يسمح لنا بدمج أوامر Server مع أوامر Client بطريقة سلسة، في هذه الفقرة سنتعرف أكثر على هذا الأسلوب.

أوامر Razor دائما ما تبدأ بالرمز @، عدا ذلك فالأسلوب يعتمد على لغة سي شارب في كتابة التعليمات البرمجية، لذلك اعتبر نفسك متقنا لأسلوب Razor إن كنت متقنا للغة سي شارب، تتبقى فقط بعض الإضافات المتعلقة بكيفية التعامل مع الصفحات بأسلوب Razor والتي سنراها لاحقا إن شاء الله، لكن حاليا تعالوا بنا نركز على أساسيات كتابة الأوامر البرمجية بأسلوب Razor.

في المثال التالي لاحظ معي كيف سمح لنا Razor بأن نخلط بين كود برمجي بلغة سي شارب مع Markup HTML بطريقة سلسة:

```
Razor Snippets:

<h1>Current Date: @DateTime.Now.ToLongDateString()</h1>
```

لو قمنا باستعراض الصفحة التي تحتوي على الأمر أعلاه، فسوف نحصل على النتيجة التالية:

Current Date: 05 August 2019

يمكننا أن نكتب أيضا بلوك من الأوامر البرمجية بأسلوب Razor، وهذا مثال يبين ذلك:

```
Razor Snippets:

@{
    var a = 3;
    var b = 5;

    var sum = a + b;

    <h3>Sum of a + b is: @(a+b)</h3>
}
```

عند التنفيذ سيتم جمع قيمتي المتغيرين a و b وتتم طباعة النتيجة كما يلي:

Sum of a + b is: 8

يمكننا أيضا التحقق من عبارة معينة باستعمال if statement كما يلي:

```
Razor Snippets:

@{
    var userName = "Khalid";
    var password = "123456";
```

```
    if (userName == "Khalid" && password == "123456")
    {
        <h2>Hello, @userName!</h2>
    }
    else
    {
        <h2>Invalid login attempt.</h2>
    }
}
```

في المثال أعلاه سيتم التحقق من قيمتي المتغيرين، فإن تحقق الشرط تم عرض رسالة ترحيبية، وإن لم يتحقق تم عرض رسالة مفادها أن محاولة تسجيل الدخول لم تنجح.

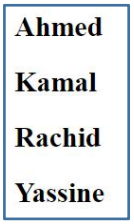
لأن الشرط في حالتنا متحقق، فعتد التنفيذ سنحصل على النتيجة التالية:

يمكننا أيضا القيام بعمليات تكرارية كما في لغة سي شارب باستعمال الحلقات، وهذا مثال يوضح ذلك:

Razor Snippets:

```
@{
    string[] names = {"Ahmed", "Kamal", "Rachid", "Yassine"};

    foreach (var name in names)
    {
        <h2>@name</h2>
    }
}
```

عند تنفيذ الصفحة أعلاه، سنلاحظ أن الآلية التكرارية تقوم بعرض جميع عناصر المصفوفة النصية names، وفيما يلي نتيجة البلوك:



يمكننا القيام بنفس العمل من خلال استعمال باقي الأساليب التكرارية الأخرى مثل for و do و while كما يلي:

Razor Snippets:

```
@{
    string[] names = { "Ahmed", "Kamal", "Rachid", "Yassine" };

    // Using for
    <h2>Using for loop</h2>
    for (int i = 0; i < names.Length; i++)
    {
        <h3>@names[i]</h3>
    }

    // Using do
    <h2>Using do loop</h2>
    var counter = 0;
    do
    {
        <h3>@names[counter]</h3>
        counter++;
    } while (counter < names.Length);



    // Using while
    <h2>Using while loop</h2>
    counter = 0;
    while (counter < names.Length)
    {
        <h3>@names[counter]</h3>
        counter++;
    }
}
```

عند التنفيذ سنلاحظ ظهور نفس النتيجة السابقة مع كل آلية تكرارية.

أحيانا نحتاج إلى التعامل مع البيانات القادمة من Controller عبر Action معينة، لنستطيع ذلك لابد من جلب مجال الأسماء الذي ينتمي إليه نوع البيانات المرجع وذلك من خلال كتابة الأمر @using كما يلي:

Razor Snippets:

```
@using Bookstore.Web.Models;
```

بعد ذلك نقوم بتعريف Model الذي سنتعامل معه في الصفحة من خلال الأمر @model كما يلي:

Razor Snippets:

```
@model IList<Book>;
```

يمكننا أيضا التصريح عن الدوال functions بأسلوب Razor، لعمل ذلك لاحظ معي المثال التالي:

```
Razor Snippets:

@functions
{
    DateTime GetCurrentDateTime()
    {
        return DateTime.Now;
    }

    string GetName()
    {
        return "Khalid";
    }

    int Sum(int a, int b)
    {
        return a + b;
    }
}
```

يمكننا استدعاء هذه الدوال بالطريقة الاعتيادية كما يلي:

```
Razor Snippets:

<h3>Current Date: @GetCurrentDateTime()</h3>
<h3>Name: @GetName()</h3>
<h3>The sum of 5 + 6 is: @Sum(5,6)</h3>
```

وفيما يلي النتيجة التي سنحصل عليها عند التنفيذ:

Current Date: 05/08/2019 11:15:29

Name: Khalid

The sum of 5 + 6 is: 11

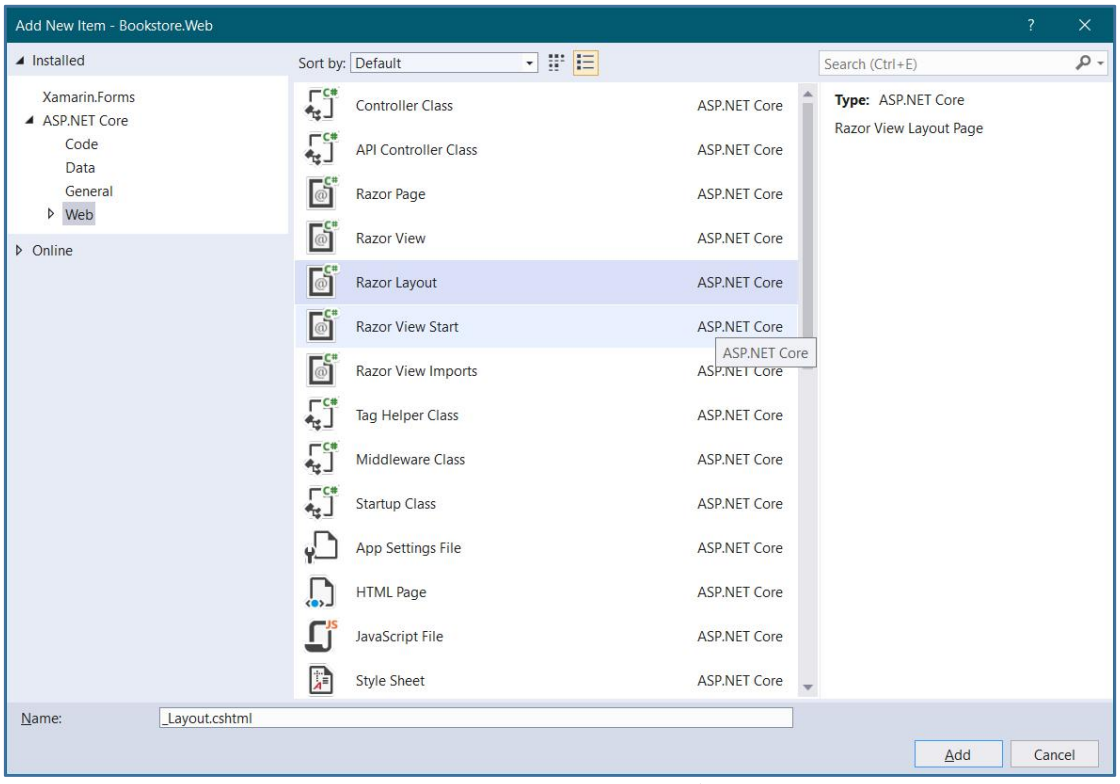
توجد عدة أمور أخرى يسنح لنا بها Razor كالأمر inherits الذي يسمح لنا بالوراثة من كلاس معين، أو الأمر inject الذي يسمح لنا بعمل injection ل service معين من service container المستعمل في ASP.NET Core من أجل تطبيق آلية Dependency Injection، ولنا عودة مع هذا المفهوم المهم في فصل لاحق إن شاء الله.

بناء الهيكل الأساسي لصفحات التطبيق ودور Layout في ذلك

عملية تصميم صفحات المواقع تستغرق وقتا إذا لم تدرس بعناية، وإذا لم يوفر الفريموورك المستعمل آليات لتسريع عملية التصميم، لذلك فإن ASP.NET تقدم لنا عدة حلول لتسهيل عملية بناء صفحات الموقع، ومن بين هذه الآليات نجد آلية Layout، هذا الأخير يسمح لنا بوضع الأجزاء التي تتكرر في كافة صفحات الموقع مثل header و footer في صفحة واحدة تسمى Layout، ثم نقوم باستدعائه في الصفحات الأخرى التي تحتاج إلى عرض نفس المكونات باستثناء تغير المحتوى.

فمثلا بدل أن نصمم عدة صفحات تحتوي على عناصر متكررة، نقوم بوضع هذه العناصر في Layout، ونغير فقط المحتوى في الصفحات التي تطبق هذا Layout، كما سنرى الآن.

تعالوا بنا ننشئ مجلدا باسم Views داخل المشروع من أجل تخزين الصفحات بداخله، ولنقم بإنشاء مجلد فرعي نسميه Shared داخل مجلد Views، داخل هذا المجلد نضغط بيمين الماوس ونختار Add new item، ثم من الواجهة التالية نقوم باختيار Razor Layout:



الصورة 12 - إضافة Layout

لنترك الاسم على ماهو عليه _Layout.cshtml، ونضغط على الزر Add، عند الدخول إلى ملف Layout ستلاحظ أنه عبارة عن صفحة HTML عادية تحتوي على بعض أوامر Razor مثل @ViewBag.Title و @RenderBody:

_Layout.cshtml:

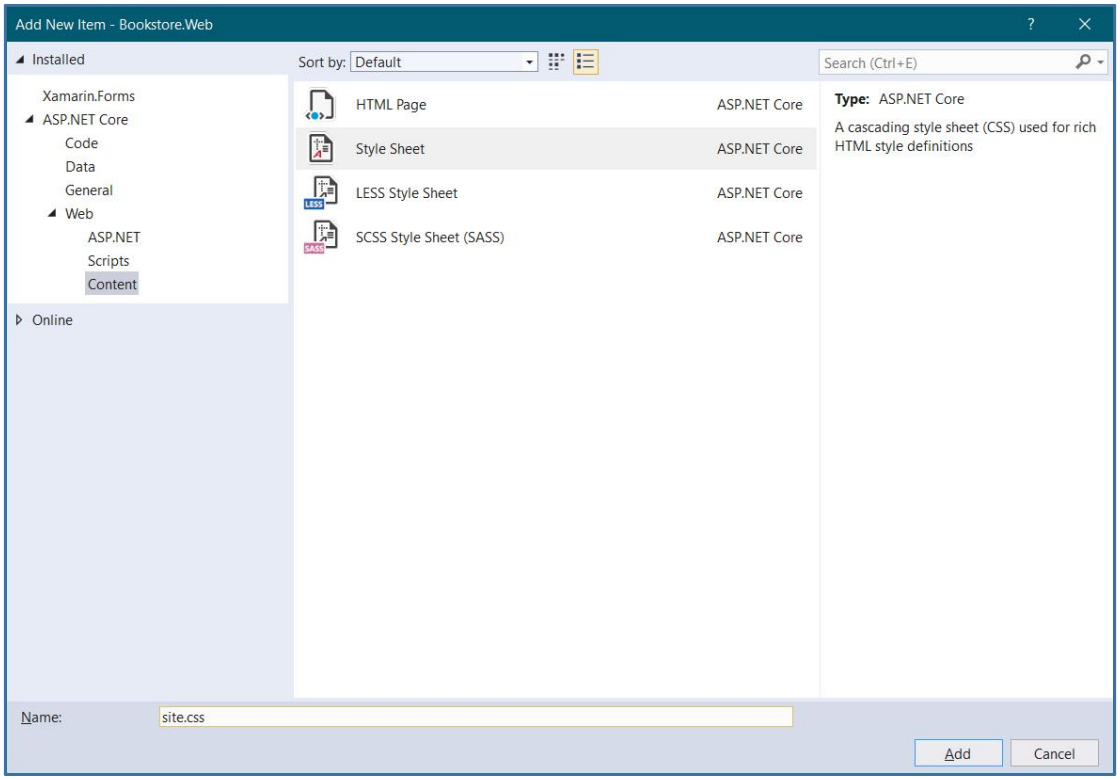
```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

الآن سنضيف إلى هذا Layout بعض المكونات مثل header و footer كما يلي:

_Layout.cshtml:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="header">
        <h1>This is the header</h1>
    </div>
    <div>
        @RenderBody()
    </div>
    <div class="footer">
        <h1>This is the footer</h1>
    </div>
</body>
</html>
```

الآن سنضيف بعض التحسينات على التصميم من خلال إنشاء كلاسات css اللازمة، لعمل ذلك، لندخل إلى مجلد wwwroot ونقم بإضافة مجلد نسميه css ثم نضيف داخله ملف من نوع css ولنسمه مثلا site.css:



الصورة 13 - إضافة ملف css

ثم نضيف الخصائص التالية لملف site.css:

site.css:

```
.header, .footer {
    background-color: #20b2aa;
    padding: 40px;
}

.header h1 {
    font-family: fantasy;
    color: crimson;
}

.footer h1 {
    font-family: cursive;
    color: aliceblue;
}
```

الآن سنقوم بإضافة رابط ملف site.css إلى Layout الذي صممناه قبل قليل:

site.css:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/css/site.css" rel="stylesheet" />
</head>
```

```
<body>
    <div class="header">
        <h1>This is the header</h1>
    </div>
    <div>
        @RenderBody()
    </div>
    <div class="footer">
        <h1>This is the footer</h1>
    </div>
</body>
</html>
```

بعد ذلك سنذهب إلى الصفحات التي نريد تطبيق هذا Layout عليها ونضيف البلوك التالي:

viewName.cshtml:

```
@{
    Layout = "_Layout";
}
```

حيث _Layout هو اسم Layout الذي أنشأناه، الآن لو عرضنا الصفحة سنجد تلقائيا بأن محتواها يظهر داخل عناصر Layout التي صممناها كما يلي:

الصورة 14 - نتيجة التنفيذ

المحتوى الثابت الخاص ب Layout بقي كما هو، الذي تغير هو المحتوى الخاص ب View والذي يمثله على مستوى Layout الأمر @RenderPage الذي يقوم بإنشاء Placeholder لاستقبال المحتوى من الصفحات التي ستطبق Layout.

أهمية Sections في جعل التصميم مرنا

أحيانا نريد أن نضيف بعض المحتوى الخاص ب View معينة أو مجموعة من Views فنجد أن Layout تلزمنا بمحتوى مشترك بين جميع الصفحات، لذلك سنحتاج إلى استعمال

RenderSerction التي تمكننا من إضافة Sections حسب حاجتنا لها، حيث يمكن لل Views أن تعرف هذه Sections أو تتجاهلها، ويمكن لل Section أن تكون اختيارية أو إلزامية.
لنفهم ذلك أكثر، سنفترض أن بعض صفحات موقعنا قد تحتاج إلى ملفات css إضافية، هذه الملفات لسنا في حاجة لها في باقي الصفحات، وبالتالي وضعها على مستوى Layout سيؤدي إلى تحميلها حتى في الصفحات التي لا تستعملها وبالتالي التأثير سلبا على أداء التطبيق، لذلك سنعرف Section خاصة بالملفات التي لا ينبغي تحميلها على مستوى جميع الصفحات التي تطبق Layout، لعمل ذلك يمكننا كتابة الأوامر التالية:

_Layout.cshtml:

```
@RenderSection("OptionalCSS",required:false)
```

لاحظ أننا أعطينا للبرامتر required القيمة false فيما يدل على أن هذه Section اختيارية، الآن يمكننا الذهاب إلى الصفحة التي نريد تعريف ملفات css الاختيارية عليها، ونستدعي Section التي أسميناها OptionalCSS كما يلي:

viewName.cshtml:

```
@section OptionalCSS
{
    <link href="~/css/theme.css" rel="stylesheet" />
}
```

الملف أعلاه سيكون خاصا بالصفحة التي قامت باستدعاء OptionalCSS section وبالتالي ليس لزاما على Layout تحميله في كافة صفحات التطبيق.
الآن سنختم بمثال آخر لنفهم آلية RenderSection بشكل جيد، لنفترض أن صفحة معينة أو بعض الصفحات تحتاج إلى عرض قائمة جانبية Sidebar، في حين باقي الصفحات ليست في حاجة لذلك، إذن لا يمكننا وضع هذه القائمة على مستوى Layout لأن محتواه سيكون مشتركا وبالتالي سنلزم جميع الصفحات بعرض هذه القائمة الجانبية، في حين نحن نريد عرضها فقط على مستوى بعض الصفحات.
تعالوا بنا ندخل إلى Layout ونقوم بتعريف Section خاصة بهذا الأمر ولنسمها مثلا Sidebar كما يلي:

_Layout.cshtml:

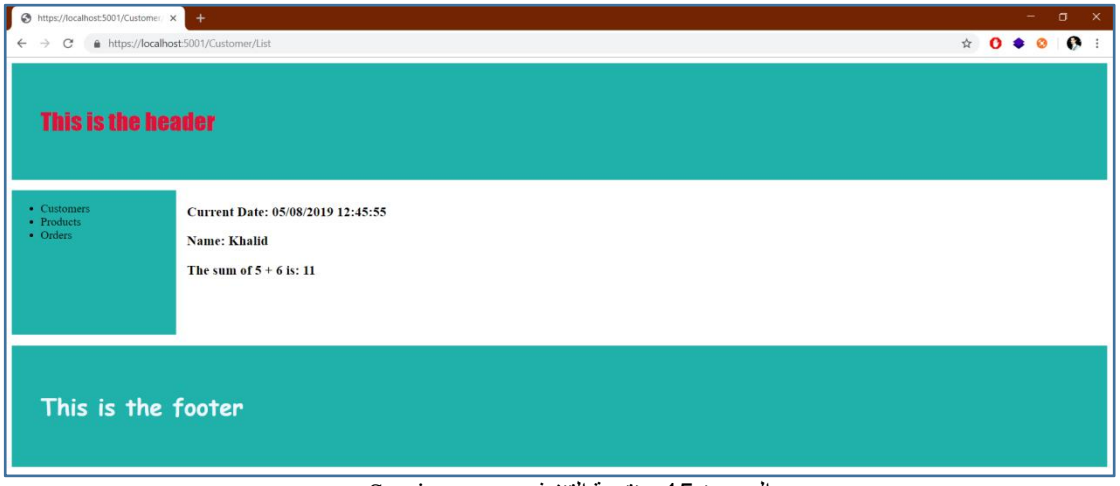
```
<div class="sidebar">
    @RenderSection("Sidebar", required: false)
</div>
```

قمت بإضافة بعض خصائص css لأظهر القائمة جنبا إلى جنب مع المحتوى المتغير الذي تمثله الوظيفة RenderBody، وسأقوم بعرض الأمثلة كاملة في نهاية هذا الفصل، الآن تعالوا بنا ندخل إلى الصفحات التي تحتاج إلى عرض القائمة الجانبية، ولنضف ما يلي:

viewName.cshtml:

```
@section Sidebar
{
    <ul>
        <li>Customers</li>
        <li>Products</li>
        <li>Orders</li>
    </ul>
}
```

لو نفذنا، ستلاحظ ظهور التصميم كما يلي:



الصورة 15 - نتيجة التنفيذ وعرض Section

في باقي الصفحات التي لا تعرف Sidebar Section لن تظهر القائمة الجانبية.

فيما يلي الأوامر الكاملة للصفحة _Layout.cshtml:

_Layout.cshtml:

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>

    <link href="~/css/site.css" rel="stylesheet" />
    @RenderSection("OptionalCSS", required: false)
</head>
<body>
    <div class="header">
        <h1>This is the header</h1>
```

```
    </div>
    <div>
        @RenderSection("Sidebar", required: false)
    </div>
    <div class="content">
        @RenderBody()
    </div>
    <div class="footer">
        <h1>This is the footer</h1>
    </div>
</body>
</html>
```

وهنا محتوى ملف site.css:

Site.css:

```
.header, .footer {
    background-color: #20b2aa;
    padding: 40px;
}

.header h1 {
    font-family: fantasy;
    color: crimson;
}

.footer h1 {
    font-family: cursive;
    color: aliceblue;
}

.sidebar{
    width: 15%;
    height: 200px;
    background-color: lightseagreen;
    float: left;
    margin-top: 15px;
}
```

وهنا محتوى الصفحة viewName.cshtml التي تستدعي Sidebar Section:

viewName.cshtml:

```
@{
    Layout = "_Layout";
}

@section Sidebar
{
    <ul  class="sidebar">
        <li>Customers</li>
        <li>Products</li>
        <li>Orders</li>
    </ul>
}
<h3>Current Date: @DateTime.Now</h3>
```

```
<h3>Name: Khalid</h3>
<h3>The sum of 5 + 6 is: @(5 + 6)</h3>
```

تحديد Layout مشترك للصفحات من خلال _ViewStart

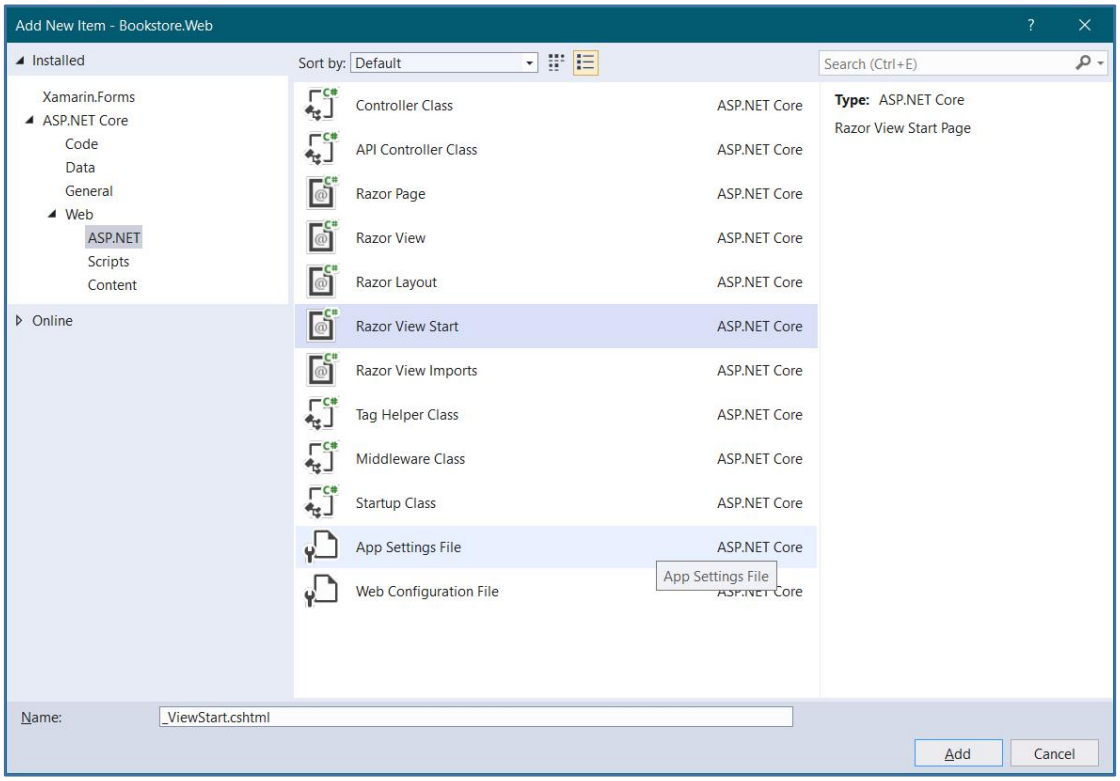
رأينا فيما تقدم أنه لكي نطبق Layout على صفحة معينة لابد من وضع الأمر التالي:

viewName.cshtml:

```
@{
    Layout = "_Layout";
}
```

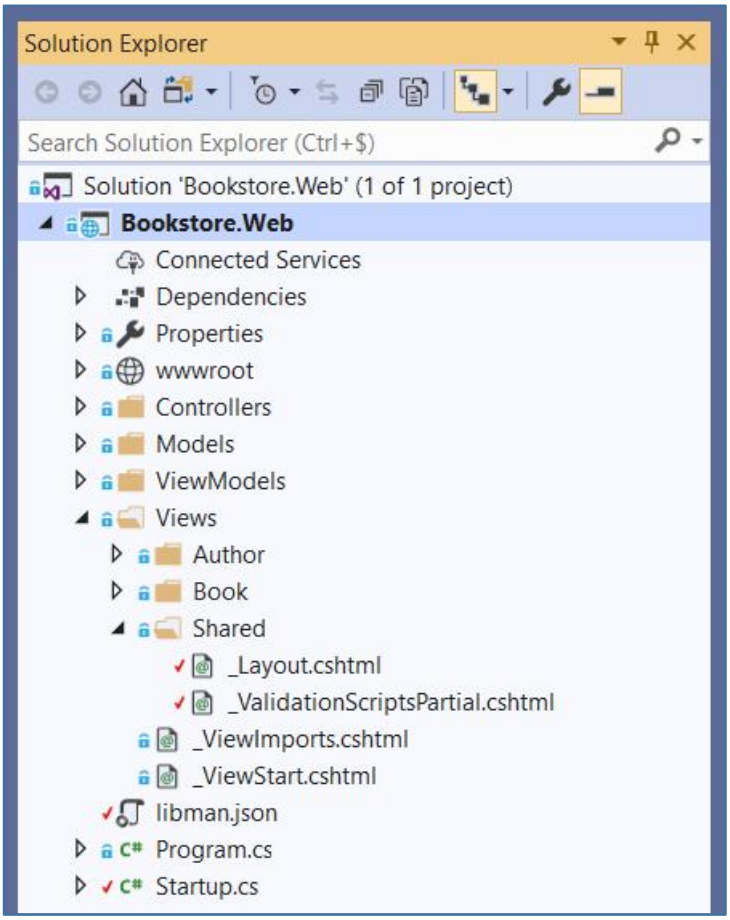
لكن تخيل أن عندنا في المشروع العديد من الصفحات، هل لابد من وضع هذا الأمر في كل صفحة على حدة؟

الجواب لا، يمكننا أن نضيف ملفا يسمى _ViewStart.cshtml ونضع فيه الأوامر التي نريد تنفيذها أولا قبل الدخول إلى views وبالتالي يمكننا كتابة الأوامر المشتركة بين الصفحات داخله، بما فيه الأمر الذي يستدعي Layout، لعمل ذلك تعالوا بنا نضيف هذا الملف مباشرة داخل مجلد Views ليتم تطبيقه على جميع الصفحات الموجودة داخل هذا المجلد وداخل المجلدات الفرعية داخله التي تحتوي على صفحات كما يلي:



الصورة 16 - إضافة ViewStart

لتكون بعد ذلك بنية المجلد كما يلي:



الصورة 17 - بنية مجلد Views

لو دخلنا إلى الملف _ViewStart.cshtml سنجده مسبقا يحتوي على الكود الخاص باستدعاء Layout الافتراضي كما يلي:

_ViewStart.chtml:

```
@{
    Layout = "_Layout";
}
```

الآن أي صفحة سيتم تطبيق Layout عليها دون أن نحتاج إلى تكرار كتابة الأمر أعلاه داخلها، لأنه كما قلنا عند التنفيذ سيتم تنفيذ ViewStart أولا ثم بعد ذلك باقي الصفحات.

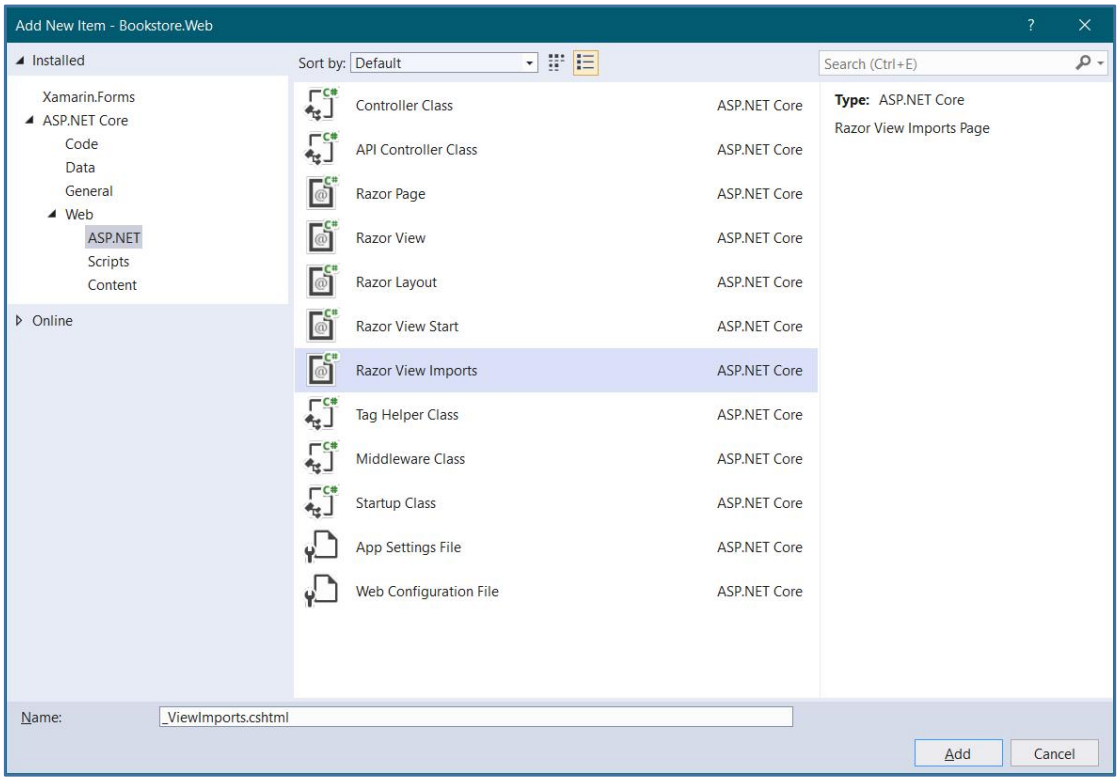
يمكننا أن نضيف أكثر من ViewStart داخل المشروع، بحيث كل ViewStart مثلا تؤشر إلى Layout يخص مجلدا معينا.

منع تكرار الأوامر واختصارها مع _ViewImports

يلعب ملف ViewImports دورا قريبا من ملف ViewStart، حيث بدوره يسمح لنا بمشاركة بعض الأوامر البرمجية التي تتكرر بين الصفحات، ووضعها في مكان واحد داخل ملف

ViewImports، على سبيل المثال قد تكون عدة صفحات تعمل using لنفس مجال الأسماء، بدل أن نكتب داخل كل صفحة نفس using نقوم بوضعها في هذا الملف وتلقائيا سيتم تضمينها في الصفحات التي تحتاج إلى عناصر من هذا namespace.

سنتعرف أكثر على هذا الملف من خلال التطبيق، لنقم بإضافته أولا، علما أنه من الممكن أن يكون موجودا مسبقا في مشروعك، إن لم تجده فقم بإضافته مباشرة داخل مجلد Views كما يلي:



الصورة 18 - إضافة ملف ViewImports

لنضف إليه مجالات الأسماء التي سنحتاجها، وليكن مثلا:

_ViewImports.chtml:

```
@using Bookstore.Web.Models
```

الآن في الصفحات التي تحتاج إلى هذا namespace لسنا بملزمين بكتابته لأنه مسجل مسبقا في ViewImports، بمعنى لو عندنا صفحة ستحتاج إلى عنصر من عناصر هذا namespace فسوف نقوم باستدعائه مباشرة دون الحاجة إلى كتابة مجال الأسماء، كما يبين المثال التالي:

viewName.chtml:

```
@model IList<Customer>;
@foreach (var customer in Model)
```

```
{
    <div>
        <h3>First Name: <span>@customer.FirstName</span></h3>
        <h3>Last Name: <span>@customer.LastName</span></h3>
        <h3>Phone: <span>@customer.Phone</span></h3>
        <h3>Age: <span>@customer.Age</span></h3>
    </div>
}
```

لاحظ أننا استعملنا النوع Customer دون ذكر namespace لأنه ينتمي إلى namespace المذكور في ملف ViewImports.

المثال أعلاه فقط للتوضيح، لست مطالبا بفهم الكود لأننا لم نتطرق بعد إلى بنية MVC وكيفية نقل البيانات من Actions إلى Views وهو ما سنشرع فيه ابتداء من الفصل المقبل إن شاء الله.

يمكن أن يوجد في المشروع أكثر من ملف ViewImports حسب الحاجة، إذ يمكننا وضع الملف في مجلد معين يحتاج إلى بعض الأمور المشتركة بين صفحاته، أو الاكتفاء بملف واحد نضعه مباشرة في مجلد Views ليتم تطبيق محتواه على جميع الصفحات.

يمكننا داخل ملف ViewImports أن نضيف أيضا Tag helpers بغرض استعمالها في صفحاتنا، فماهي هذه Tag helpers؟

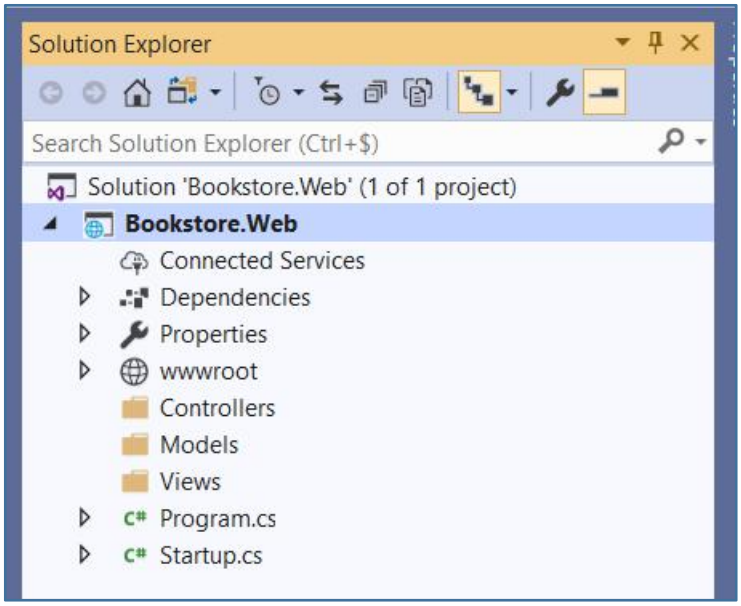
هذا ما سنتعرف عليه في أحد الفصول المقبلة إن شاء الله.

ثم انقضاض مباغت على MVC

بنية MVC أو Model View Controller تسمح لنا ببناء مشاريع ويب قوية قابلة للتحديث والاختبار بسبب اعتمادها على مبدأ فصل المهام، حيث يتم تفويض كل وحدة لتقوم بعمل معين، فمثلا Model خاص بتعريف البيانات، و View خاص بعرض واستقبال البيانات، بينما يلعب Controller دور الوسيط الذي يستقبل HTTP requests ثم يقوم بمعالجتها وتنفيذها من خلال قراءة البيانات من Model أو إرسالها إليه، ثم في الختام اختيار View المناسب لعرض الناتج، لفهم هذه البنية جيدا، تعالوا بنا ندخل إلى مشروعنا الذي قمنا بإنشائه ولنقم بإنشاء المجلدات الثلاثة التالية:

- **Models:** سنضيف إليه كافة Models التي قد يحتاجها مشروعنا
- **Controllers:** سنخزن فيها كافة Controllers
- **Views:** سنضيف إليه الصفحات اللازمة

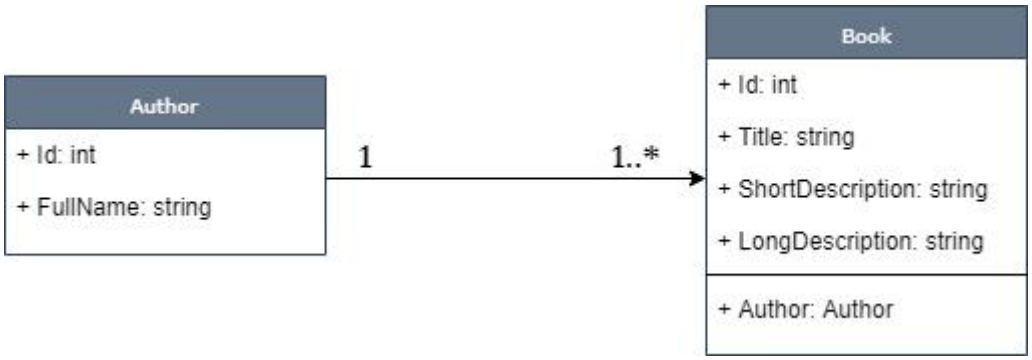
وهذا شكل المشروع بعد إضافة المجلدات الثلاثة:



الصورة 19 - إضافة المجلدات اللازمة

المؤلفون والكتب، مسرحنا الجديد

المشروع الذي سنشتغل عليه يسمح لنا بإدارة الكتب والمؤلفين، بحيث كل مؤلف يتوفر على كتاب أو أكثر، بينما الكتاب الواحد يتم تأليفه من كاتب واحد كما يبين Class Diagram التالي:



الصورة 20 - نموذج كلاسات المشروع

بناء الأساس مع Models

بعد ذلك سندخل إلى مجلد Models، وسنضيف إليه كلاس جديد ولنسمه مثلا Author.cs، ثم نضيف إليه Properties التالية:

Author.cs:

```
public class Author
{
    public int Id { get; set; }
    public string FullName { get; set; }
}
```

بنفس الكيفية سنضيف الكلاس الخاص بالكتب، دائما داخل المجلد Models سنضيف كلاس نسميه مثلا Book.cs، وهذا هو محتواه:

Book.cs:

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string ShortDescription { get; set; }
    public string LongDescription { get; set; }

    public Author Author { get; set; }
}
```

تعرف عن قرب على Repository Design Pattern

لكي نتمكن من توزيع المهام في مشروعنا سننشئ كلاسات تحتوي على الوظائف الخاصة بقراءة وكتابة البيانات عبر استعمال Repository Design Pattern، علما أن استعمال هذا النموذج يحتم علينا أيضا استعمال نموذج Unit Of Work بغرض حفظ البيانات لكن لن نتطرق إلى ذلك حتى لا نعقد الأمور وندخل في تفاصيل أخرى ربما يحسن تفصيلها في كتاب آخر، لذلك سنضع كل الوظائف التي نحتاجها في Repository بصرف النظر عن خرقنا لمبادئ التصميم البرمجي.

لعمل ذلك سنأتي داخل مجلد Models وننشئ مجلدا جديدا نسميه مثلا Repositories، داخله سننشئ هذه المرة Interface ولنسمها مثلا IRepository وليكن محتواها كما يلي:

IRepository.cs:

```
public interface IRepository<T>
{
    IList<T> GetAll();
    T GetById(int id);
    void Add(T item);
    void Edit(int id, T item);
    void Delete(int id);
}
```

الواجهة كما ترى تحتوي على الوظائف التي تقوم بقراءة، إضافة، تعديل وحذف البيانات، وحتى نستعمل هذه الواجهة مع كافة Models في مشروعنا، جعلنا النوع قابل للتغيير بحيث يمكن استعمالها من طرف أكثر من نوع وذلك باستعمال مفهوم Generics، الآن سننشئ كلاس خاصة ب Repository لكل Models في مشروعنا، وسنقوم بتطبيق الواجهة IRepository على هذه Repository لنعيد تعريف الوظائف السابقة.

داخل المجلد Repositories سننشئ كلاس نسميه مثلا AuthorRepository.cs كما يلي:

AuthorRepository.cs:

```
public class AuthorRepository : IRepository<Author>
{
    public void Add(Author item)
    {
        throw new NotImplementedException();
    }

    public void Delete(int id)
    {
        throw new NotImplementedException();
    }

    public void Edit(int id, Author item)
    {
        throw new NotImplementedException();
    }

    public IList<Author> GetAll()
    {
        throw new NotImplementedException();
    }

    public Author GetById(int id)
    {
        throw new NotImplementedException();
    }
}
```

طبعا الوظائف لا تحتوي إلى حدود الساعة على أية Implementation لأننا قمنا فقط بتطبيق Interface وبالتالي ورثنا وظائفها، الآن سنقوم بإنشاء List لنخزن فيها بيانات المؤلفين، وطبعا لأننا لا نتعامل مع قاعدة البيانات حاليا وإنما مع Objects مخزنة في الذاكرة فيما يعرف ب In Memory Objects، سنستعمل قائمة من نوع List كما يلي:

AuthorRepository.cs:

```
public IList<Author> Authors { get; set; }
```

من أجل إعطاء قيم بدئية لهاته List سنقوم بإنشاء Constructor في الكلاس وننشئ فيه عدة Objects من نوع Author كما يلي:

AuthorRepository.cs:

```
public AuthorRepository()
{
    Authors = new List<Author>() {
        new Author
        {
            Id=1, FullName="Khalid ESSAADANI"
        },
        new Author
        {
            Id=2, FullName="Hamid MAKBOUL"
        },
        new Author
        {
            Id=3, FullName="Said HAMRI"
        },
    };
}
```

بالنسبة للوظيفة Add فهي تقوم بإضافة Object القادم في البرامتر إلى List التي أسميناها Authors، وبالتالي سيكون محتواها كما يلي:

AuthorRepository.cs:

```
public void Add(Author author)
{
    Authors.Add(author);
}
```

أما وظيفة الحذف Delete فهي بكل بساطة تستقبل برامتر من نوع int للبحث عن المؤلف بواسطة معرفه، وعند العثور عليه تقوم بحذفه من قائمة Authors كما يلي:

AuthorRepository.cs:

```
public void Delete(int id)
{
    var author = Authors.SingleOrDefault(a => a.Id == id);

    Authors.Remove(author);
}
```

أما وظيفة التعديل Edit فهي قريبة من الوظيفة Delete حيث بدورها تبحث عن المؤلف المراد تعديله، ثم تقوم بعد ذلك بإسناد البيانات الجديدة إليه كما يلي:

AuthorRepository.cs:

```
        public void Edit(int id, Author newAuthor)
        {
            var oldAuthor = Authors.SingleOrDefault(a => a.Id == id);
            oldAuthor.FullName = newAuthor.FullName;
        }
```

أما بالنسبة للوظيفة GetAll فهي تقوم بكل بساطة بإرجاع جميع المؤلفين، والكود الخاص بها كما يلي:

AuthorRepository.cs:

```
        public IList<Author> GetAll()
        {
            return Authors;
        }
```

بقيت فقط الوظيفة GetById التي تعيد لنا المؤلف من خلال المعرف الخاص به، وسيكون كودها كما يلي:

AuthorRepository.cs:

```
        public Author GetById(int id)
        {
            return Authors.Single(a=>a.Id==id);
        }
```

يمكننا تنقيح محتوى AuthorRepository، حيث في وظائف الحذف والتعديل بدل استعمال نفس الكود الخاص بالبحث عن المؤلف، يمكننا استدعاء الوظيفة GetById التي تقوم بنفس العمل، وبالتالي يكون الكود الكامل الخاص ب AuthorRepository على الشكل التالي:

AuthorRepository.cs:

```
    public class AuthorRepository : IRepository<Author>
    {
        public IList<Author> Authors { get; set; }
        public AuthorRepository()
        {
            Authors = new List<Author>() {
                new Author
                {
                    Id=1, FullName="Khalid ESSAADANI"
                },
```

```
                new Author
                {
                    Id=2, FullName="Hamid MAKBOUL"
                },
                new Author
                {
                    Id=3, FullName="Said HAMRI"
                },
            };
        }

        public void Add(Author author)
        {
            Authors.Add(author);
        }

        public void Delete(int id)
        {
            var author = GetById(id);

            Authors.Remove(author);
        }

        public void Edit(int id, Author newAuthor)
        {
            var oldAuthor = GetById(id);
            oldAuthor.FullName = newAuthor.FullName;
        }

        public IList<Author> GetAll()
        {
            return Authors;
        }

        public Author GetById(int id)
        {
            return Authors.Single(a=>a.Id==id);
        }
    }
```

الآن بعد أن انتهينا من الكلاس AuthorRepository التي تقوم بإدارة المؤلفين وتوفير كافة الوظائف التي قد نحتاجها في مشروعنا، يمكننا أن نقوم بإنشاء Repository الخاص بالكتب، لعمل ذلك سنضيف كلاس جديد داخل مجلد Repositories ولنسمه مثلا BookRepository.cs.

الآن نحتاج إلى التركيز قليلا، لأن الكلاس Book يحتوي على خاصية من نوع Author، وبالتالي وجب إضافة مؤلف لكل كتاب.

حتى نفهم محتوى الكلاس BookRepository سنقوم بشرح كل وظيفة على حدة، في الأول تعالوا بنا ننشئ List لتسجيل الكتب ونقوم بتعبئتها في Constructor كما يلي:

BookRepository.cs:

```
        private IList<Book> books;

        public BookRepository()
        {
            books = new List<Book>()
            {
                new Book
                {
                    Id=1,
                    Title="C# Programming",
                    LongDescription="Long description...",
                    ShortDescription="Short description...",
                    Author=new Author{Id=1, FullName="Khalid ESSAADANI"}
                },
                new Book
                {
                    Id=2,
                    Title="Java Programming",
                    LongDescription="Long description...",
                    ShortDescription="Short description...",
                    Author=new Author{Id=1, FullName="Khalid ESSAADANI"}
                },
                new Book
                {
                    Id=3,
                    Title="Python Programming",
                    LongDescription="Long description...",
                    ShortDescription="Short description...",
                    Author=new Author{Id=1, FullName="Khalid ESSAADANI"}
                },
            };
        }
```

الوظيفة الخاصة بإضافة الكتب ستقوم بزيادة المعرف id بواحد من خلال جلب أكبر قيمة والزيادة عليها، ثم نضيف الكتاب إلى قائمة الكتب كما يلي:

BookRepository.cs:

```
        public void Add(Book book)
        {
            book.Id = books.Max(b => b.Id) + 1;

            books.Add(book);
        }
```

باقي الوظائف هي مثل الوظائف التي قمنا بتعريفها على مستوى الكلاس AuthorRepository، وبالتالي دعونا نعرض الكود الكامل الخاص بالكلاس BookRepository وهو كما يلي:

BookRepository.cs:

```
public class BookRepository : IRepository<Book>
{
    private IList<Book> books;

    public BookRepository()
    {
        books = new List<Book>()
        {
            new Book
            {
                Id=1,
                Title="C# Programming",
                LongDescription="Long description...",
                ShortDescription="Short description...",
                Author=new Author{Id=1, FullName="Khalid ESSAADANI"}
            },
            new Book
            {
                Id=2,
                Title="Java Programming",
                LongDescription="Long description...",
                ShortDescription="Short description...",
                Author=new Author{Id=1, FullName="Khalid ESSAADANI"}
            },
            new Book
            {
                Id=3,
                Title="Python Programming",
                LongDescription="Long description...",
                ShortDescription="Short description...",
                Author=new Author{Id=1, FullName="Khalid ESSAADANI"}
            },
        };
    }

    public void Add(Book book)
    {
        book.Id = books.Max(b => b.Id) + 1;

        books.Add(book);
    }

    public void Delete(int id)
    {
        var book = GetById(id);
        books.Remove(book);
    }

    public void Edit(int id, Book editedBook)
    {
        var book = GetById(id);

        book.Title = editedBook.Title;
        book.LongDescription = editedBook.LongDescription;
        book.ShortDescription = editedBook.ShortDescription;
        book.Author = editedBook.Author;
    }
```

```
        public IList<Book> GetAll()
        {
            return books;
        }

        public Book GetById(int id)
        {
            var book = books.Where(b => b.Id == id).SingleOrDefault();
            return book;
        }
    }
```

بهذه الكيفية نكون قد أسسنا بشكل جيد لمشروعنا، حيث لو أردنا أن نقوم بربطه بقاعدة بيانات من خلال استعمال فريموورك معين مثل Entity Framework Core سنقوم فقط باستعمال الوظائف التي عرفناها على مستوى IRepository Interface ثم نكتب Implementation وهو ما يعد تطبيقا جيدا لمفهوم Dependency Injection الذي سنتعرف عليه في الفصل اللاحق قبل أن نكمل مشروعنا من خلال إنشاء Controllers و Views اللازمة.

في سبيل القضاء على وسواس Dependency Injection

إذا أردنا أن نقوم بتطوير مشاريع قابلة للصيانة Maintainable وقابلة للاختبار Testable، فإننا من غير شك نقوم باستعمال الكلاسات في مشاريعنا، ونخصص لكل كلاس مسؤولية واحدة فقط لكي لا نقع في مشاكل تضر بجودة تصميم البرمجيات Software Design، وهذا التخصيص للمسؤوليات هو أول مبدأ من مبادئ التصميم الخمسة المعروفة ب SOLID والتي سنتطرق لها لاحقا في كتاب مستقل إن شاء الله، أول مبدأ من هذه المبادئ هو Single Responsibility والذي يلزمنا بأن نوكل للكلاس الواحد مسؤولية واحدة فقط وألا نكلفه بأكثر من مهمة، فلا يمكن للكلاس الواحد مثلا أن يقوم بمعالجة البيانات، وحفظها، والتحقق من صحتها، وتسجيل العمليات التي تحدث فيما يعرف ب Logging، وإنما ينبغي أن نخصص لكل مسؤولية من هذه المسؤوليات كلاس خاص بها، وذلك من أجل رفع جودة التصميم ومن أجل كتابة شفرة نقية Clean Code، لكن ومع ذلك فطبيعة المشاريع تحتم علينا أن نربط الكلاسات مع بعضها البعض، فمثلا ممكن للكلاس Order أن تحتاج إلى نسخة instance من الكلاس Customer، وغير ذلك من الحالات التي نجد أننا ملزمين فيها بإنشاء ارتباطات بين الفئات، هذه الارتباطات تعرف برمجيا باسم Dependencies، كلما كبر المشروع كلما زادت هذه Dependencies، والسيء في العملية أنه كلما كثرت Dependencies في المشروع كلما كان صعبا التعديل عليه، أو تحديثه، أو صيانته، لأن التعديل على كلاس معين سيؤدي حتما إلى التعديل

على الكلاسات المرتبطة به، وبالتالي قد تحدث بعض المشاكل في الكود وبالتالي بعض العمليات التي كانت تشتغل قد تصبح غير قادرة على الاشتغال، وذلك بسبب الارتباطات.
لذلك ينبغي أن نقلل الارتباطات أقصى ما نستطيع في مشاريعنا من أجل جعلها قابلة للتعديل دون أن يؤثر ذلك على باقي أجزاء المشروع، ومن بين الحلول التي تسهل عملية إضعاف الارتباطات نجد التطبيق الفعلي للمبدأ الخامس من مبادئ SOLID وهو مبدأ Dependency Inversion Principle، والذي يقضي باختصار بأن الكلاسات لا ينبغي أن ترتبط بالتفاصيل Details وإنما ينبغي أن ترتبط بالتجريد Abstraction، فيما معناه بدل أن تتعامل الكلاس مع ارتباط معين من نوع كلاس مما يحتم عليها ارتباطها بتفاصيله البرمجية، فإنه من الأحسن أن ترتبط مع عنصر مجرد مثل Interface أو Abstract Class، وفي ذلك يقول روبرت مارتن الشهير بالعم بوب:

> Robert Martin AKA Uncle BOB:
>
> High-level modules should not depend on low-level modules. Both should depend on the abstraction.
> Abstractions should not depend on details. Details should depend on abstractions.

يمكننا ترجمة العبارات أعلاه إلى ما يلي:

1. الوحدات العليا لا ينبغي أن ترتبط بالوحدات الدنيا، كلتاهما ينبغي أن ترتبطا بالتجريد
2. التجريد لا ينبغي أن يرتبط بالتفاصيل، التفاصيل من ينبغي لها أن ترتبط بالتجريد

الوحدات العليا يقصد بها الكلاسات التي ترتبط بكلاسات أخرى، والوحدات الدنيا هي الكلاسات التي تستعمل كارتباطات في الكلاسات العليا، على سبيل المثال: الكلاس Order يحتاج إلى ارتباط من نوع Customer، إذن فالكلاس Order عبارة عن وحدة عليا، والكلاس Customer عبارة عن وحدة دنيا.
أما العبارة الثانية فالمقصود بها، أن كل من الكلاسات العليا والدنيا ينبغي أن ترتبط بعناصر مجردة مثل Interface أو Abstract Class بدل الارتباط بتفاصيل الكلاسات.
وحتى نفهم هذا المبدأ على أحسن وجه ينبغي أن نتطرق في الأول إلى شرح Inversion of Control Principle، لكن لأن المقام غير مناسب، فسنكتفي بهذا القدر.

إجمالا يمكننا القول أن الغرض الأساسي من تطبيق مبدأ Dependency Inversion هو إضعاف الارتباط بين الكلاسات، وذلك بغرض بناء أنظمة قابلة للصيانة والاختبار، وإحدى أدوات تطبيق مبدأ Dependency Inversion هو نموذج Dependency Injection الذي يقضي بأن عملية إنشاء نسخ من الكلاسات المرتبط بها ينبغي أن يتم خارج الكلاس الذي يحتاجها، ويمكن ذلك من خلال عدة حلول.

أحد الحلول الجاهزة التي تعفينا من عناء بناء IoC container خاص بنا أو استعمال Third-Party IoC هو آلية Dependency Injection الموجودة مسبقا في ASP.NET Core والتي تسمح لنا بتسجيل الكلاسات التي سنحتاجها في مشروعنا، والتي تسمى بلغة Dependency Injection بالخدمات services، تسمح لنا آلية Dependency Injection الموجودة في ASP.NET Core بتسجيل هذه services في IoC Container مهيأ مسبقا، وحينما نقول IoC Container فهي اختصار ل Inversion of Control Container ونسميه كذلك DI Container اختصارا ل Dependency Injection Container وهو المسؤول عن تفعيل آلية Dependency Injection، بحيث كلما طلبنا نسخة منها قام هذا الفريموورك بإنشاء نسخة لنا لنستعملها، دون أن نحمل هم كيفية إدارة هذه الارتباطات.

أعلم أن الكلام قد يبدو نظريا بشكل كبير، لكن بالمثال سيتضح المقال.

لذلك دعونا في الفقرة المقبلة نتعرف على كيفية إدارة الارتباطات في ASP.NET Core عبر تسجيلها في Built-in IoC container.

فكونوا على أتم الاستعداد.

بالمثال يتضح المقال يا Dependency Injection

مبدئيا، دعونا ندخل إلى الكلاس Startup.cs، وتحديدا إلى الوظيفة ConfigureServices والتي تسمح لنا بتسجيل services في DI Container الذي توفره لنا تقنية ASP.NET Core:

Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{

}
```

يمكنك أن تلاحظ أن هذه الوظيفة تستقبل برامتر اسمه services من نوع IServiceCollection وهي عبارة عن قائمة يستعملها IoC Container من أجل تسجيل الارتباطات التي يحتاجها المشروع، وهي التي سنستعملها لتخزين الارتباطات التي نريد استعمالها في مشروعنا.

يوجد نوعان من Dependencies التي يمكننا إدارتهما بواسطة IoC Container وهما كما يلي:

- **الارتباطات الخاصة بالفريموورك:** وهي ارتباطات تشكل جزء من ASP.NET Core والتي يمكننا استعمالها حسب حاجتنا لها مثل تسجيل DbContext في حال استعمال Entity Framework Core.
- **الارتباطات الخاصة بالمشروع:** وهي الكلاسات التي ننشئها في مشروعنا، ونحتاج إلى تسجيلها في IoC Container.

سنتعرف الآن على كيفية إنشاء الكلاسات وربطها ب Interface ثم تسجيلها في IoC Container ومن ثم استعمالها بعد ذلك في مكان ما من المشروع، وستلاحظون كيف يقوم هذا container بتسجيل الارتباطات وتوفيرها عند الحاجة دون أن نتدخل نحن في عملية إنشاء Objects.

لنفهم العملية بشكل مبسط سنشتغل على مثال بسيط جدا، تعالوا بنا في الأول ننشئ Interface كما يلي:

ICustomer.cs:

```
public interface ICustomer
{
    string GetName();
}
```

ثم بعد ذلك سننشئ كلاس يطبق هذه Interface كما يلي:

Customer.cs:

```
public class Customer : ICustomer
{
    public string GetName()
    {
        return "Khalid ESSAADANI";
    }
}
```

الكلاس كما تلاحظون لا يقوم بأي عمل مهم، فقط يرجع الاسم في الوظيفة GetName، الآن لو افترضنا أننا نحتاج إلى نسخة من هذا الكلاس في مشروعنا وليكن في Controller مثلا، فإننا بالطريقة الكلاسيكية العادية سنستخدم الكلمة new كما يلي:

Customer.cs:

```
Customer customer = new Customer();
```

وقد بينا فيما مضى سلبيات هذا النوع من الاستنساخ المباشر الذي يجعل الكلاسات مرتبطة فيما بينها بشكل كبير، لذلك سنستعمل آلية Dependency Injection، وذلك من خلال الذهاب إلى الوظيفة ConfigureServices، ثم نقوم بتسجيل الكلاس بالإضافة إلى Interface التي يطبقها، وذلك في تطبيق صريح لمبدأ Dependency Inversion، تعالوا بنا ندخل إلى الملف Startup.cs وتحديدا إلى الوظيفة ConfigureServices ثم نسجل الكلاس كما يلي:

Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddScoped<ICustomer, Customer>();
}
```

لتسجيل الارتباطات في ASP.NET IoC Container يمكننا استعمال إحدى الوظائف التالية:

- **AddScoped:** والتي تعني أنه سيتم إنشاء object من الكلاس في كل HTTP request
- **AddTransient:** تعني أنه سيتم إنشاء object من الكلاس في كل مرة نطلب ذلك
- **AddSingleton:** وتعني أنه سيتم إنشاء Object واحد خلال تنفيذ التطبيق، وهذه العملية تعد تطبيقا فعليا لنموذج التصميم Singleton Design Pattern والذي يقضي بإنشاء نسخة واحدة من الكلاس في كل المشروع.

بعد أن قمنا بتسجيل الكلاس في IoC container، بقي فقط أن نستعمله في المكان الذي نود، وليكن مثلا Controller، لعمل ذلك سنستخدم أسلوب الحقن بواسطة Constructor كما يبين الكود التالي:

CustomerController.cs:

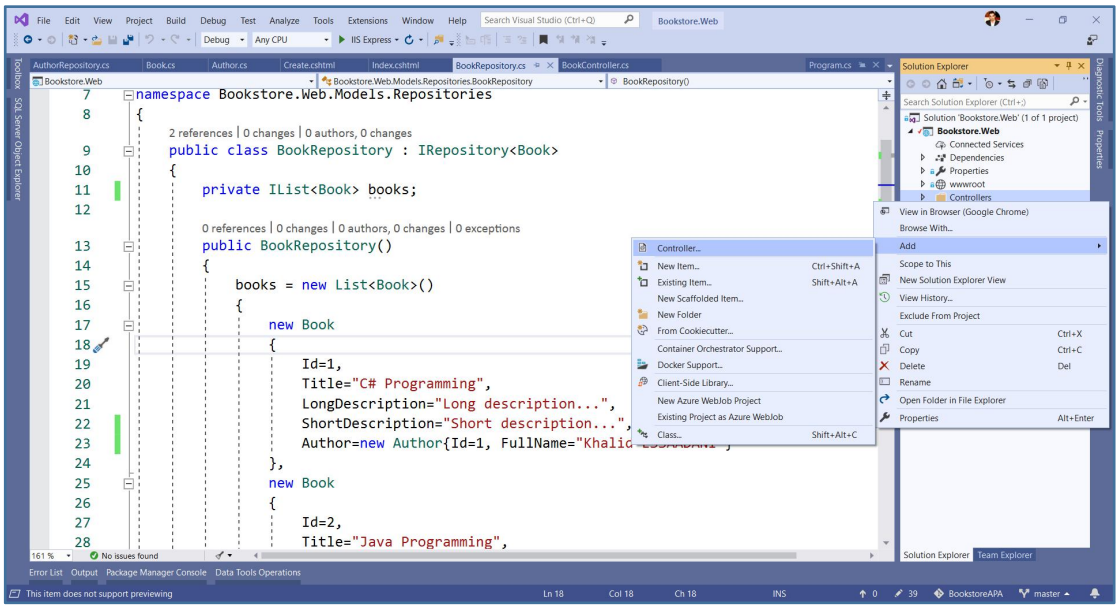
```
public class CustomerController : Controller
{
    public ICustomer Customer { get;  set; }

    public CustomerController(ICustomer customer)
    {
        Customer = customer;
    }


    public IActionResult Index()
    {
        var customerName = Customer.GetName();

        return View("Index",customerName);
    }
}
```

لو قمنا بتنفيذ المشروع، ستلاحظ أنه تم إنشاء نسخة من الكلاس Object وتم استدعاء الوظيفة GetName بنجاح من أجل إرجاع النص الذي نريد.

إلى هنا نكون قد تعرفنا على آلية Dependency Injection في ASP.NET Core وكيفية استعمالها.

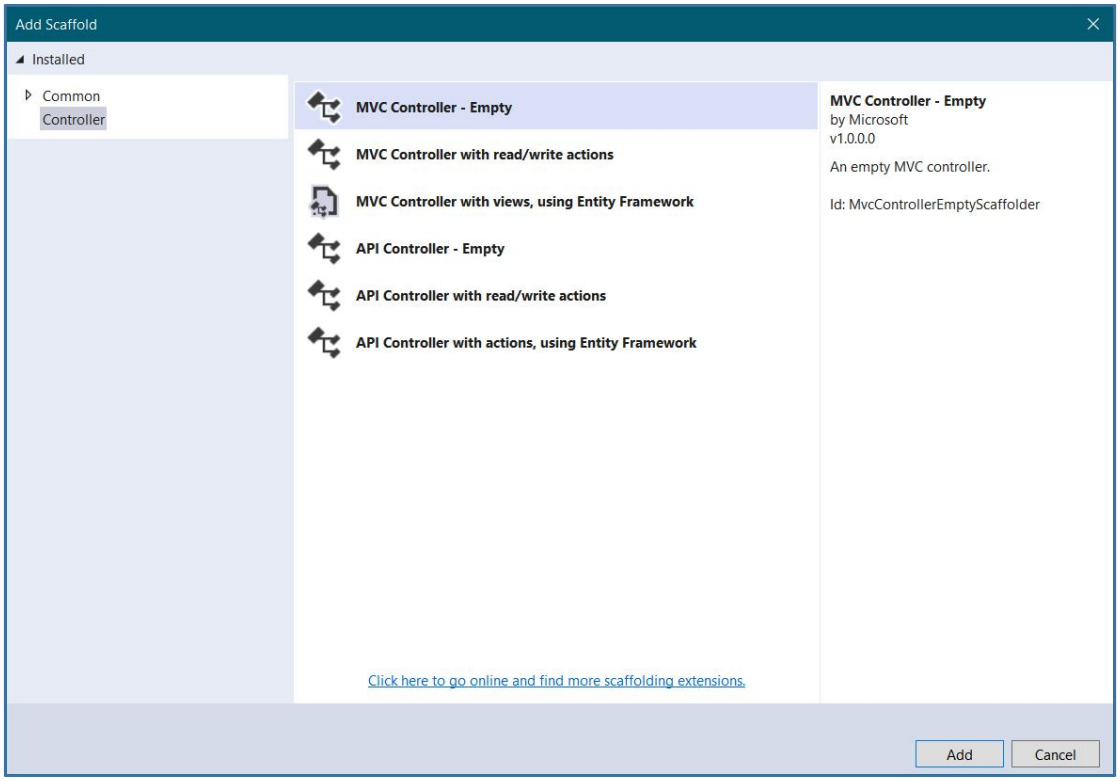
قلب التطبيق النابض Controllers

الآن سنقوم بإنشاء أول Controller لنا في المشروع، لنضغط بيمين الماوس على المجلد Controllers ثم نختار Add، ونحدد Controller كما يلي:



الصورة 21 - إضافة كونترولر جديد

ستظهر لنا بعد ذلك الواجهة التالية:



الصورة 22 - اختيار نوع الكونترولر

لنحدد الخيار الأول MVC Controller - Empty، ثم نضغط على الزر Add، لتظهر لنا الشاشة الصغيرة التالية:

الصورة 23 - تسمية الكونترولر

بعد ذلك نضغط على الزر Add لتتم إضافة Controller إلى المجلد، لو دخلنا إليه سنجد محتواه كما يلي:

AuthorController.cs:

```
public class AuthorController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

بكل بساطة، Controller مجرد كلاس ترث من الكلاس الرئيسي Controller، ويحتوي على وظائف لاستقبال HTTP Requests، وكما تلاحظ معي فهنالك Action افتراضي اسمه Index، هذا الأخير يمكننا الوصول إليه بواسطة HTTP Protocol في حالة التبويب الافتراضي لل ASP.NET Core عبر الرابط التالي:

```
Endpoint example:

http://localhost:portNumber/Author/Index
```

حيث portNumber هو رقم البورت الذي يشتغل عليه التطبيق، و Author هو اسم Controller التي نريد الوصول إليها، ولاحظ أننا أزلنا الجزء الأخير من اسم Controller واكتفينا بالاسم Author، ثم أضفنا إلى route اسم Action وهو Index (علما أنه حينما يكون اسم action هو index فلسنا ملزمين بكتابته لأنها action الافتراضية في كل Controller لكن في حالة باقي الأسماء لابد من وضعها في route).

الآن سنقوم بعمل Injection لل AuthorRepository بواسطة مشيد الكونترولر، ثم بعد ذلك سنقوم بإنشاء الكود الذي يجلب جميع Authors في Index Action.

لعمل Injection لل AuthorRepository service فإن الطريقة كما يلي:

AuthorController.cs:

```
public IRepository<Author> Repository { get; }

public AuthorController(IRepository<Author> repository)
{
    Repository = repository;
}
```

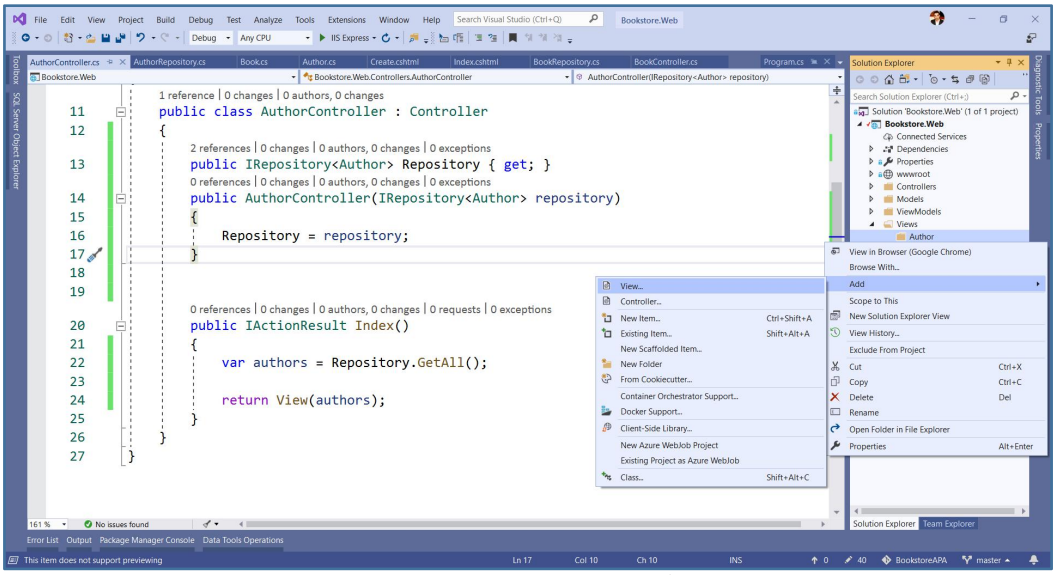
ثم نأتي إلى Index Action ونستدعي الوظيفة GetAll التي تجلب جميع المؤلفين، ونقوم بإرجاع النتيجة إلى View بغرض عرضها كما يلي:

AuthorController.cs:

```
public IActionResult Index()
{
    var authors = Repository.GetAll();

    return View(authors);
}
```

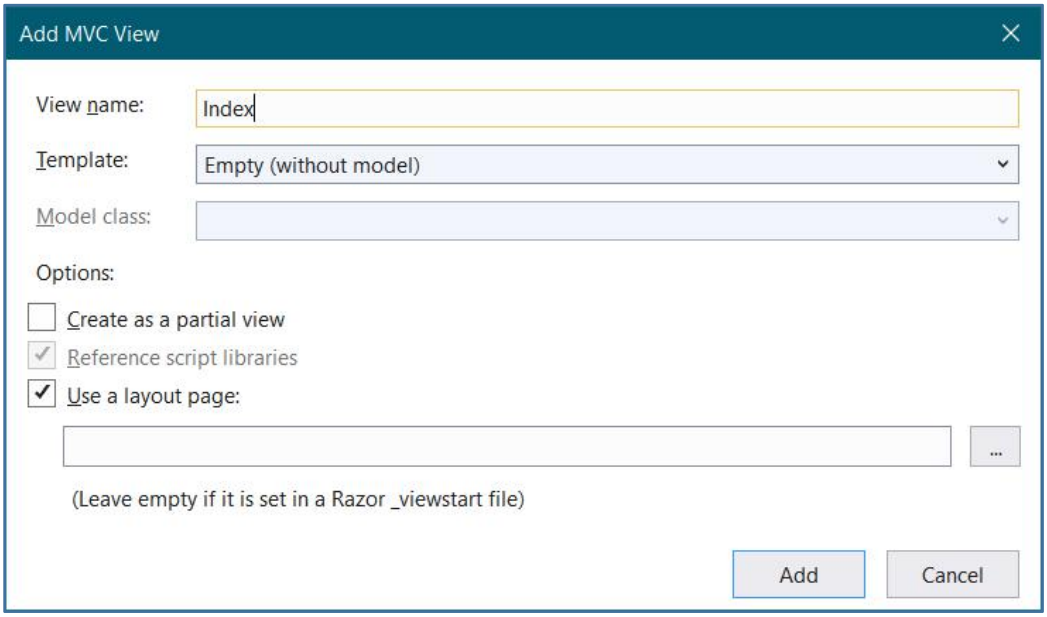
لسان الحال أو Views ودورها في بدء وإنهاء سلسلة التصنيع

الآن لنذهب إلى مجلد Views، ونقوم بإضافة مجلد جديد نسميه Author، ثم بداخله نضيف صفحة من نوع Razor view نعطيها نفس اسم Action وهو Index:



الصورة 24 - إضافة صفحة جديدة

ونترك الخيارات على ماهي عليه:



الصورة 25 - تسمية الصفحة

ثم نضع بداخلها الأوامر التالية التي تسمح لنا بعرض جميع المؤلفين على شكل جدول، مع عرض روابط لتعديل وحذف ومشاهدة تفاصيل كل مؤلف:

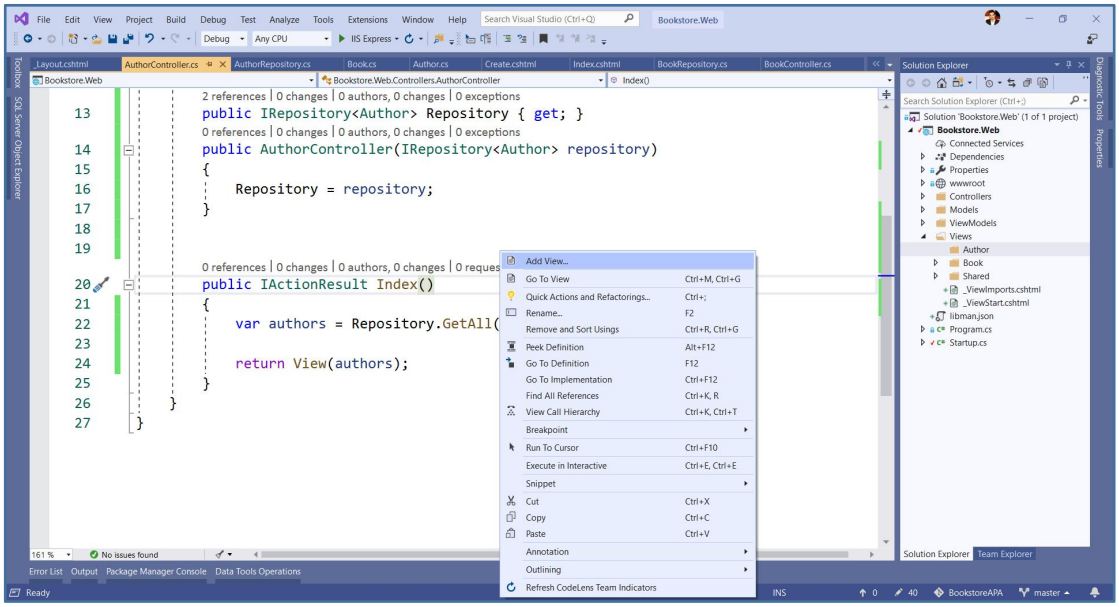
Author\Index.cshtml:

```
@model IEnumerable<Bookstore.Web.Models.Author>
```

```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Id)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.FullName)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Id)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.FullName)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey
*/ }) |
                @Html.ActionLink("Details", "Details", new { /*
id=item.PrimaryKey */ }) |
                @Html.ActionLink("Delete", "Delete", new { /*
id=item.PrimaryKey */ })
            </td>
        </tr>
}
    </tbody>
</table>
```

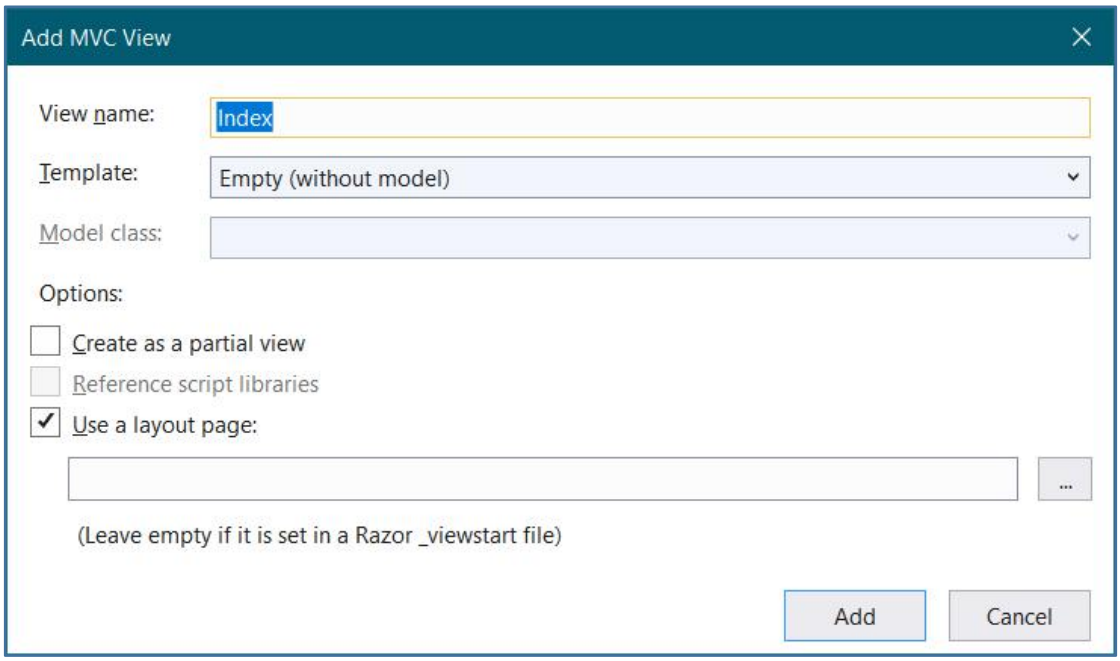
قد تبدو لك التعليمات البرمجية صعبة بعض الشيء لكنها مجرد عملية Loop من نوع foreach على البيانات المرجعة والتي هي على هيئة IEnumerable<Author>، لكنك قد تجد الأوامر كثيرة وتقول هل سأكتب هذه الأوامر في كل صفحة، أبشرك لست ملزما بذلك، لأن ASP.NET توفر لنا آلية تسمى Scaffolding تسمح لنا ببناء الصفحات تلقائيا، لذلك قم بحذف الصفحة Index التي أنشأتها وعد معي إلى الكونترولر AuthorController.

ضع مؤشر الماوس على اسم Index Action، واضغط بيمين الماوس ثم اختر Add View كما يلي:



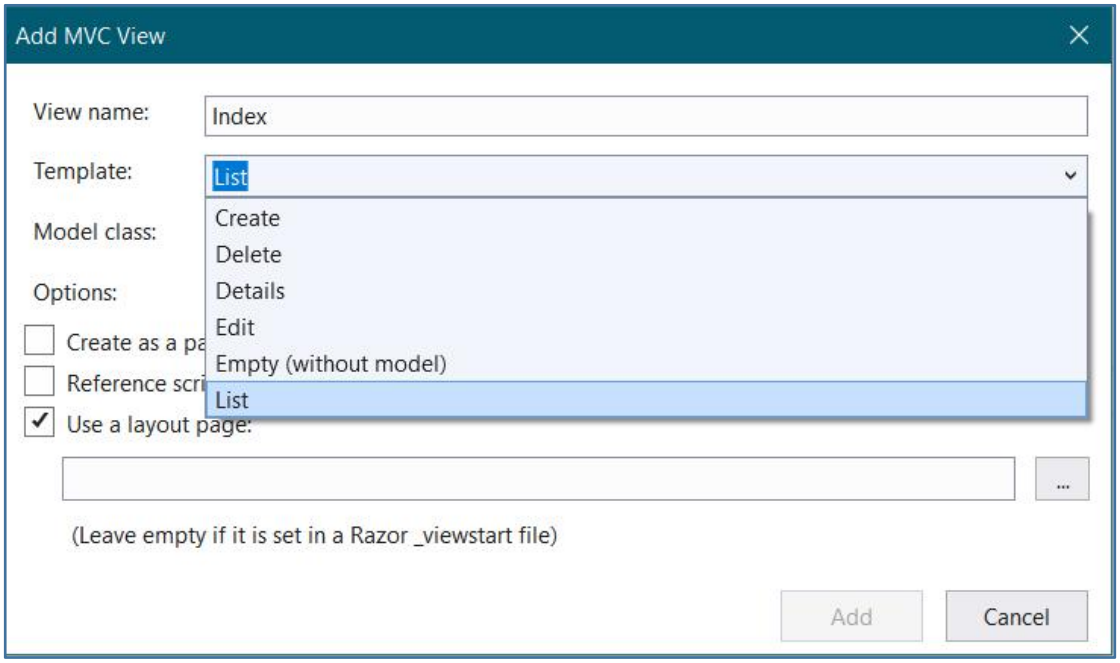
الصورة 26 - إضافة الصفحة تلقائيا

ستطالعك الواجهة التالية:



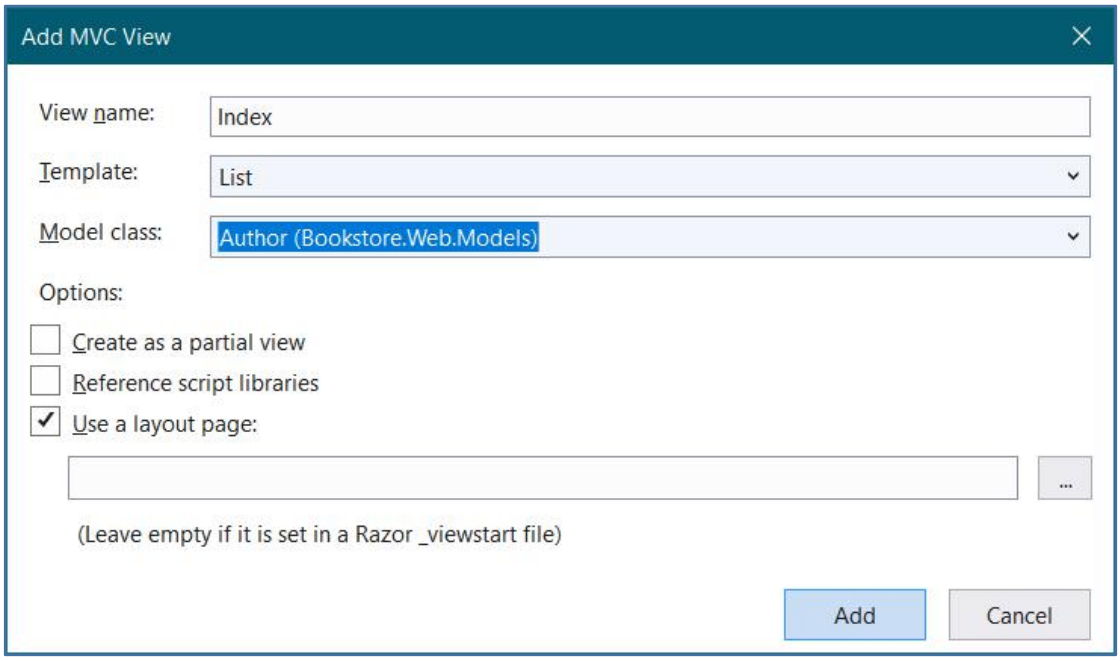
الصورة 27 - إعدادت الصفحة

اترك اسم View كما هو، فقط قم بتغيير Template واختر من القائمة List:



الصورة 28 - نوع الصفحة

ثم في Model Class قم باختيار الموديل الخاص بنا وهو Author، لتصبح الاختيارات في الأخير كما يلي:



الصورة 29 - تحديد الموديل

اضغط على الزر Add ولاحظ كيف سيتم بناء الصفحة Index بمعطيات الموديل Author بشكل تلقائي.

قبل أن نقوم بتنفيذ الصفحة، لابد أن نقوم بتسجيل AuthorRepository في Dependency Injection System ليتم التعرف عليه، لذلك لندخل إلى ملف Startup.cs، ثم إلى الوظيفة ConfigureServices، ونضيف إليها ما يلي:

Startup.cs:

```
            services.AddSingleton<IRepository<Book>, BookRepository>();
            services.AddSingleton<IRepository<Author>, AuthorRepository>();
```

لاحظ أنني قمت أيضا بتسجيل Repository الخاصة بالكتب حتى لا ننساها حينما نصل إليها، بقي فقط أن نضيف MVC أيضا إلى services وذلك داخل نفس الوظيفة ConfigureServices لتصبح كما يلي:

Startup.cs:

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
            services.AddSingleton<IRepository<Book>, BookRepository>();
            services.AddSingleton<IRepository<Author>, AuthorRepository>();
        }
```

وفي الوظيفة Configure سنقوم ببعض الإعدادات اللازمة لتفعيل MVC Route، وكذلك عرض صفحة الأخطاء الخاصة بالمطورين Developer Exceptions Page، وذلك كما يلي:

Startup.cs:

```
 public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseMvcWithDefaultRoute();
        }
```

يمكننا أن نضيف بعض الأمور الأخرى كتفعيل استعمال الملفات الثابتة كالصور وملفات css و جافاسكريبت من خلال استدعاء الوظيفة UseStaticFiles، ويمكننا أيضا عرض أكواد الأخطاء المتعلقة ب Http response status codes، وذلك لتصير الوظيفة Configure كما يلي:

Startup.cs:

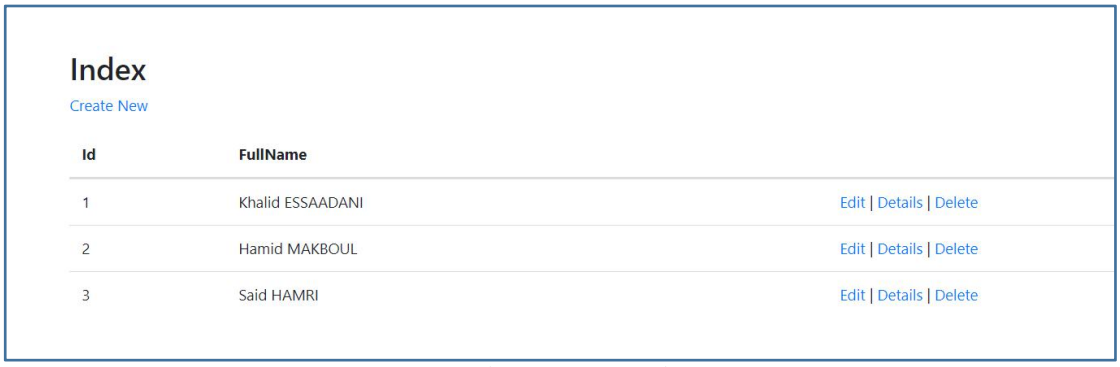
```
        public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
```

```
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();
    app.UseStatusCodePages();
    app.UseMvcWithDefaultRoute();
}
```

الآن قم بتنفيذ المشروع ثم انتقل إلى الصفحة Index عبر الرابط التالي:

https://localhost:44327/Author/Index

أو دون ذكر الكلمة Index لأنه سيذهب إليها تلقائيا، واحرص أن تغير رقم البورت 44327 إلى البورت عندك، ستلاحظ أن البيانات ظهرت بنجاح على الشكل التالي:



الصورة 30 - نتيجة التنفيذ

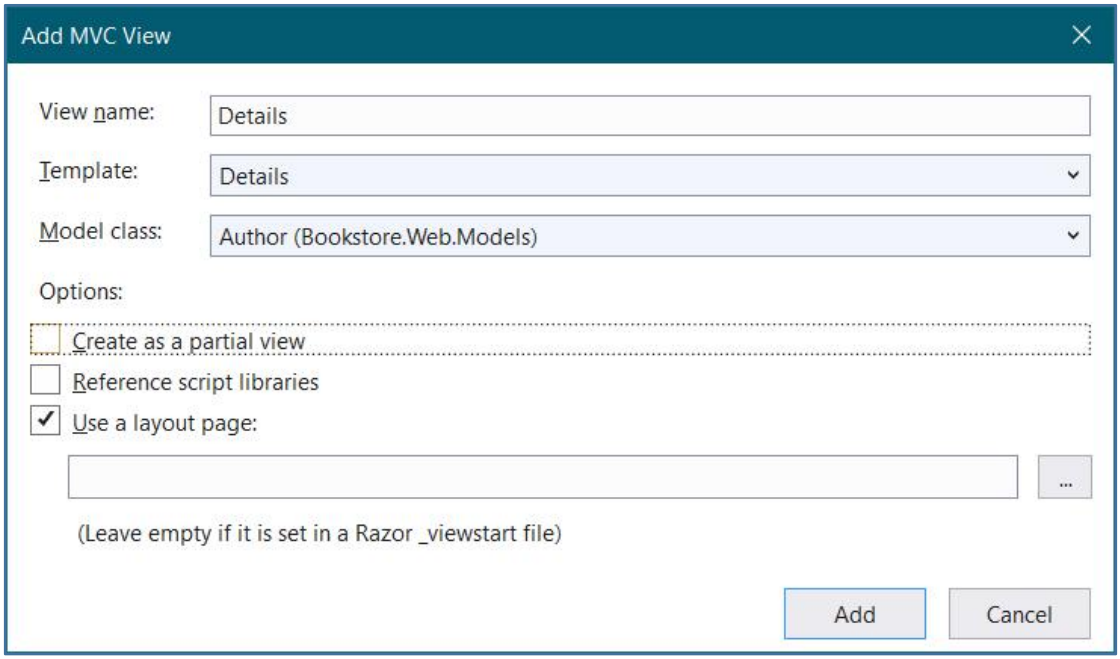
هكذا نكون قد أنجزنا المهمة الأولى بنجاح، الآن سنقوم بعملية عرض التفاصيل، بحيث حينما يضغط المستخدم على الأمر Details تظهر له تفاصيل الكاتب (على فرض أن هنالك بيانات أخرى غير ظاهرة على صفحة Index مثلا).

بعد ذلك سنعود إلى الكونترولر AuthorController ونضيف Action باسم Details وليكن محتواه كما يلي:

AuthorController.cs:

```
public ActionResult Details(int id)
{
    var author = Repository.GetById(id);

    return View(author);
}
```

ثم بنفس الكيفية نضغط على Action بيمين الماوس ونختار الأمر Add View، ونختار ما يلي:



الصورة 31 - إضافة صفحة التفاصيل

عند الضغط على الزر Add ستلاحظ أن الصفحة انضافت بنجاح وهذا هو محتواها:

Author/Details.cshtml:

```
@model Bookstore.Web.Models.Author

@{
    ViewData["Title"] = "Details";
}

<h2>Details</h2>

<div>
    <h4>Author</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.FullName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.FullName)
        </dd>
    </dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
    <a asp-action="Index">Back to List</a>
</div>
```

قبل التنفيذ سنقوم بتعديل طفيف على الصفحة Index، لندخل إليها ثم نذهب إلى الرابط الذي يسمح لنا بالانتقال إلى صفحة عرض التفاصيل ونزيل التعليق عن الجزء الذي يسمح لنا بتمرير معرف المؤلف المراد عرض تفاصيله كما يلي:

Author/Details.cshtml:

```
@Html.ActionLink("Details", "Details", new { id=item.Id }) |
```

ثم نكرر نفس الأمر مع رابط التعديل والحذف لأن كلا منهما يحتاج إلى معرف المؤلف المراد التعامل معه، بحيث تصبح الروابط الموجودة في صفحة Index على الشكل التالي:

Author/Details.cshtml:

```
<td>
    @Html.ActionLink("Edit", "Edit", new { id=item.Id}) |
    @Html.ActionLink("Details", "Details", new { id=item.Id }) |
    @Html.ActionLink("Delete", "Delete", new { id=item.Id })
</td>
```

الآن لنقم بالتنفيذ، ولنذهب إلى الصفحة Index الخاصة بالمؤلفين، ومنها نختر مؤلفا ما، ستلاحظ ظهور تفاصيله كما يلي:

الصورة 32 - عرض صفحة التفاصيل

جميل جدا، الآن سنتعرف على كيفية إضافة مؤلف جديد، تعالوا بنا ننشئ Create action في الكونترولر AuthorController وليكن محتواها على الشكل التالي:

AuthorController.cs:

```
public ActionResult Create()
{
    return View();
}
```

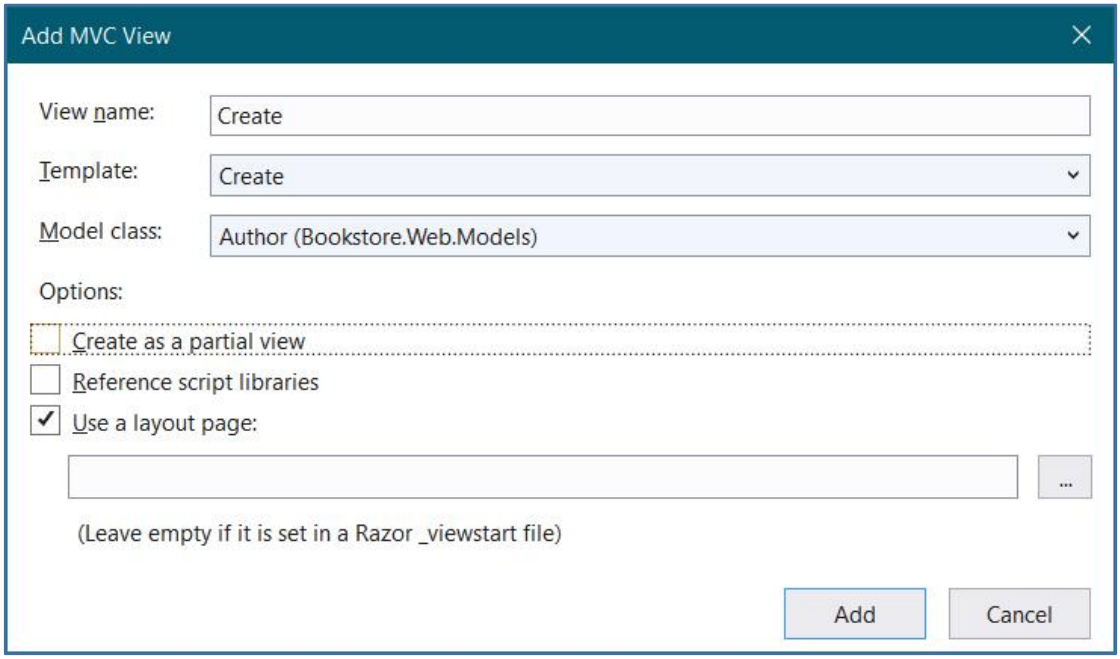
هذه المرة سأحتاج إلى تركيزك أكثر لأن الأمر مختلف بعض الشيء، حيث أن Create action التي ستسمح لنا بإنشاء المؤلفين ستحتاج إلى وظيفتين two actions، الأولى تعمل في وضع HTTP GET أي عند الدخول إلى صفحة الإنشاء وهي تقوم بعرض فورم أساسي فارغ خاص بإدخال معلومات المؤلف الجديد، ثم سنحتاج إلى action ثانية تعمل في وضع HTTP POST ودورها هو إرسال البيانات التي يقوم المستخدم بإدخالها في الفورم.

الوظيفة الأولى التي تسمح بإنشاء الفورم قد كتبناها وهذا هو محتواها كما ذكرنا قبل قليل:

AuthorController.cs:

```
public ActionResult Create()
{
    return View();
}
```

قبل أن ننشئ الوظيفة الثانية التي ترسل الفورم وتقوم بإضافة المؤلف، سنقوم بإنشاء الصفحة التي تعرض الفورم فارغا.وذلك كما جرت العادة من خلال الضغط بيمين الماوس على اسم Create action ثم اختيار الأمر Add View، وهذه المرة نقوم باختيار ما يلي:



الصورة 33 - إنشاء صفحة إضافة المؤلفين

الآن، سنقوم بإنشاء الوظيفة الثانية التي تقوم بإضافة المؤلف، ولأنها ستنفذ عند عمل post للفورم، سنقوم بوضع الواصفة HttpPost فوقها ليتم التفريق بينها وبين الوظيفة الأولى، في ASP.NET حينما لا نضع واصف من واصفات HTTP فإن الاختيار الافتراضي يكون هو GET.

محتوى الوظيفة الثانية كاملا كما يلي:

AuthorController.cs:

```
[HttpPost]
public ActionResult Create(Author author)
{
    try
    {
        Repository.Add(author);
        return RedirectToAction(nameof(Index));
    }
    catch
    {
        return View();
    }
}
```

الوظيفة باختصار تستقبل البيانات القادمة من الفورم وهي البيانات التي تشكل Author object، ثم تقوم بإضافته إلى Repository، ثم بعد ذلك ترجعنا إلى الصفحة Index بواسطة الأمر:

```
return RedirectToAction(nameof(Index));
```

لو نفذنا التطبيق وانتقلنا إلى الرابط التالي مع تغيير رقم البورت:

https://localhost:44327/Author/Create

ستلاحظ ظهور الفورم التالي:

الصورة 34 - عرض صفحة إضافة المؤلفين

عند تعبئة الفورم بمعرف واسم المؤلف ثم الضغط على الزر Create ستلاحظ أن عملية الإضافة تتم بنجاح ويتم إرجاعنا إلى الصفحة Index.

لو دخلنا إلى الأوامر البرمجية الخاصة بالصفحة Create.cshtml سنجدها على الشكل التالي:

Author/Create.cshtml:

```
@model Bookstore.Web.Models.Author

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Author</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly"
class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FullName" class="control-label"></label>
                <input asp-for="FullName" class="form-control" />
                <span asp-validation-for="FullName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>
```

ستلاحظ أن الفورم تم بناؤه بواسطة آلية Tag helper التي سنراها لاحقا، نفس الأمر بالنسبة لباقي الأدوات فسوف نلاحظ وجود أوسمة من قبيل asp-for و asp-validation-for وهي عبارة عن Tag helpers تقوم ببناء أدوات HTML وتقوم كذلك بعرض رسائل ناتجة عن عملية التحقق من البيانات (Model validation) وهي أمور سنراها في وقت لاحق إن شاء الله.

الآن سنبرمج عملية التعديل، وهي شبيهة كثيرا بعملية الإضافة، ولأننا سنحتاج إلى عرض الفورم بمعطيات المؤلف المراد تعديله، وكذلك سنحتاج إلى إرسال البيانات الجديدة، فإننا سوف نقوم بإنشاء وظيفتين كما هو الحال مع Create actions.

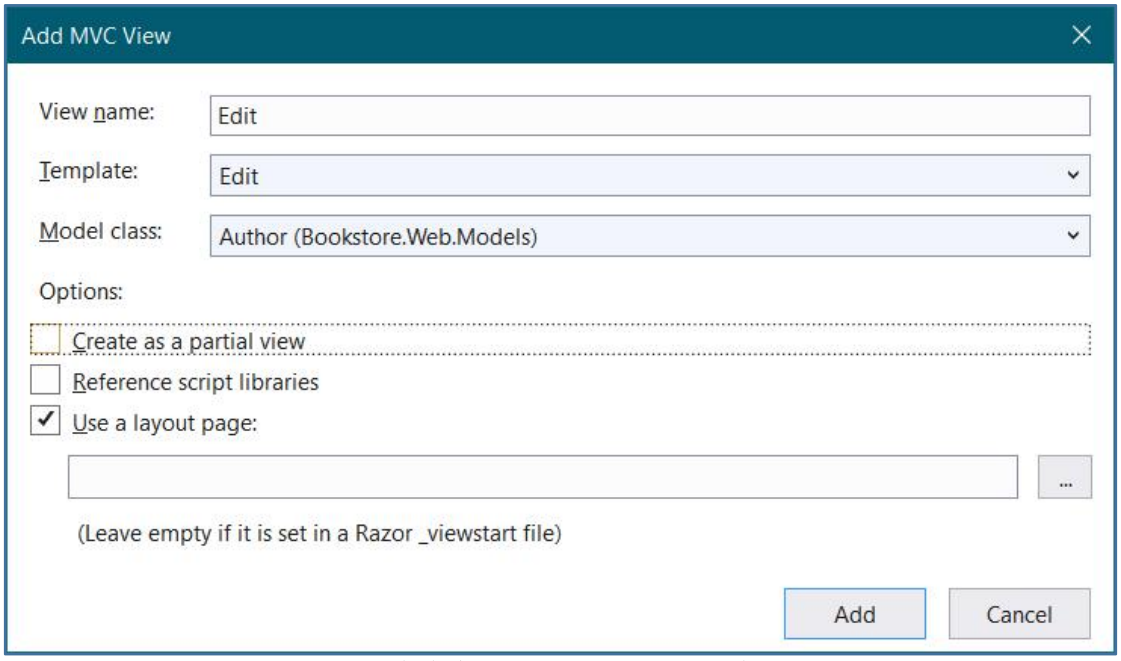
سنسمي الوظيفتين Edit، وهذا هو محتواهما:

AuthorController.cs:

```
        public ActionResult Edit(int id)
        {
            var author = Repository.GetById(id);

            return View(author);
        }


        [HttpPost]
        public ActionResult Edit(int id, Author author)
        {
            try
            {
                Repository.Edit(id, author);

                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

الوظيفة الأولى تبحث عن المؤلف وتقوم بإرجاع بياناته، بينما الوظيفة الثانية تستقبل البيانات القادمة من الصفحة ثم تقوم بعملية تعديل المؤلف من خلال الوظيفة Edit الموجودة على مستوى AuthorRepository.

الآن سننشئ Edit View بنفس الكيفية، الضغط بيمين الماوس داخل واحدة من Edit actions ثم نختار الأمر Add View، ومنه نحدد الخيارات التالية:



الصورة 35 - إضافة صفحة تعديل المؤلفين

سيتم إنشاء الصفحة بالمحتوى التالي:

Author/Edit.cshtml

```
@model Bookstore.Web.Models.Author

@{
    ViewData["Title"] = "Edit";
}

<h2>Edit</h2>

<h4>Author</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Edit">
            <div asp-validation-summary="ModelOnly"
class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Id" class="control-label"></label>
                <input asp-for="Id" class="form-control" />
                <span asp-validation-for="Id" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="FullName" class="control-label"></label>
                <input asp-for="FullName" class="form-control" />
                <span asp-validation-for="FullName"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>
```

الفورم شبيه بفورم الإضافة، يمكننا أن نستعمل فورما واحدا لكن لا مشكلة الآن، نكتفي بهذا القدر وننتقل من أجل الانتهاء من العمليات الأساسية، لنقم بعملية الحذف.

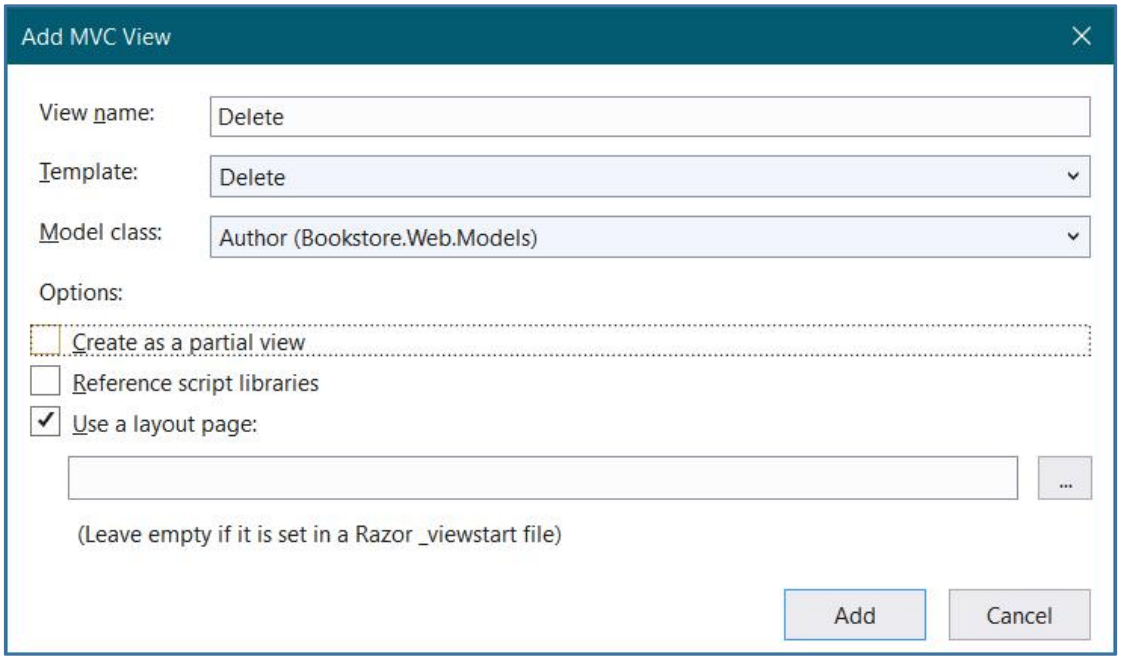
لأن عملية الحذف أيضا تتطلب عرض المؤلف المراد حذفه، ثم إرسال معلوماته ليتم حذفه سنحتاج إلى وظيفتين، واحدة تعمل في الوضع GET والثانية عند عمل POST لبيانات الفورم، وهما كما يلي:

AuthorController.cs:

```
        public ActionResult Delete(int id)
        {
            var author = Repository.GetById(id);
```

```
        return View(author);
    }


    [HttpPost]
    public ActionResult Delete(int id, Author author)
    {
        try
        {
            Repository.Delete(id);

            return RedirectToAction(nameof(Index));
        }
        catch
        {
            return View();
        }
    }
```

نأتي الآن ونقوم بتصميم صفحة الحذف من خلال الضغط بيمين الماوس على اسم Delete action ثم نختار الأمر Add View ومنه نختار ما يلي:



الصورة 36 - إضافة صفحة حذف المؤلفين

ليتم بذلك إنشاء الصفحة الآتية التي تعرض معلومات المؤلف المراد حذفه:

Author/Delete.cshtml:

```
@model Bookstore.Web.Models.Author

@{
    ViewData["Title"] = "Delete";
}
```

```
<h2>Delete</h2>

<h3>Are you sure you want to delete this?</h3>
<div>
    <h4>Author</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.FullName)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.FullName)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="submit" value="Delete" class="btn btn-default" /> |
        <a asp-action="Index">Back to List</a>
    </form>
</div>
```

لو قمنا بتنفيذ التطبيق، وذهبنا إلى الصفحة Index، ثم اختبرنا آليات التعديل والحذف سنجدها تعمل بشكل جيد.

هكذا نكون قد تعرفنا على أبرز العمليات التي نستطيع القيام بها في ASP.NET Core MVC، بنفس الكيفية يمكننا أن ننشئ Controller خاص بالكتب، وهو سيحتوي على بعض الأمور المختلفة لذلك سنتركه للفصل المقبل إن شاء الله.

سنقوم بإنشاء كونترولر جديد نسميه BookController داخل مجلد Controllers، وسنقوم بحقن BookRepository بواسطة مشيد الكونترولر كما يلي:

BookController.cs:

```
        public IRepository<Book> BookRepository { get; }

        public BookController(IRepository<Book> bookRepository)
        {
            BookRepository = bookRepository;
        }
```

بالنسبة للوظيفة Index فهي شبيهة بالتي أنشأناها للمؤلفين، لذلك ضع محتواها كما يلي وقم بإضافة View المناسبة لها وهي من نوع List لو تتذكر جيدا:

BookController.cs:

```
public ActionResult Index()
{
    var books = BookRepository.GetAll();

    return View(books);
}
```

عند عرض صفحة الكتب ستبدو كما يلي:

الصورة 37 - عرض صفحة الكتب

بنفس الكيفية التي رأيناها سابقا، قم بالتعديل على روابط الحذف والتعديل وعرض التفاصيل في الصفحة Index.cshtml لتسمح بنقل قيمة البرامتر Id كما يلي:

Book/Index.cshtml:

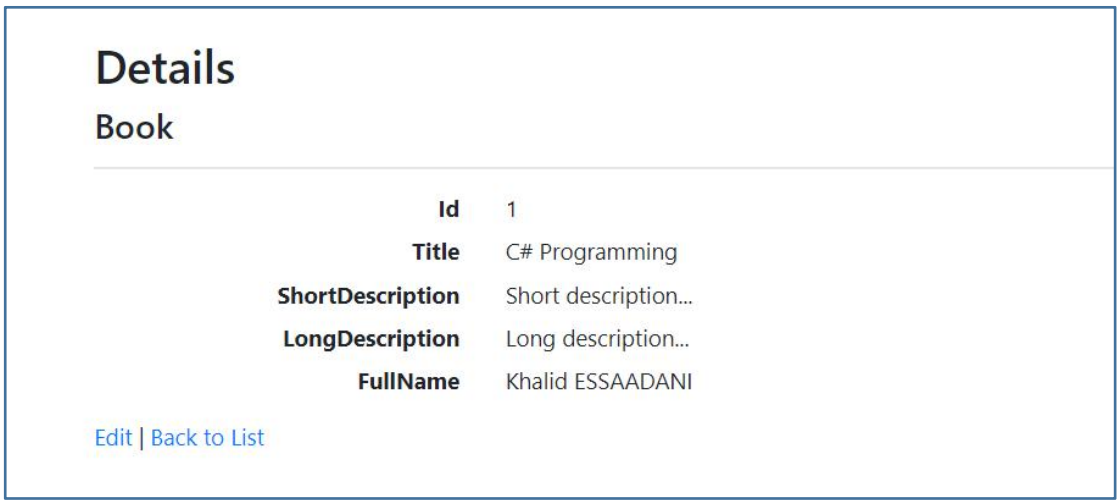
```
<td>
    @Html.ActionLink("Edit", "Edit", new { id=item.Id}) |
    @Html.ActionLink("Details", "Details", new {  id=item.Id  }) |
    @Html.ActionLink("Delete", "Delete", new {  id=item.Id })
</td>
```

الوظيفة Details الخاصة بعرض تفاصيل الكتاب شبيهة جدا بالتي استعملناها مع المؤلفين، وهذا هو محتواها:

BookController.cs:

```
public ActionResult Details(int id)
{
    var book = BookRepository.GetById(id);

    return View(book);
}
```

قم فقط بإنشاء الصفحة الخاصة بها، وقم بالتنفيذ لاستعراض النتيجة التي ستكون كما يلي:



الصورة 38 - عرض تفاصيل الكتب

شاشة العرض المختلط أو ViewModel

الآن سنقوم بإنشاء الوظيفتين اللتين تسمحان لنا بإنشاء كتاب جديد، وسنسميهما Create، الأولى تقوم ببناء الفورم وهي تعمل في وضع GET والثانية تقوم باستقبال البيانات القادمة من الفورم وهي تعمل في وضع POST، ولنبدأ مع الأولى التي تعمل في وضع GET والتي تبني الفورم.

قبل أن ننشئها علينا أن ننتبه إلى أن View الخاصة بإضافة الكتب ستحتاج منا أيضا إلى عرض قائمة منسدلة select list تحتوي على أسماء المؤلفين من أجل إضافة مؤلف الكتاب.

إذن سنقوم بتصميم Model جديد يحتوي على الحقول التي سنحتاجها من باب تسهيل عملنا، الموديل الذي يحتوي على البيانات الإضافية يسمى ViewModel لأنه يكون موجها بالأساس للعمل مع View، لذلك تعالوا معي ننشئ مجلدا جديدا نسميه ViewModels، ولننشئ بداخله كلاس جديد نسميه BookAuthorViewModel، وليكن محتواه كما يلي:

BookAuthorViewModel.cs:

```
public class BookAuthorViewModel
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public string LongDescription { get; set; }
    public string ShortDescription { get; set; }
    public int AuthorId { get; set; }

    public IList<Author> Authors { get; set; }
}
```

هذا View Model هو الذي سيقدم لنا جميع البيانات التي سنحتاجها في View، فهو يعطينا معلومات الكتاب، ومعرف الكاتب، وكذلك قائمة المؤلفين والتي سنعرضها في قائمة منسدلة.

الآن سنتعامل مع هذا View Model بدل التعامل بشكل مباشر مع Book، لننتقل إلى BookController ولننشئ الوظيفة الأولى Create وليكن محتواها كما يلي:

BookController.cs:

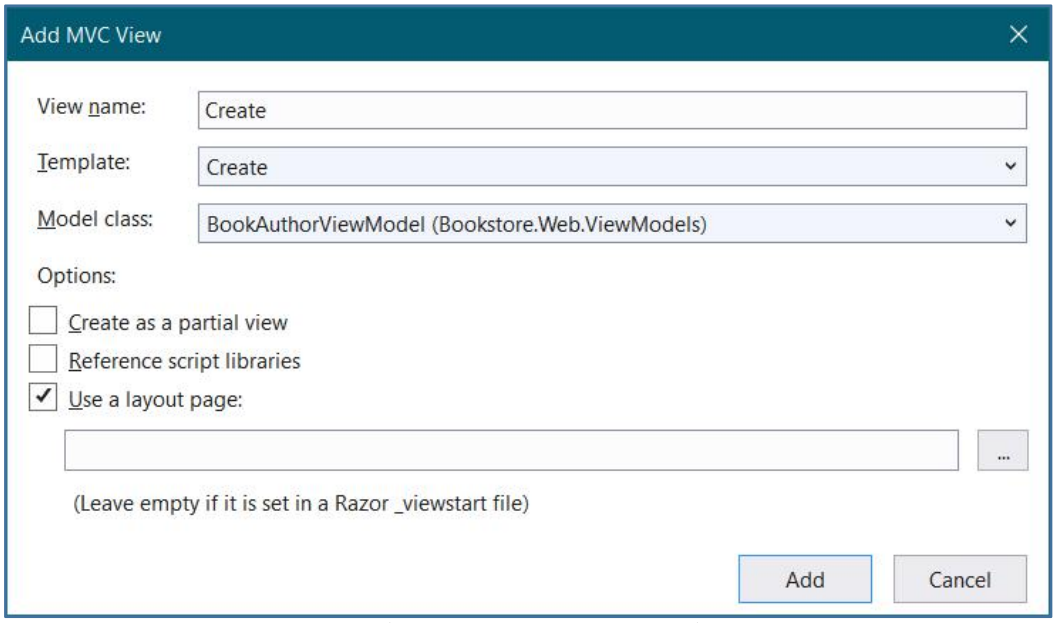
```
public ActionResult Create()
{
    BookAuthorViewModel viewModel = new BookAuthorViewModel
    {
        Authors = AuthorRepository.GetAll()
    };

    return View(viewModel);
}
```

الوظيفة باختصار ترجع لنا ViewModel فارغ باستثناء لائحة المؤلفين التي يتم جلبها من AuthorRepository، إذن سنحتاج إلى عمل Injection لهذه repository في مشيد الكونترولر كما يلي:

BookController.cs:

```
public IRepository<Author> AuthorRepository { get; }

public BookController(IRepository<Book> bookRepository,
    IRepository<Author> authorRepository)
{
    BookRepository = bookRepository;
    AuthorRepository = authorRepository;
}
```

بعد ذلك، سنقوم ببناء الصفحة من خلال الضغط بيمين الماوس على Create Action واختيار Add View كما يلي:



الصورة 39 - إنشاء صفحة إضافة الكتب

لاحظ أننا اخترنا في Model class اسم BookAuthorViewModel، الآن لنقم بتعديل الصفحة الافتراضية لتصير على الشكل التالي:

Book/Create.cshtml:

```
@model Bookstore.Web.ViewModels.BookAuthorViewModel

@{
    ViewData["Title"] = "Create";
}

<h2>Create</h2>

<h4>Book</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly"
class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ShortDescription"
class="control-label"></label>
                <input asp-for="ShortDescription" class="form-control" />
                <span asp-validation-for="ShortDescription"
class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="LongDescription"
class="control-label"></label>
                <input asp-for="LongDescription" class="form-control" />
                <span asp-validation-for="LongDescription"
class="text-danger"></span>
            </div>

            <div class="form-group">
                <label asp-for="AuthorId" class="control-label"></label>
                <select asp-for="AuthorId" class="form-control" size="1"
                        asp-items="@(new
SelectList(Model.Authors,"Id","FullName"))"></select>
                <span asp-validation-for="Authors" class="text-danger"></span>
            </div>

            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-success" />
            </div>
        </form>
    </div>
</div>

<div>
```

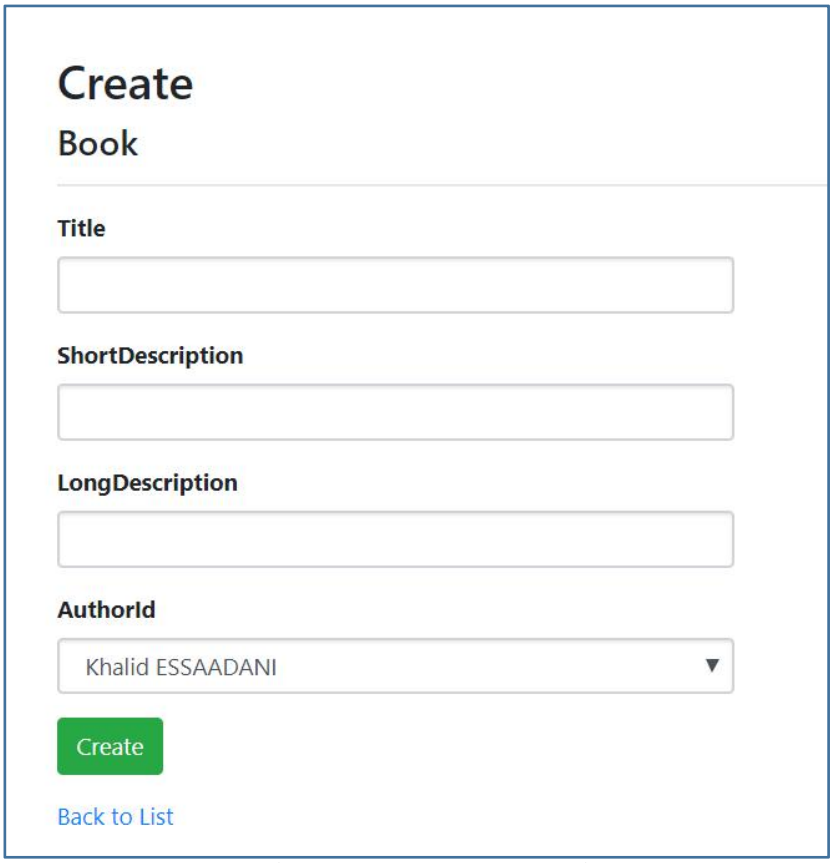
```
    <a asp-action="Index">Back to List</a>
</div>
```

انتبه إلى الجزء الخاص بتعبئة القائمة المنسدلة وهو:

```
            <div class="form-group">
                <label asp-for="AuthorId" class="control-label"></label>
                <select asp-for="AuthorId" class="form-control" size="1"
                        asp-items="@(new
SelectList(Model.Authors,"Id","FullName"))"></select>
                <span asp-validation-for="Authors" class="text-danger"></span>
            </div>
```

حيث قمنا بتحديد مصدر البيانات وهو Authors التي ترجعها لنا BookAuthorViewModels والتي عبأناها في Create action التي تعمل في وضع GET.

لو نفذنا الصفحة واستعرضناها سنجدها كما يلي:



الصورة 40 - عرض صفحة إضافة الكتب

الآن سنقوم بإنشاء الوظيفة الثانية التي تعمل submit للفورم، وسيكون محتواها كما يلي:

BookController.cs:

```
        [HttpPost]
        public ActionResult Create(BookAuthorViewModel viewModel)
        {
            try
            {
                var book = new Book
                {
                    Title=viewModel.Title,
                    LongDescription=viewModel.LongDescription,
                    ShortDescription=viewModel.ShortDescription,
                    Author = new Author
                    {
                        Id = viewModel.AuthorId,
                        FullName =
AuthorRepository.GetById(viewModel.AuthorId).FullName
                    }
                };

                BookRepository.Add(book);
                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

الوظيفة تقوم ببعض الأمور التي لم نرها من قبل مثل إسناد القيم من BookAuthorViewModel القادم من الفورم إلى Object من نوع Book، ثم تقوم بالبحث عن مؤلف الكتاب بواسطة معرفه القادم أيضا من View Model وتسند اسمه إلى اسم مؤلف الكتاب، ثم في الختام تضيف الكتاب بمعلوماته إلى BookRepository.

عملية التعديل تشتغل تقريبا بنفس السيناريو، محتوى وظيفتي التعديل كما يلي:

BookController.cs:

```
        public ActionResult Edit(int id)
        {
            var book = BookRepository.GetById(id);
            BookAuthorViewModel viewModel = new BookAuthorViewModel
            {
                BookId=book.Id,
                Title=book.Title,
                LongDescription=book.LongDescription,
                ShortDescription=book.ShortDescription,
                AuthorId=book.Author.Id,
                Authors = AuthorRepository.GetAll()
            };

            return View(viewModel);
        }
```

```
        [HttpPost]
        public ActionResult Edit(int id, BookAuthorViewModel viewModel)
        {
            try
            {
                var editedBook = new Book
                {
                    Id=viewModel.BookId,
                    Title=viewModel.Title,
                    LongDescription=viewModel.LongDescription,
                    ShortDescription=viewModel.ShortDescription,
                    Author=new Author
                    {
                        Id=viewModel.AuthorId,

FullName=AuthorRepository.GetById(viewModel.AuthorId).FullName
                    }
                };


                BookRepository.Edit(id, editedBook);

                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

بهذه الكيفية نكون قد تعرفنا على أهم العمليات الممكن القيام بها على البيانات في ASP.NET Core.

إذا كان Razor لا يكفي، فلك في Tag Helpers غنية

تمثل Tag Helpers أحد أهم الإضافات التي عرفتها ASP.NET Core، حيث تسمح لنا بكتابة أوامر السيرفر بأسلوب HTML، ولمن استعمل HTML Helpers في الإصدارات السابقة من ASP.NET MVC لن يجد فرقا كبيرا بين المفهومين ماعدا في أسلوب الكتابة الذي صار مثل أوسمة HTML الاعتيادية.

لاستعمال Tag Helpers علينا إضافة الموجهة addTagHelper في الصفحة التي سنحتاج فيها إلى استدعاء Tag Helpers، كما يمكننا تعريف هذه الموجهة في ملف ViewImports لنستطيع استعمال Tag Helpers في جميع صفحات المشروع، لعمل ذلك سندخل إلى ملف _ViewImports.cshtml ونضيف إليه الموجهة التالية:

_ViewImports.cshtml:

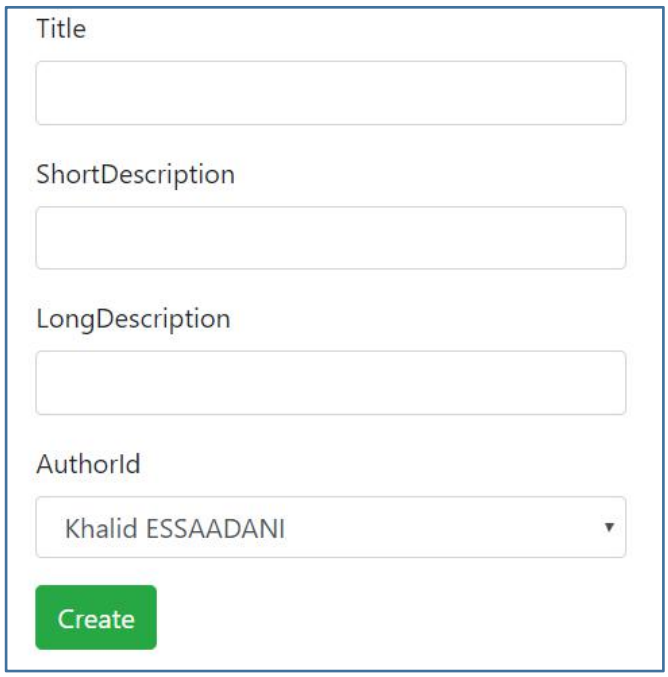
```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

الآن بإمكاننا استعمال Tag Helpers في صفحاتنا، سنشتغل على مثال بسيط نقوم من خلاله ببناء فورم كما يلي:

Book/Create.cshtml:

```
@model Bookstore.Web.ViewModels.BookAuthorViewModel

<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="ShortDescription"
class="control-label"></label>
                <input asp-for="ShortDescription" class="form-control" />
            </div>
            <div class="form-group">
                <label asp-for="LongDescription"
class="control-label"></label>
                <input asp-for="LongDescription" class="form-control" />
            </div>

            <div class="form-group">
                <label asp-for="AuthorId" class="control-label"></label>
                <select asp-for="AuthorId" class="form-control" size="1"
                        asp-items="@(new
SelectList(Model.Authors,"Id","FullName"))"></select>
            </div>

            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-success" />
            </div>
        </form>
    </div>
</div>
```

لاحظ أننا استعملنا Tag Helper من نوع form لبناء الفورم، ثم استعملنا نوعا آخر وهو asp-action الذي يسمح لنا بتحديد Action المراد تنفيذها عند عمل Post لهذا الفورم، بالإضافة إلى asp-for الذي يقوم بعمل ربط binding بين خصائص model وبين عناصر HTML، لو استعرضنا الصفحة ستظهر كما يلي:



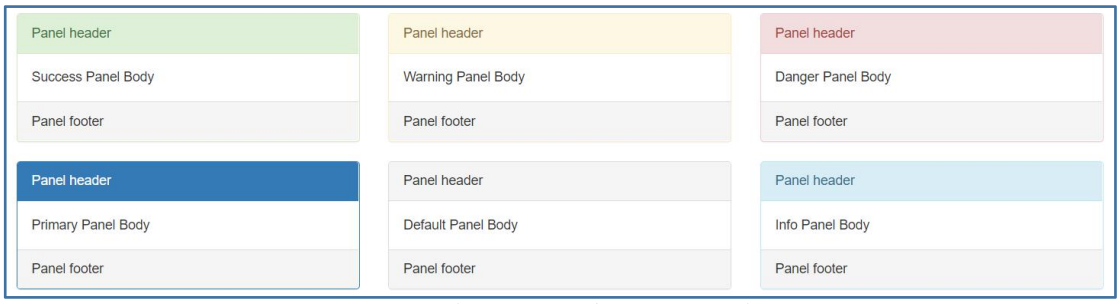
الصورة 41 - عرض صفحة إضافة الكتب

من أجل تحسين شكل الفورم استعملنا مكتبة Twitter Bootstrap، لكن ليس هذا هو موضوعنا لذلك لن نهتم بالشكل وسنركز على كيفية استعمال Tag Helpers ولنا عودة مع تويتر بوتستراب للفصل المقبل إن شاء الله.

توجد العديد من Tag Helpers من قبيل environment والتي من خلالها نحدد بيئة التطوير هل هي development أو غيرها، وتوجد كذلك asp-action التي تسمح لنا بالانتقال إلى action معينة وغير ذلك.

يمكننا أيضا أن ننشئ Tag Helpers خاصة بنا، إذ يكفي أن نقوم بإنشاء كلاس ترث من الكلاس TagHelper، ثم نعرف طريقة اشتغاله عبر إعادة تعريف الوظيفة Process.

سنقوم بإنشاء مثال كامل نشرح من خلاله كيف نقوم بإنشاء Tag Helpers خاصة بنا، حيث سنصمم وسما جديدا يبسط بناء panels التي تأتي مع البوتستراب:



الصورة 42 - شكل Panels في البوتستراب

لإنشاء panel باستعمال البوتستراب فإننا نحتاج إلى كتابة أوامر HTML كما يلي:

Book/Index.cshtml:

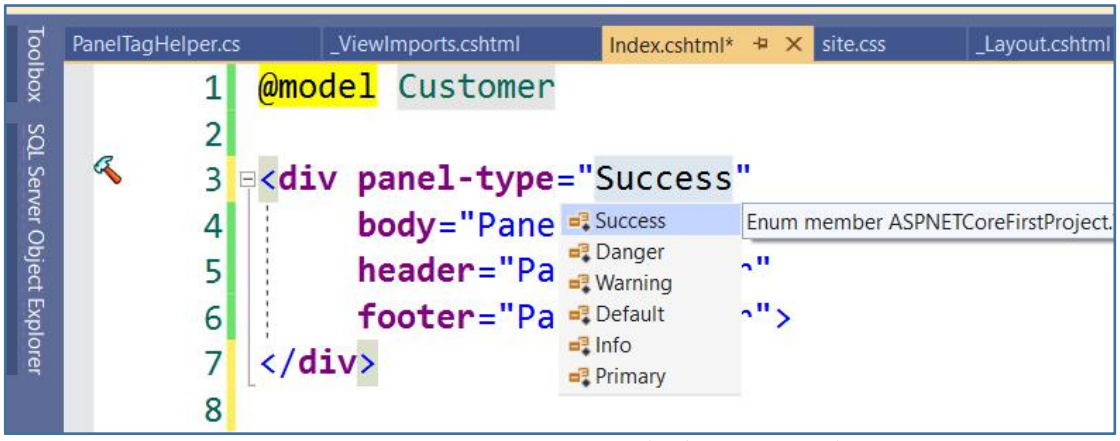
```
<div class="panel panel-default">
    <div class="panel-heading">Panel Heading</div>
    <div class="panel-body">Panel Content</div>
    <div class="panel-footer">Panel Footer</div>
</div>
```

سنقوم بتبسيط الكتابة من خلال استعمال وسم واحد لتكون كما يلي:

Book/Index.cshtml:

```
<div panel-type="Success"
     body="Panel body"
     header="Panel header"
     footer="Panel footer">
</div>
```

بالإضافة إلى أن نوع Panel سيكون ديناميكيا نختاره من خلال Enum نقوم بتعريفه في الكود، لتظهر لنا إمكانية اختيار نوع panel كما يلي عند التصميم:



الصورة 43 - مثال على Intellisense مع Tag Helpers

لإنشاء Tag helper، تعالوا بنا ننشئ مجلدا جديدا نسميه مثلا TagHelpers، ولنضف بداخله كلاس نسميه PanelTagHelper.cs، هذا الكلاس باختصار سيرث من الكلاس الرئيسي TagHelper، ثم سنعرف بداخله الخصائص التي سنحتاجها كما يلي:

PanelTagHelper.cs:

```
    [HtmlTargetElement("div", Attributes = "panel-type")]
    public class PanelTagHelper : TagHelper
    {
```

```
        [HtmlAttributeName("panel-type")]
        public PanelTypesEnum PanelType { get; set; }
        [HtmlAttributeName("header")]
        public string Header { get; set; }
        [HtmlAttributeName("body")]
        public string Body { get; set; }
        [HtmlAttributeName("footer")]
        public string Footer { get; set; }

        public override void Process(TagHelperContext context, TagHelperOutput
output)
        {
            string content = $@"<div class='panel-heading'>{Header}</div>
                                <div class='panel-body'>{Body}</div>
                                <div class='panel-footer'>{Footer}</div>";

            output.Attributes.SetAttribute("class", $"panel panel-" +
PanelType.ToString().ToLowerInvariant());

            output.Content.AppendHtml(content);

            base.Process(context, output);
        }
    }
```

في السطر الأول استعملنا الواصفة HtmlTargetElement التي حددنا فيها نوع العنصر الذي سنشتغل عليه وفي حالتنا هذه هو div، ثم أضفنا الخصائص اللازمة، ووضعنا فوق كل خاصية الاسم الذي سنستعمله في الصفحات من خلال الواصفة HtmlAttributeName.

بعد ذلك أعدنا تعريف الوظيفة Process، التي وضعنا فيها الأوسمة التي تسمح لنا بإنشاء panel باستعمال البوتستراب، مع تغيير القيم الديناميكة ووضع أسماء الخصائص الخاصة بنا مكانها.

ثم قمنا بإضافة هذا المحتوى إلى output الذي سنصدره للاستعمال في الصفحات، بقي فقط أن ننشئ enum التي تحتوي على أنواع Panels والتي استعملناها في الكود أعلاه، وهذا هو محتواها:

PanelTagHelper.cs:

```
    public enum PanelTypesEnum
    {
        Default,
        Primary,
        Success,
        Info,
        Warning,
        Danger
    }
```

الآن، سنذهب إلى ملف ViewImports.cs وسنضيف مجال الأسماء الذي يحتوي على Tag Helper الذي أنشأناه، لكي نتمكن من استعماله في مشروعنا:

_ViewImports.cshtml:

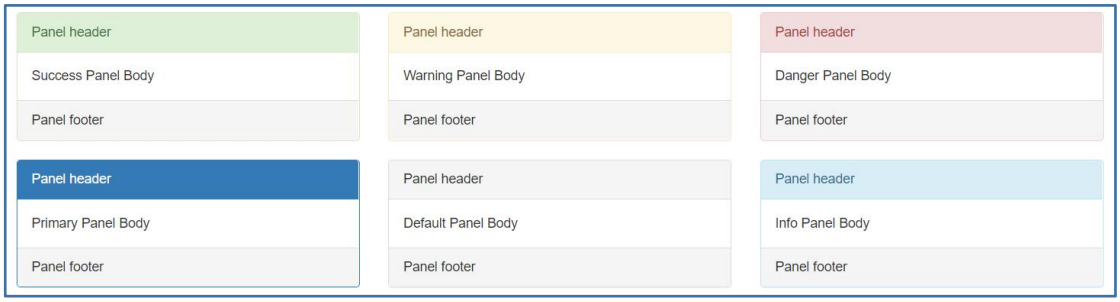
```
@addTagHelper *, Bookstore.Web
```

حيث Bookstore.Web هو مجال الأسماء الرئيسي، الآن يمكننا أن نستعمل هذا Tag Helper حيثما شئنا، وهذه أمثلة على ذلك:

Book/Index.cshtml:

```
<div class="row">
    <div class="col-md-4">
        <div panel-type="Success"
             body="Success Panel Body"
             header="Panel header"
             footer="Panel footer">
        </div>
    </div>
    <div class="col-md-4">
        <div panel-type="Warning"
             body="Warning Panel Body"
             header="Panel header"
             footer="Panel footer">
        </div>
    </div>
    <div class="col-md-4">
        <div panel-type="Danger"
             body="Danger Panel Body"
             header="Panel header"
             footer="Panel footer">
        </div>
    </div>
    <div class="col-md-4">
        <div panel-type="Primary"
             body="Primary Panel Body"
             header="Panel header"
             footer="Panel footer">
        </div>
    </div>
    <div class="col-md-4">
        <div panel-type="Default"
             body="Default Panel Body"
             header="Panel header"
             footer="Panel footer">
        </div>
    </div>
    <div class="col-md-4">
        <div panel-type="Info"
             body="Info Panel Body"
             header="Panel header"
             footer="Panel footer">
        </div>
    </div>
</div>
```

عند التنفيذ سنحصل على النتائج التالية:

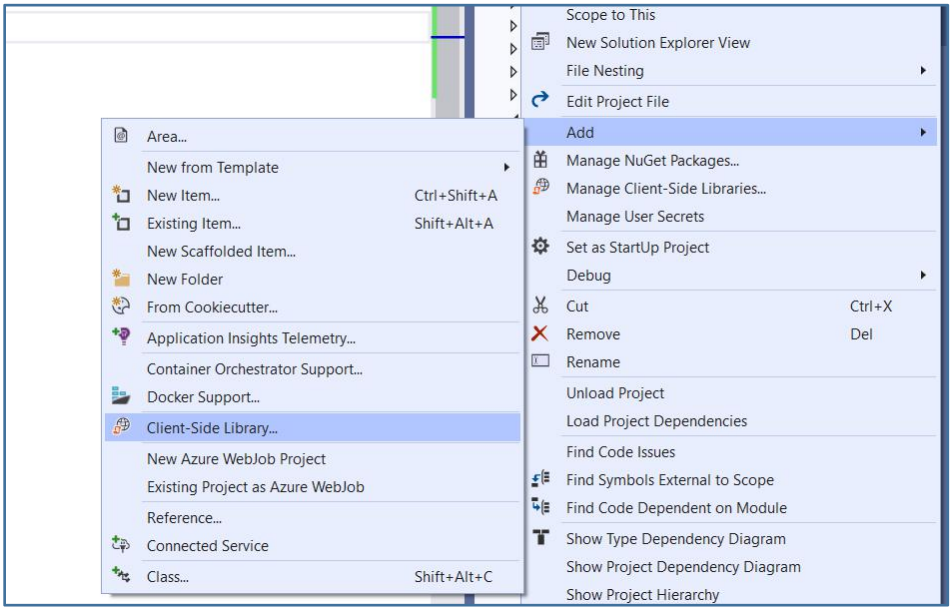


الصورة 44 - نتيجة التنفيذ

أداة LibMan مدير المكتبات الأنيق!

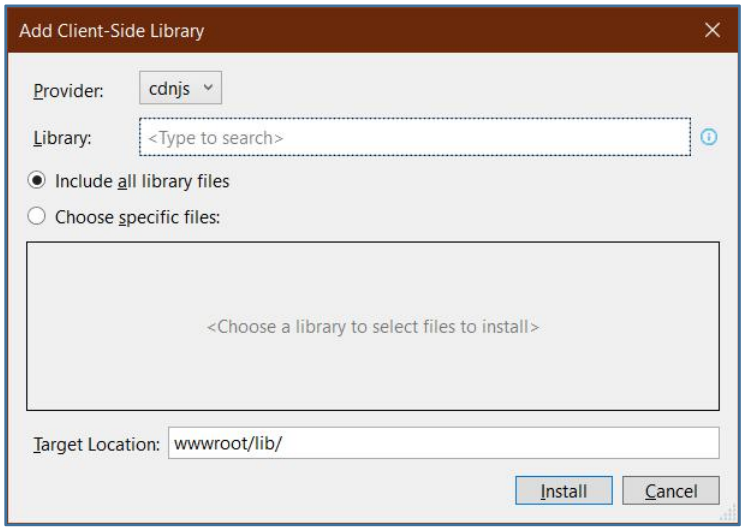
من المعلوم أننا نستطيع التعامل مع الملفات في مشروعنا من خلال إضافتها يدويا إلى مجلد wwwroot، لكن أحيانا نحتاج إلى تسريع وثيرة الشغل وتفويض مهمة إدارة المكتبات والملفات لأداة تقوم بذلك، في وقت سابق كنا نستعمل أداة Bower التي تسمح لنا بتحميل المكتبات التي نحتاجها مثل jquery و بوتستراب وغيرها، كما يمكننا ذلك أيضا من خلال استعمال Node Package Manager التي تأتي مع بيئة NodeJS والتي تسمح لنا بتحميل الملفات اللازمة، لكن في ASP.NET Core صارت الأمور أسهل، وصرنا نستطيع تحميل المكتبات من خلال أداة LibMan والمعروفة كذلك ب Library Manager، للتعرف أكثر على هذه الأداة، تعالوا بنا نقوم بتحميل البوتستراب باستعمالها.

في الأول نقوم بالضغط بيمين الماوس على اسم المشروع، ثم نختار الأمر Add، ثم نختار الأمر Client-Side Library:



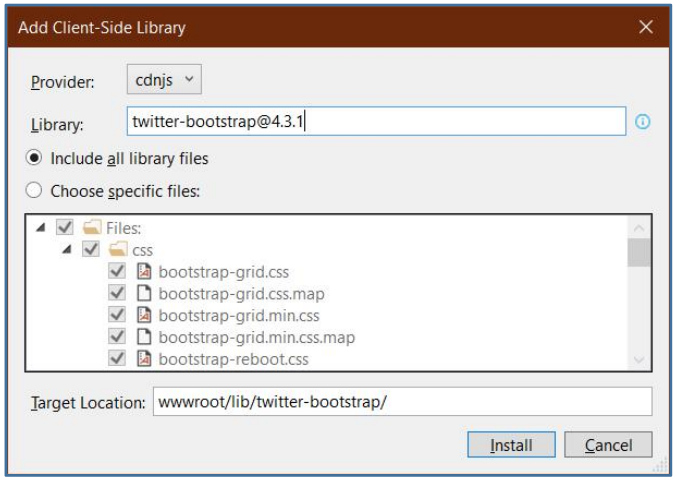
الصورة 45 - إضافة Client-Side Library

بعد ذلك ستطالعنا الواجهة التالية:



الصورة 46 - واجهة إضافة المكتبات

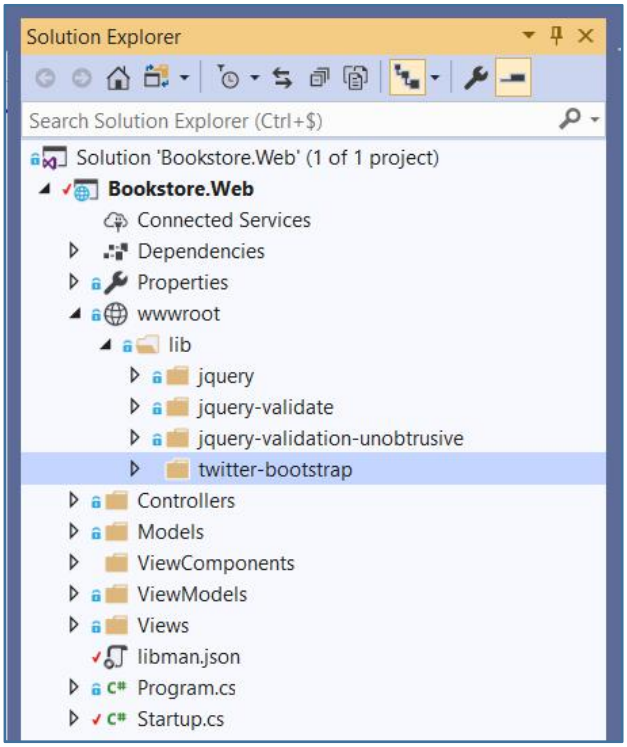
في مربع النص الخاص ب Library نكتب اسم المكتبة التي نريد تحميلها، وليكن مثلا twitter bootstrap باعتبار أن هذا هو الاسم الحقيقي لمكتبة البوتستراب بسبب أن تويتر هي من أصدرته، ستلاحظ وأنت تكتب أنه يعطيك خيارات حسب القيمة التي كتبتها، قم باختيار twitter-bootstrap وستلاحظ أيضا أنه قام باختيار أحدث إصدار كما تبين الصورة التالية:



الصورة 47 - البحث عن مكتبة معينة

قم بتحديد الملفات التي ستحتاجها من المكتبة كملفات css و ملفات جافاسكريبت، ثم من Target Location قم باختيار المسار الذي تريد، ثم اضغط على الزر Install.

بعد ذلك ستلاحظ أنه تم تثبيت المكتبة في المسار المحدد:



الصورة 48 - بنية المجلد

وستلاحظ كذلك ظهور ملف جديد باسم libman.json:

لو دخلت إليه ستجد محتواه كما يلي:

Libman.json:

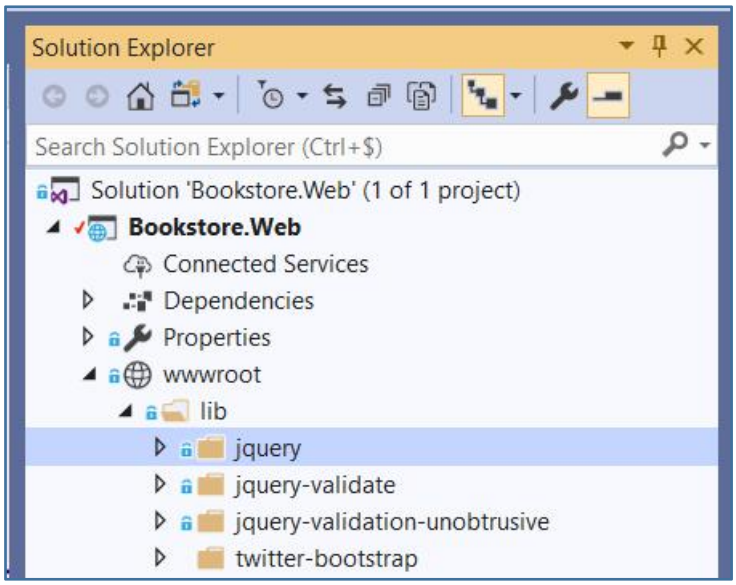
```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "twitter-bootstrap@4.3.1",
      "destination": "wwwroot/lib/twitter-bootstrap/"
    }
  ]
}
```

الملف يحتوي على كافة الملفات التي قمنا بتحميلها باستعمال LibMan، يمكننا أيضا أن نقوم بتحميل الملفات من خلال إضافتها مباشرة إلى هذا الملف وعند حفظه سيقوم الفيجوال ستوديو بتحميل الملفات المطلوبة، مثلا يمكننا تحميل مكتبة jquery من خلال إضافة المكتبة إلى ملف libman.json كما يلي:

```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "twitter-bootstrap@4.3.1",
      "destination": "wwwroot/lib/bootstrap/"
    },
    {
      "library": "jquery-validation-unobtrusive@3.2.11",
      "destination": "wwwroot/lib/jquery-validation-unobtrusive/"
    },
    {
      "library": "jquery-validate@1.19.1",
      "destination": "wwwroot/lib/jquery-validate/",
      "files": [
        "jquery.validate.min.js",
        "jquery.validate.js"
      ]
    },
    {
      "library": "jquery@3.4.1",
      "destination": "wwwroot/lib/jquery/"
    }
  ]
}
```

الصورة 49 - ملف libman.json

لاحظ أنه تم تحميل المكتبة بنجاح بمجرد ما نعمل حفظ لملف libman.json:



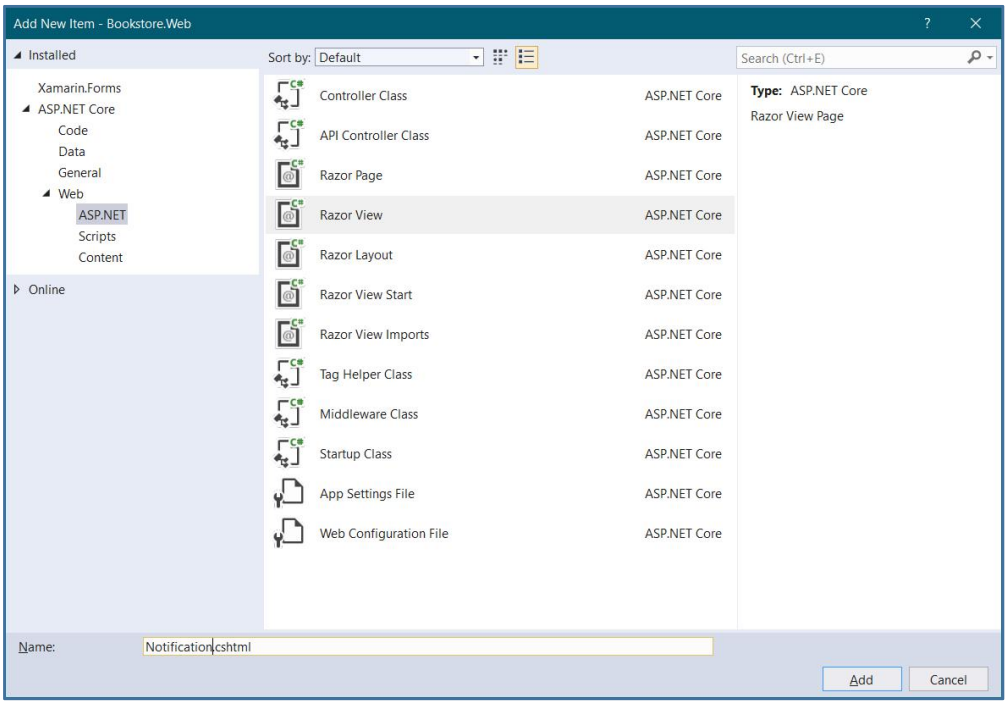
الصورة 50 - بنية المجلد بعد تحميل المكتبة

دور Partial Views في تجويد تصميم التطبيقات

نحتاج إلى الصفحات الفرعية أو Partial Views حينما نريد تصميم أجزاء من الصفحات التي نريد استعمالها في عدة صفحات، وهي شبيهة جدا بمفهوم User controls في تقنية ASP.NET Web Forms، والغرض من الصفحات الفرعية كما ذكرنا هو بناء أجزاء من الصفحات قابلة لإعادة الاستخدام، وغالبا ما نستخدمها في الصفحات المعقدة التي نقوم بتقسيمها إلى أجزاء صغيرة على شكل صفحات فرعية.

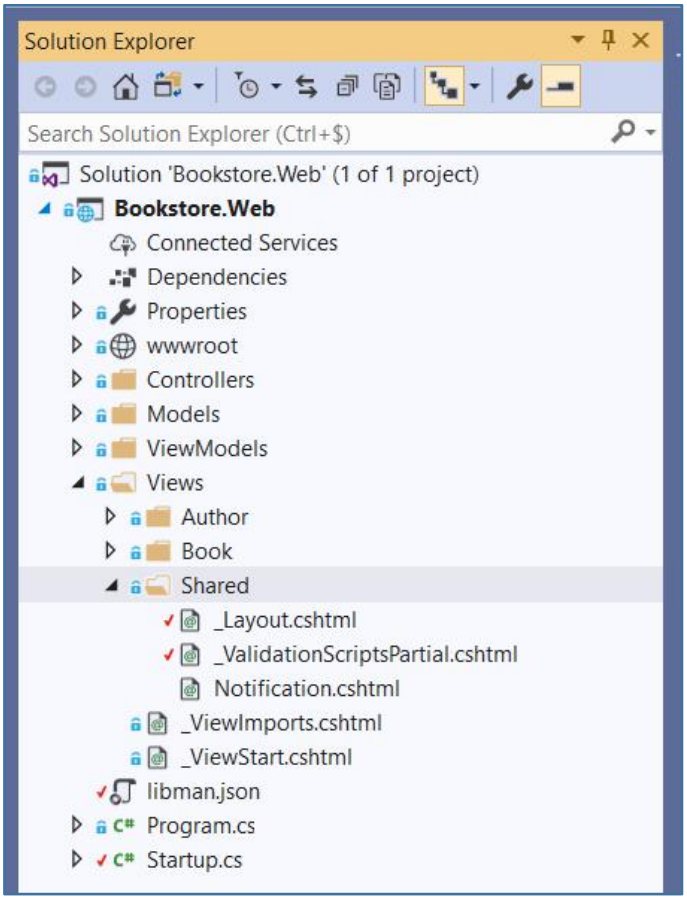
في الأول ننشئ Partial view وهي عبارة عن صفحة عادية، ثم بعد ذلك نقوم باستدعائها في الصفحات التي تحتاجها، والفرق بين الصفحات الفرعية Partial views والصفحات الاعتيادية Views هو أن هذه الأخيرة لا يتم تنفيذ محتواها إلا بعد تنفيذ محتوى ملف _ViewStart.cshtml بينما الصفحات الفرعية تنفذ مباشرة لأنها لا تحتاج إلى Layout أو إلى أي تصميم، لأن تصميمها سيكون مرتبطا بالصفحة التي تستدعيها كما لو أنها جزء منها.

تعالوا بنا الآن لنقوم بإنشاء أول صفحة فرعية داخل المجلد Shared الموجود في مجلد Views ولنسمها مثلا Notification.cshtml كما يلي:



الصورة 51 - إضافة صفحة جديدة

لتكون بذلك بنية المشروع كما يلي:



الصورة 52 - بنية المشروع

الآن لندخل إلى الصفحة الفرعية ونكتب فيها الأوامر التالية:

Shared/Notification.cshtml:

```
<div class="alert alert-info">
    Hello, this alert will be displayed from the partial view: Notification!
</div>
```

تجدر الإشارة إلى أنه ممكن أن نربط الصفحات الفرعية Partial views مع Model معين، وبالتالي نقوم بعرض محتوى هذا الأخير بالشكل الذي نريد مع إمكانية استدعاء الصفحة الفرعية في صفحة رئيسية ترسل نفس نوع Model على شكل برامتر.

الآن سنقوم باستدعاء هذه الصفحة الفرعية في إحدى صفحاتنا، لعمل ذلك نكتب ما يلي:

Index.cshtml:

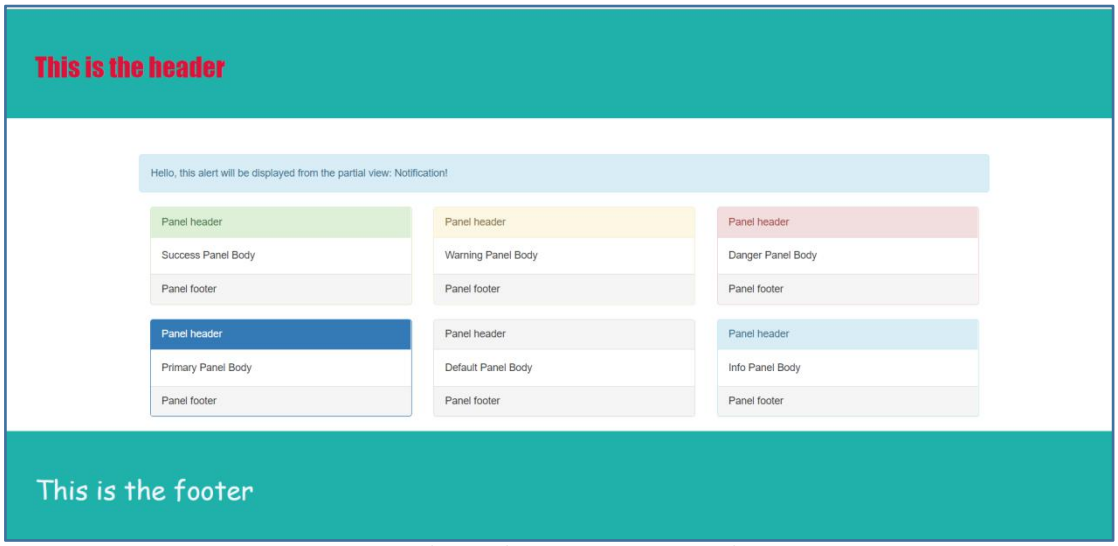
```
@Html.Partial("Notification")
```

كما يمكننا الاستفادة من Tag helpers عبر استعمال الوسم partial tag helper كما يلي:

Index.cshtml:

```
<partial name="Notification"/>
```

عند التنفيذ سنحصل على نفس النتيجة في الحالتين معا:



الصورة 53 - استعراض الصفحة الفرعية

إذا كانت Partial Views لا تكفي، فلك في View Components غنية

تعد View Component من بين الإضافات الجديدة التي عرفتها ASP.NET Core والتي تؤدي عملا مماثلا للصفحات الفرعية مع اختلاف طفيف، حيث أن هذه الأخيرة إذا كانت متصلة ب Model معين فلابد أن يكون من نفس نوع Model الذي تستعمله الصفحة الرئيسية Parent View، بينما مفهوم View Component يسمح لنا ببناء صفحات متصلة ب Model مختلف، مما يسمح لها أن تعمل بشكل مستقل عن الصفحة التي تستدعيها.

إذا أردنا أن ننشئ ViewComponent فالطريقة سهلة، يكفي أن نقوم بإنشاء كلاس ترث من الفئة الرئيسية ViewComponent ثم تحتوي على الوظيفة Invoke أو InvokeAsync إذا أردنا أن نشتغل في Asynchronous mode، ثم نضع داخل هذه الوظيفة التعليمات البرمجية التي نريد استعمالها في الصفحة.

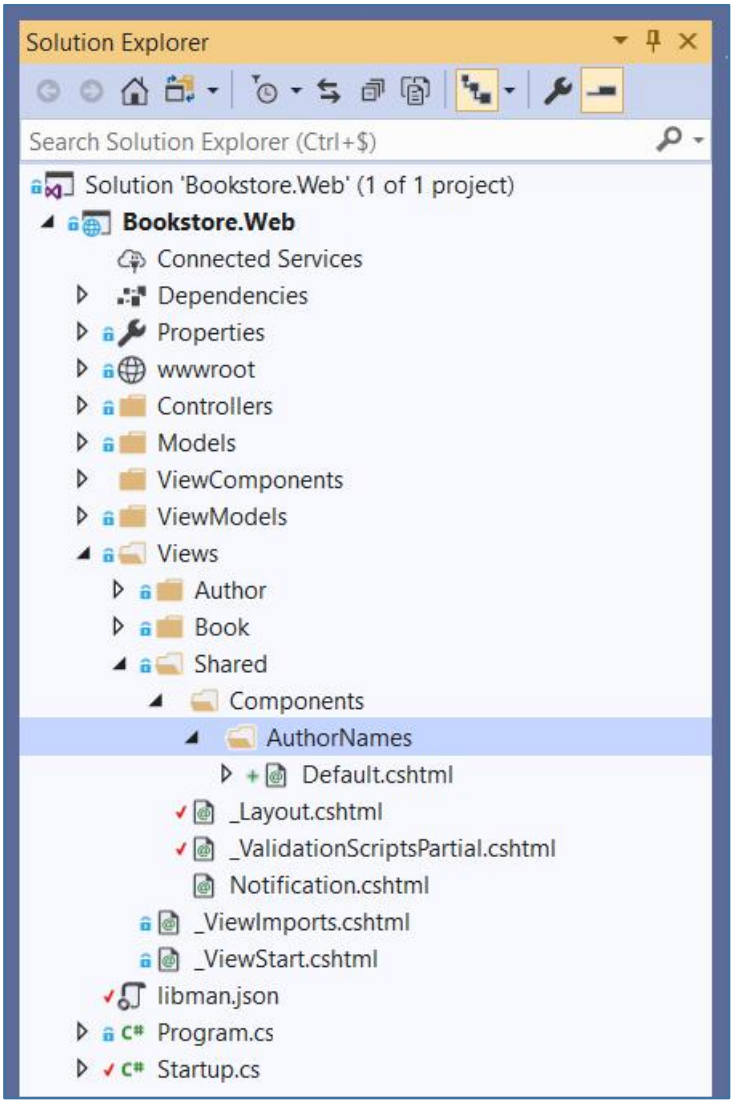
لشرح هذه الأمور بالتفصيل، تعالوا بنا ننجز مثالا بسيطا يرجع أسماء مجموعة من المؤلفين بغرض استعالهم في صفحة معينة، في الأول لنقم بإنشاء مجلد نسميه ViewComponents ولنضف إليه كلاس باسم AuthorNamesViewComponent ثم نضع فيه الكود التالي:

AuthorNamesViewComponent.cs:

```
using Bookstore.Web.Models;
using Bookstore.Web.Models.Repositories;
using Microsoft.AspNetCore.Mvc;
using System.Linq;

namespace Bookstore.Web.ViewComponents
{
    public class AuthorNamesViewComponent:ViewComponent
    {
        private readonly IRepository<Author> _repository;

        public AuthorNamesViewComponent(IRepository<Author> repository)
        {
            _repository = repository;
        }

        public IViewComponentResult Invoke()
        {
            var authorNames = _repository.GetAll()
                                         .Select(c => c.FullName)
                                         .ToList();

            return View(authorNames);
        }
    }
}
```

بالنسبة للكلاس الخاصة ب AuthorNamesViewComponent فقد استعملنا نسخة من IRepository<Author> باستعمال آلية dependency injection التي رأيناها سابقا.

الآن سنقوم بتصميم الصفحة اللازمة لعرض محتوى هذا ViewComponent، لندخل إلى مجلد Shared ونقم بإضافة مجلد فرعي نسميه ViewComponents وبداخل هذا الأخير نضيف مجلدا آخر باسم ViewComponent الخاص بنا والذي أسميناه AuthorNamesViewComponent ثم بداخل هذا المجلد الأخير نضيف صفحة جديدة باسم Default، لتكون بنية المشروع كما يلي:



الصورة 54 - بنية المجلد من جديد

لندخل إلى الصفحة Default.cshtml ونكتب فيها الأوامر الخاصة بعرض أسماء المؤلفين:

Shared/Components/AuthorNames/Default.cshtml:

```
@model IList<string>

<h2>Author Names</h2>
<ul class="list-group">
    @foreach (var name in Model)
    {
        <li class="list-group-item">@name</li>
    }
</ul>
```

ثم في الختام ندخل إلى الصفحة التي نريد عرض محتوى ViewComponent عليها ونكتب الأمر التالي:

Index.cshtml:

```
@await Component.InvokeAsync("AuthorNames")
```

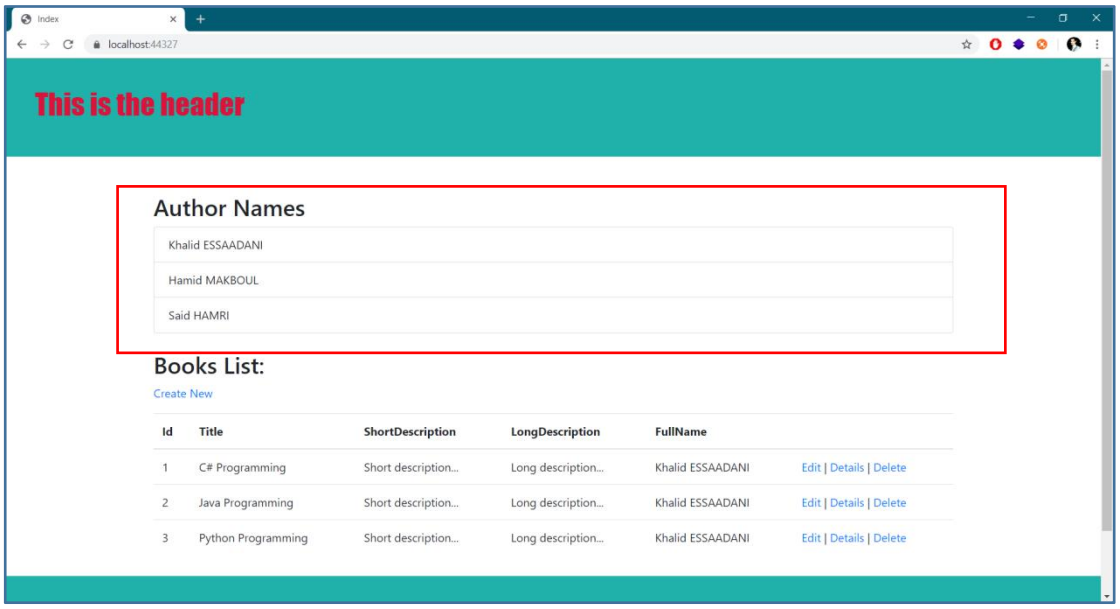
أو نكتبه باستعمال Tag helper كما يلي:

Index.cshtml:

```
<vc:author-names></vc:author-names>
```

حيث vc اختصار ل View Component.

وفي الحالتين معا، سوف نحصل على نفس النتيجة:



الصورة 55 - استعراض ViewComponent

شرطة البيانات على مستوى السيرفر المعروفة ب Model Validation

تسمح لنا تقنية ASP.NET Core بالتحقق من سلامة البيانات التي تدخل إلى التطبيق من خلال آلية Model Validation، حيث يمكننا أن نضيف بعض القيود على خصائص Models من قبيل أن يكون الحقل إلزاميا، أو أن يكون طوله الأقصى بعدد حروف معين، أو أن يكون موافقا ل pattern معين مثل الإيميل، رقم الهاتف، رابط موقع وغير ذلك.

يمكننا الاستفادة من هذه الآلية عبر جلب مجال الأسماء System.ComponentModel.DataAnnotations، والتي تأتي معها بمجموعة من الواصفات Attributes والتي تؤدي المطلوب منها.

في هذا الجدول نستعرض معا بعض هذه الواصفات التي تسمح لنا بتطبيق آلية Model validation:

الواصفة	دورها
Required	تعني أن قيمة الحقل إجبارية ولا يسمح بإرساله فارغا.
Compare	يسمح بمقارنة حقلين من حقولModel
MaxLength	يسمح بتحديد العدد الأقصى لحروف السلسلة النصية لحقل معين
MinLength	يسمح بتحديد العدد الأدنى لحروف السلسلة النصية لحقل معين
StringLength	يسمح بتحديد الحد الأقصى والأدنى لسلسلة نصية
Range	تسمح بتحديد مجال رقمي ينبغي أن تكون قيمة الحق داخله
RegularExpression	يعني أن قيمة الحق ينبغي أن تكون مطابقة لل Regular Expressionالموضوعة في الخاصية.
CustomValidation	تسمح لنا بإنشاء طريقة خاصة بنا للتحقق من سلامة قيمة حقل معين
EmailAddress	تعني أن قيمة الحقل ينبغي أن تكون على شكل بريد الكتروني
CreditCard	تعني أن قيمة الحقل ينبغي أن تكون على شكل بطاقة ائتمان
Url	تعني أن قيمة الحقل ينبغي أن تكون على شكل رابط
FileExtension	تتحقق من امتداد الملف من خلال قيمة الحقل

تعرف على Model Validation عن كثب

الآن تعالوا بنا نتعرف على مفهوم Model Validation بالتطبيق، من خلال العودة إلى الموديل BookAuthorViewModel والذي كنا قد أعطيناه البنية التالية:

BookAuthorViewModel.cs:

```
    public class BookAuthorViewModel
    {
        public int BookId { get; set; }
        public string Title { get; set; }
        public string LongDescription { get; set; }
        public string ShortDescription { get; set; }
        public int AuthorId { get; set; }

        public IList<Author> Authors { get; set; }
    }
```

سنجعل كافة الخصائص إلزامية، لعمل ذلك سنضيف الواصفة Required فوق كل خاصية من الخصائص، لكن قبل ذلك علينا جلب مجال الأسماء System.ComponentModel.DataAnnotations:

BookAuthorViewModel.cs:

```
using Bookstore.Web.Models;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Bookstore.Web.ViewModels
{
    public class BookAuthorViewModel
    {
        [Required]
        public int BookId { get; set; }

        [Required]
        [MinLength(4)]
        [MaxLength(25)]
        public string Title { get; set; }

        [Required]
        [MinLength(50)]
        [MaxLength(500)]
        public string LongDescription { get; set; }

        [Required]
        [MinLength(15)]
        [MaxLength(50)]
        public string ShortDescription { get; set; }

        [Required]
        public int AuthorId { get; set; }
```

```
        public IList<Author> Authors { get; set; }
    }
}
```

يمكننا أن نضيف بعض الواصفات الأخرى التي تحدد طول النص كما يلي:

```
        [Required]
        [MinLength(4)]
        [MaxLength(25)]
        public string Title { get; set; }
```

يمكننا الجمع بين MinLength و MaxLength في واصفة واحدة من خلال استعمال StringLength وذلك كما يلي:

```
        [StringLength(25,MinimumLength = 4)]
```

حيث 25 هي القيمة القصوى، و 4 هي القيمة الدنيا.

بعد إضافة الواصفات اللازمة للخصائص، يتبقى فقط أن نذهب إلى Action التي تستقبل البيانات عند الإضافة Create والتعديل Edit اللتان تعملان في وضع POST، أي حينما نرسل البيانات من الفورم إلى Action، ونضيف إليهما أمر التحقق من سلامة البيانات من خلال استعمال الخاصية IsValid التابعة لل ModelState كما يلي:

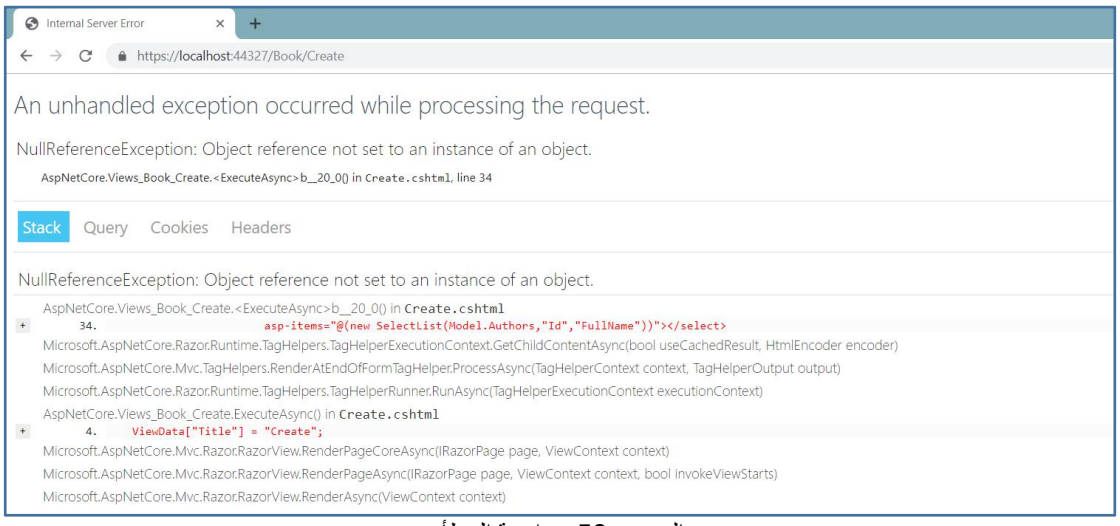
BookController.cs:

```
        [HttpPost]
        public ActionResult Create(BookAuthorViewModel viewModel)
        {
            if (ModelState.IsValid)
            {
                try
                {
                    var book = new Book
                    {
                        Title = viewModel.Title,
                        LongDescription = viewModel.LongDescription,
                        ShortDescription = viewModel.ShortDescription,
                        Author = new Author
                        {
                            Id = viewModel.AuthorId,
                            FullName =
AuthorRepository.GetById(viewModel.AuthorId).FullName
                        }
                    };

                    BookRepository.Add(book);
                    return RedirectToAction(nameof(Index));
                }
                catch (Exception ex) // catches all exceptions
                {
                    return View(ex.Message);
```

```
            }
        }
        ModelState.AddModelError("", "Error");
        return View();
    }
```

الكود أعلاه يتحقق من أن ModelState لا تحتوي على أية Errors وبالتالي قيمة IsValid هي true، إن كان كذلك سيقوم بعملية الإضافة بالشكل الاعتيادي، وإن فشلت عملية التحقق من سلامة بيانات ViewModel فإن ModelState ستقوم بعرض الأخطاء الناتجة.

عند التنفيذ وترك الفورم فارغا والضغط على الزر Create ستلاحظ أنه حصل الخطأ التالي:



الصورة 56 - واجهة الخطأ

هذا الخطأ مفاده أن القائمة المنسدلة الخاصة بعرض أسماء المؤلفين تؤشر إلى object وفي حالتنا هذه هو Authors دون أن تكون له قيمة، وحل هذا الخطأ باختصار يكون عبر إعادة تعبئة هذا Object عند عمل POST للفورم حيث نضيف نفس الكود الذي يعمل في Create التي تشتغل في وضع GET في آخر Create التي تعمل في وضع POST كما يلي:

BookController.cs:

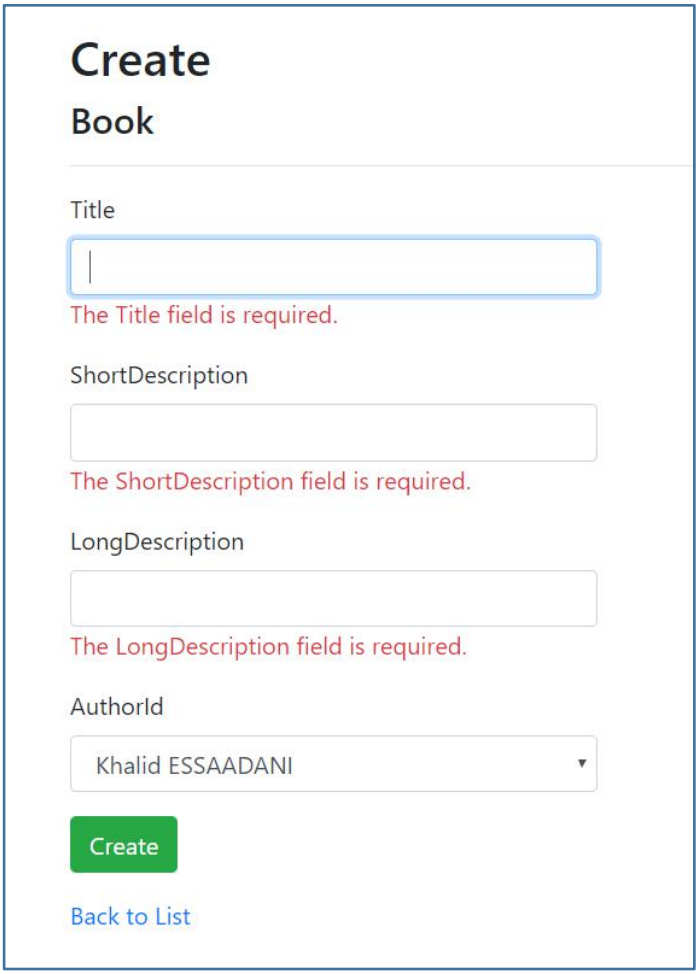
```
        BookAuthorViewModel bookAuthorViewModel = new BookAuthorViewModel
        {
            Authors = AuthorRepository.GetAll()
        };

        return View(bookAuthorViewModel);
```

ليكون بذلك الكود الكامل لل Create Post action كما يلي:

BookController.cs:

```
        [HttpPost]
        public ActionResult Create(BookAuthorViewModel viewModel)
        {
            if (ModelState.IsValid)
            {
                try
                {
                    var book = new Book
                    {
                        Title = viewModel.Title,
                        LongDescription = viewModel.LongDescription,
                        ShortDescription = viewModel.ShortDescription,
                        Author = new Author
                        {
                            Id = viewModel.AuthorId,
                            FullName =
AuthorRepository.GetById(viewModel.AuthorId).FullName
                        }
                    };

                    BookRepository.Add(book);
                    return RedirectToAction(nameof(Index));
                }
                catch (Exception ex) // catches all exceptions
                {
                    return View(ex.Message);
                }
            }

            ModelState.AddModelError("", "Cannot add this book, please check the
required fields!");

            BookAuthorViewModel bookAuthorViewModel = new BookAuthorViewModel
            {
                Authors = AuthorRepository.GetAll()
            };

            return View(bookAuthorViewModel);
        }
```

الآن لو نفذنا وضغطنا على زر Create دون إعطاء قيم للحقول الإلزامية أو عدم احترام الواصفات التي وضعناها للخصائص في View Model، فإننا سنحصل على الأخطاء التالية:



الصورة 57 - ظهور رسائل التحقق من سلامة المدخلات

الآن ربما تتساءل كيف تم عرض الأخطاء في الصفحة؟ الجواب: هو تلك Tag helpers التي كنا قد أجلنا شرحها من قبيل asp-validation-for و asp-validation-summary الموجودة في View، هي من تقوم بعرض الأخطاء بحيث كل وسم يتكلف بخاصية معينة، لذلك ستجد أسفل كل حقل وسم خاص بعرض الأخطاء الناتجة كما يلي:

Book/Create.cshtml:

```
    <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ShortDescription"
class="control-label"></label>
                <input asp-for="ShortDescription" class="form-control" />
                <span asp-validation-for="ShortDescription"
class="text-danger"></span>
            </div>
            <div class="form-group">
```

```
            <label asp-for="LongDescription"
class="control-label"></label>
            <input asp-for="LongDescription" class="form-control" />
            <span asp-validation-for="LongDescription"
class="text-danger"></span>
        </div>
```

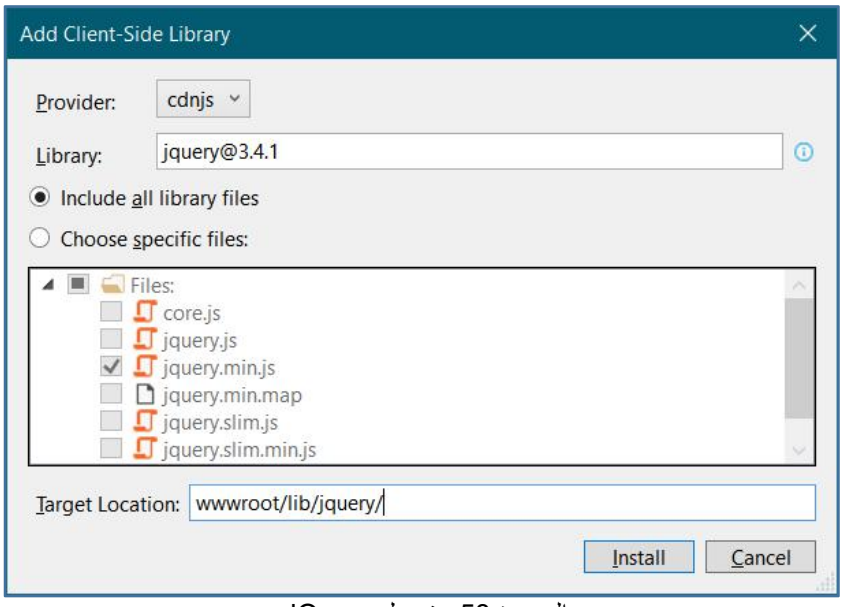
هكذا نكون قد تعرفنا بشكل مبسط على كيفية القيام بالتحقق من سلامة البيانات على مستوى السيرفر باستعمال Data annotations.

شرطة البيانات على مستوى المتصفح Client Validation وكيفية تفعيلها

في بعض الأحيان تكون عملية التحقق من البيانات المدخلة بسيطة ولا تستلزم أن نذهب إلى السيرفر لأجل ذلك، مثل التحقق هل قيمة حقل معين فارغة، هل طول قيمة معينة بقدر كذا، هل التاريخ المدخل صحيح وما إلى ذلك.

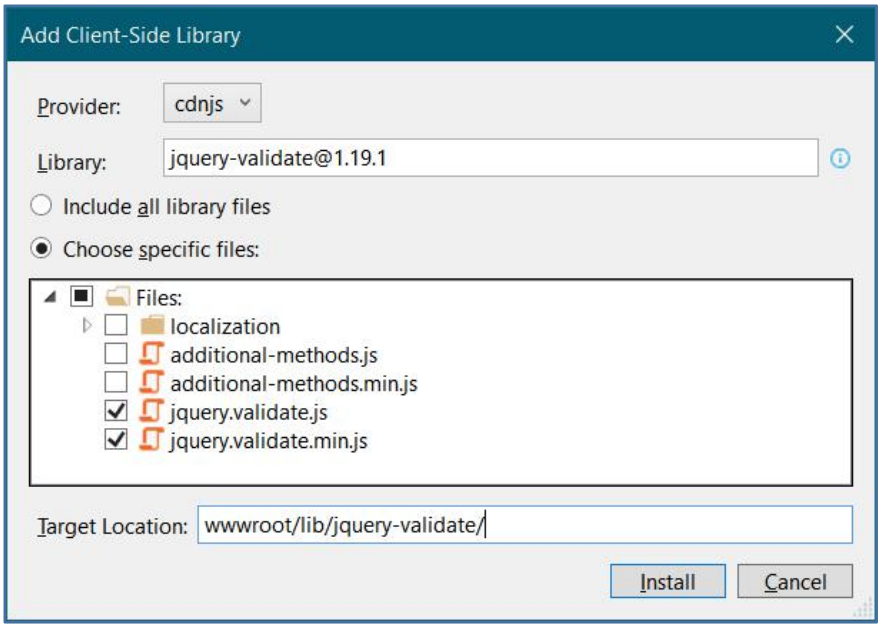
هنا نحتاج إلى القيام بعملية التحقق من المدخلات على متصفح العميل، يمكننا أن ننشئ سكريبت خاص بنا بواسطة جافاسكريبت يقوم بذلك، كما بإمكاننا استعمال بعض المكتبات الجاهزة التي تؤدي هذا الغرض مثل jquery validation و jquery unobtrusive، في الأول علينا تحميل jQuery، ثم jquery-validate، ثم jquery-validation-unobtrusive إما يدويا أو باستعمال LibMan كما يلي:

نقوم بتحميل jQuery:



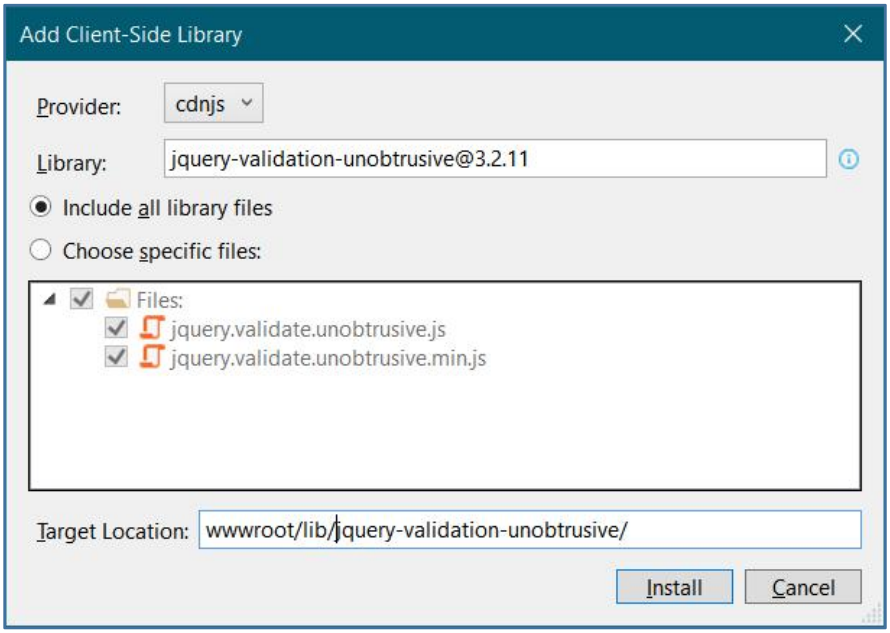
الصورة 58 - تحميل jQuery

ثم نقوم بتحميل jquery-validate كما يلي:



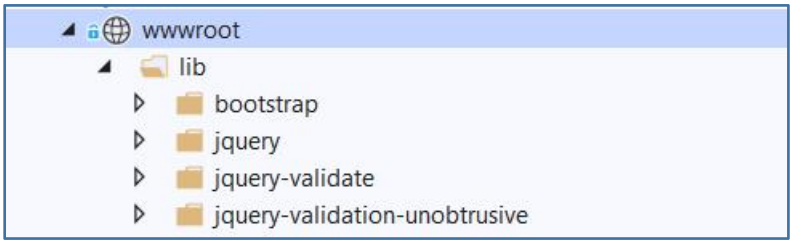
الصورة 59 - تحميل jQuery-Validate

ثم نقوم بتحميل jquery-validation-unobtrusive :



الصورة 60 - تحميل jQuery-Validation-Unobtrusive

ليصير مجلد wwwroot كما يلي:



الصورة 61 - بنية مجلد wwwroot

الآن سنقوم بإضافة المكتبات إلى الصفحات التي نحتاج فيها إلى تفعيل التحقق من جهة العميل Client-side validation، ولتكن مثلا صفحة Create.cshtml، لننزل إلى أسفل ونستدعي السكريبتات الثلاثة على التوالي كما يلي:

Book/Create.cshtml:

```
        <script src="~/lib/jquery/jquery.js"></script>
        <script src="~/lib/jquery-validate/jquery.validate.js"></script>
        <script
src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></
script>
```

الآن عند التنفيذ ستلاحظ أن عملية التحقق باتت تحدث على مستوى العميل دون الحاجة إلى الذهاب إلى السيرفر.

يمكننا جمع هذه الملفات الثلاثة ووضعها في صفحة فرعية مستقلة ثم نستدعيها عند الحاجة، أو نقوم بوضعها في صفحة رئيسية من نوع Layout.

تعرف على Routes الطرق السيارة التي توجه HTTP Requests

عملية التبويب Routing مهمة جدا في تطبيقات الويب بشكل عام، لأنها تسمح بتحديد الوظيفة المراد تنفيذها، وفيASP.NET Core يمر Request من عملية التبويب من أجل الحصول على الرابط URI وعلى HTTP Verb لكي يتم البحث في جدول التبويبات Route tables عن التبويب المناسب، ومن ثم يتم تحديد الوظيفة المتوافقة وتنفيذها.

يمكننا التحكم في التبويب الافتراضي وإضافة تبويبات جديدة من خلال الذهاب إلى الكلاس Startup.cs وإضافة الأمر التالي:

Startup.cs:

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }

            app.UseStaticFiles();
            app.UseStatusCodePages();

             app.UseMvc(routes =>
             {
                 routes.MapRoute("default",
                                 "{controller=Book}/{action=Index}/{id?}")
                             ;
             });

        }
```

ستلاحظ أننا أنشأنا تبويبا افتراضيا أسميناه default، ثم وضعنا بعد ذلك Template التي تحدد شكل التبويب مع إضافة قيم افتراضية ليتم الدخول إليها عند تنفيذ التطبيق.

بمعنى أن التبويب يفرض على الروابط التي ستدخل إلى ASP.NET Pipeline أن تكون على الشكل التالي:

```
Default route template:

http://www.WebSiteName.com/ControllerName/ActionName/OptionalId
```

حيث www.WebSiteName.com هو اسم الموقع، و ControllerName هو اسم Controller المراد الدخول إليها، وActionName هو اسم Action المراد تنفيذها، و OptionalId هو برامتر اختياري يمكن أن نرسله في Request أو لا لذلك وضعنا بعده في Template رمز علامة الاستفهام.

باستعمال route السابق يمكننا استدعاء الوظائف بهذا الشكل على سبيل المثال:

```
Default route examples:

http://www.WebSiteName.com/Book/Details/2

http://www.WebSiteName.com/Book/Create

http://www.WebSiteName.com/Book
```

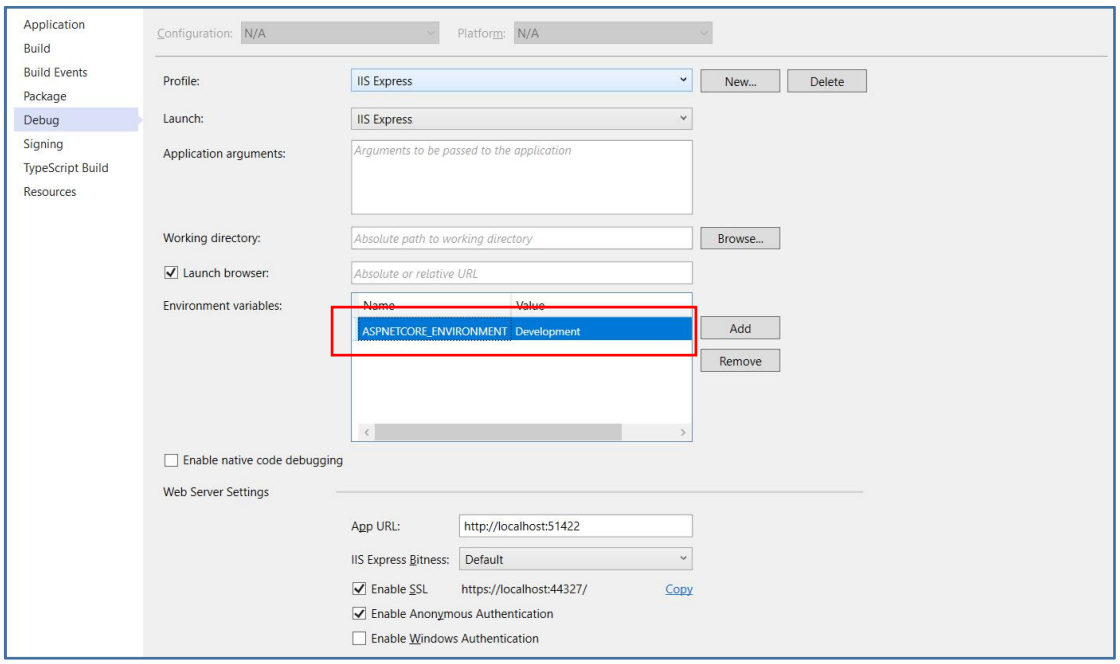
كما ذكرنا، فحينما يصل Request معين إلى التطبيق، سيتم استخراج HTTP Verb منه لمعرفة هل هو أمر GET أو POST أو غيره، ثم رابط Resource المراد التعامل معها أي URI، فيقوم نظام التبويب بتحديد controller و action المناسبين للطلب القادم، فيتم تنفيذهما، وفي حال لم يجد أي Resource ب URI القادم، يعيد للعميل الخطأ 404 والذي يفيد بأنResource المراد غير موجود.

يمكننا تغيير شكل التبويب الافتراضي، كما يمكننا أيضا إضافة تبويبات أخرى حسب حاجتنا لها.

شرح Environment حيث نضع أقدامنا وحيث سنضعها!

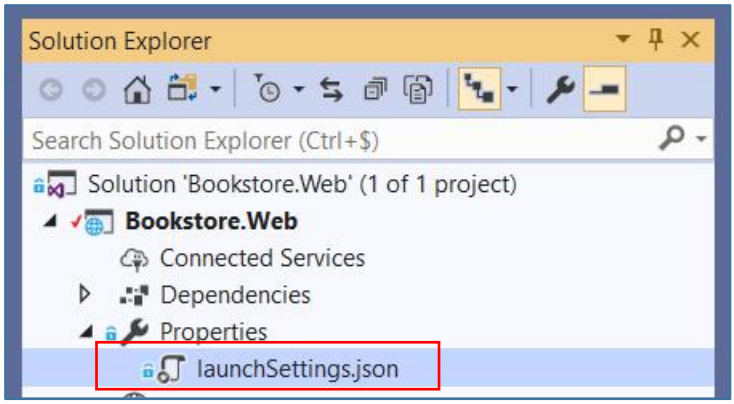
جرت العادة أن يمر التطبيق من عدة بيئات قبل أن يتم توزيعه ونشره ليتم استعماله من قبل العميل، هذه البيئات تبدأ ببيئة التطوير development environment ثم نمر بعدها إلى staging environment وفي الختام نصل إلى Production environment، وممكن أن نحتاج إلى بيئات أخرى حسب طبيعة المشروع، لكن في المجمل هذه هي البيئات الثلاثة التي ينبغي أن يمر منها المشروع قبل توزيعه ونشره، والغرض الأساسي من هذه المراحل هو جعل التطبيق يتعامل مع بعض التفاصيل البرمجية بشكل مختلف، فمثلا في مرحلة التطوير development phase نحتاج إلى مشاهدة الأخطاء البرمجية وعرضها من أجل التعامل معها، لكن في production phase لا ينبغي أن نظهر الأخطاء والاستثناءات البرمجية، أو أن نقوم مثلا باستعمال بعض الملفات في بيئة التطوير وملفات أخرى في بيئة الإنتاج مثل ملفات جافاسكريبت و css، كما يمكننا أيضا أن نجعل كود معين يعمل في بيئة معينة دون غيرها وهكذا دواليك حسب ما تقتضيه حاجتنا البرمجية.

للتحكم في البيئة التي نريد اعتمادها في فترات التطوير، تقدم لنا ASP.NET Core متغيرا يسمى ASPNETCORE_ENVIRONMENT والذي نستطيع تغيير قيمته حسب البيئة التي نريد، وذلك من خلال الدخول إلى خصائص المشروع، والانتقال إلى الأمر Debug كما يلي:



الصورة 62 - واجهة Debug

كما تلاحظ فإن القيمة الافتراضية للمتغير ASPNETCORE_ENVIRONMENT هي Development فيما يعني أن البيئة الحالية هي بيئة التطوير، يمكننا تغيير قيمة هذا المتغير من هنا لتحديد بيئة التطوير الحالية، كما يمكننا أيضا أن نقوم بنفس الأمر من خلال الدخول إلى ملف launchSettings.json الموجود أسفل Properties من Solution Explorer:



الصورة 63 - ملف launchSettings.json

إذا دخلنا إلى الملف launchSettings.json سنجد محتواه على الشكل التالي:

launchSettings.json:

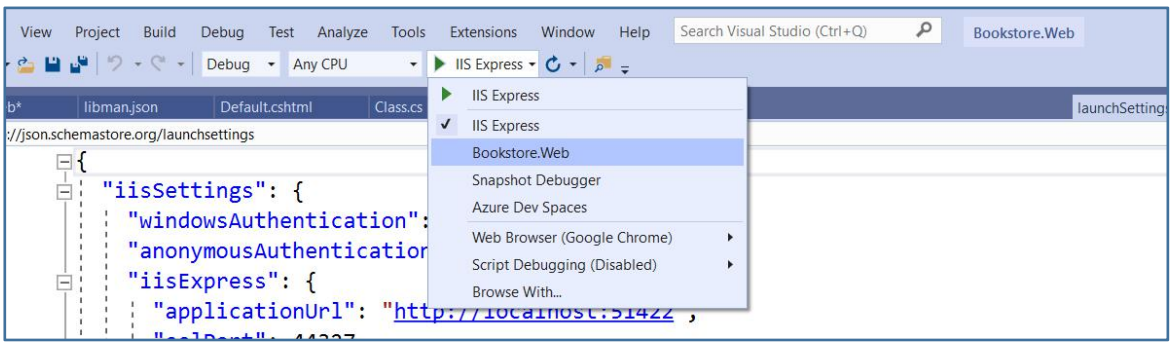
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
```

```
    "iisExpress": {
      "applicationUrl": "http://localhost:60339",
      "sslPort": 44379
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Bookstore.Web": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    }
  }
}
```

ستلاحظ أن ملف launchSettings.json يحتوي على عدة بروفايلات profiles تسمى debug profiles وكل واحدة منها لها إعداداتها، حيث في الجزء الأول يتم تحديد إعدادات IIS، ثم بعد ذاك نجد البروفايلات المستعملة، وفي حالتنا الافتراضية يوجد اثنان وهما IIS Express، والبروفايل الخاص بالمشروع الحالي والذي في حالتنا اسمه Bookstore.Web وهو اسم المشروع الذي نشتغل عليه والذي يعتمد على الويب سيرفر الداخلي المعروف ب Kestrel internal web server.

ستلاحظ أيضا أن كل بروفايل يتوفر على environment variable وفي الحالتين معا القيمة الافتراضية هي development مما يعني أن البيئة الحالية لتنفيذ المشروع هي بيئة التطوير.

تلقائيا يتم تنفيذ المشروع على البروفايل IIS Express، ويمكننا اختيار البروفايل الذي نريد من خلال زر التنفيذ كما يلي:



الصورة 64 - تغيير البروفايل عند التنفيذ

يمكننا التحكم في المشروع على حسب البيئة التي يشتغل عليها، وذلك من خلال الوظيفة Configure الموجودة في ملف Startup.cs، وهذا مثال يبين ذلك:

Startup.cs:

```
        public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {
            if (env.IsDevelopment())
            {
                // This code will be executed in Development Enviroment
            }

            if (env.IsStaging())
            {
                // This code will be executed in Staging Enviroment
            }

            if (env.IsProduction())
            {
                // This code will be executed in Production Enviroment
            }
        }
```

كما يمكننا أيضا التعامل مع environment variable من خلال tag helper يسمح لنا بتحديد البيئة ومن ثم سيقوم بتحميل ملفات معينة أو تنفيذ أوامر معينة حسب هذه البيئة كما يبين المثال التالي:

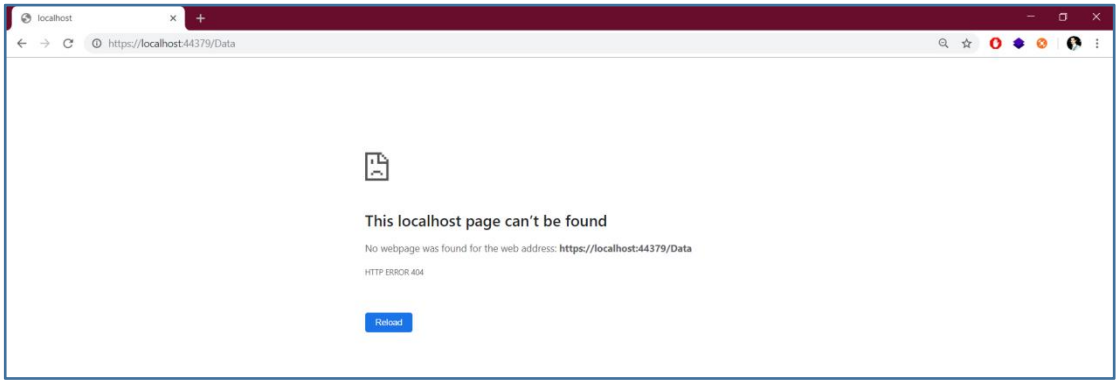
Shared/_Layout.cshtml:

```
<environment names="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment names="Staging,Production">
    <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/css/bootstrap.min.css
"
          asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          asp-fallback-test-class="sr-only"
asp-fallback-test-property="position" asp-fallback-test-value="absolute" />
    <link rel="stylesheet" href="~/css/site.min.css"
asp-append-version="true" />
</environment>
```

في المثال أعلاه فإن environment tag helper سيقوم بتحميل الملفات حسب بيئة التنفيذ، وهذه خاصية مهمة جدا لتدبير مهمة تحميل الملفات إلى ASP.NET Core حسب بيئة التنفيذ الحالية بدل القيام بذلك بأنفسنا.

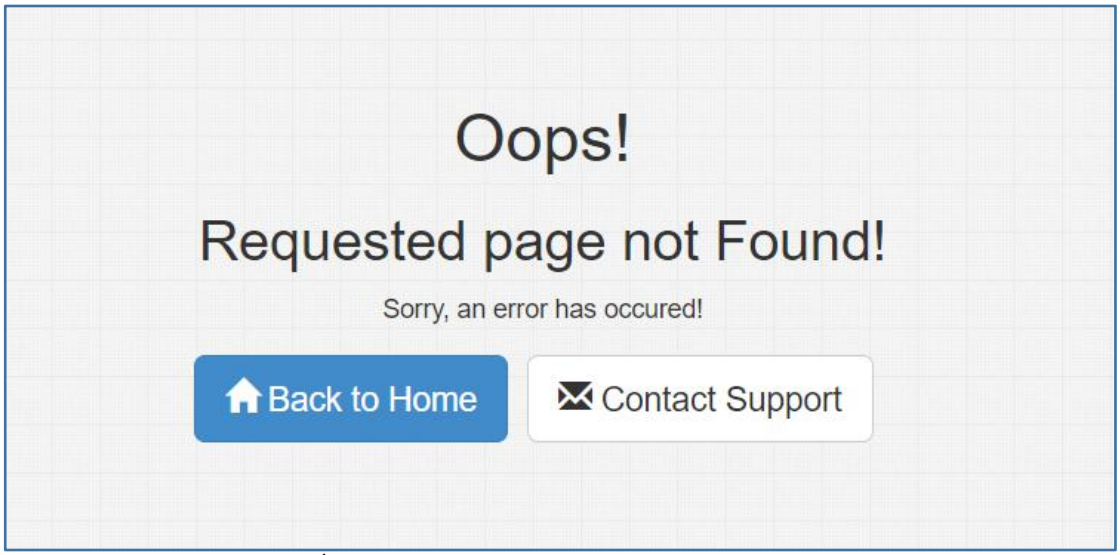
صديقك الوفي في عرض تفاصيل الاستثناءات Exceptions

تعرفنا في الفصل الماضي على مفهوم environment وذكرنا بأنها تسمح لنا بالتحكم في بيئة التنفيذ الحالية، وذكرنا بأن كل بيئة تنفيذ لها دواعي استعمالها، مثلا بيئة التطوير ينبغي أن تعرض لنا الخطأ في حال حدوثه بالتفصيل لنتمكن من التعرف على سببه ومن ثم حله، أما في مرحلة الانتاج فلا ينبغي أن نعرض الأخطاء البرمجية، وإنما ينبغي أن نعرض رسائل عامة للمستخدم دون إقحامه في التفاصيل، مثلا بدل أن نعرض له الصفحة التالية في حال عدم الحصول على Resource التي يريدها:



الصورة 65 - الصفحة الافتراضية عند عدم العثور على Resource ما

يمكننا أن نعرض له هذه الصفحة وهي معبرة عن عدم العثور على الملف المطلوب:



الصورة 66 - واجهة عدم العثور على Resource بتصميم أفضل

في هذا المثال سنتعرف على كيفية عرض معطيات عن الخطأ الحاصل في حال الاشتغال على بيئة التطوير، وأن نظهر صفحة تعرض خطأ عاما في حال الاشتغال في بيئة الإنتاج أو غيرها، تعالوا بنا نكتب الكود التالي في الوظيفة Configure داخل Startup.cs:

Startup.cs:

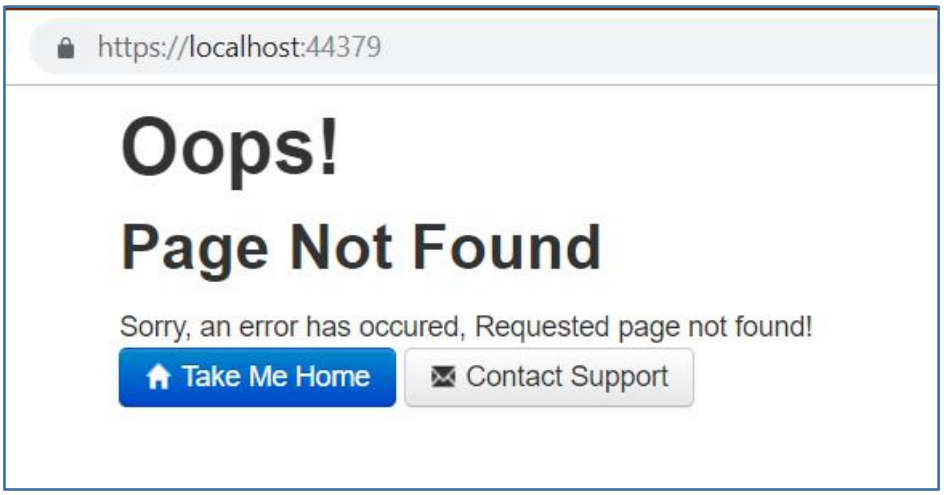
```
        public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
        {

            if (env.IsDevelopment())
            {
                app.UseStatusCodePages();
            }
            else
            {
                app.UseStatusCodePagesWithReExecute("/Shared/Error");
            }

            app.UseMvcWithDefaultRoute();
        }
```

في الكود أعلاه سيتم عرض رقم الخطأ في بيئة التطوير وهو شيء مفهوم بالنسبة للمطور، لكن في باقي البيئات سيتم عرض خطأ عام، إذا غيرنا قيمة المتغير ASPNETCORE_ENVIRONMENT إلى Production مثلا:

launchSettings.json:

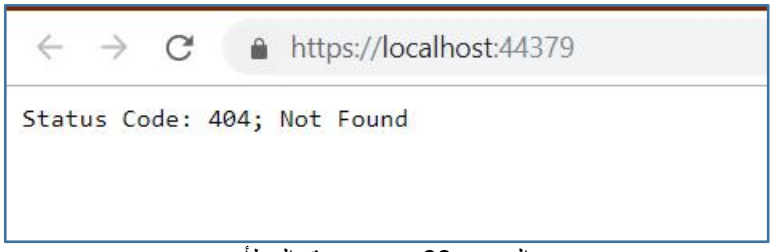
```
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
```

عند تنفيذ التطبيق ومحاولة الوصول إلى صفحة غير موجودة في المشروع ستظهر لنا الصفحة التالية:



الصورة 67 - تغيير الواجهة الافتراضية لعدم العثور على Resource

لما نرجع بيئة التطوير إلى Development سنحصل على الصفحة التالية:



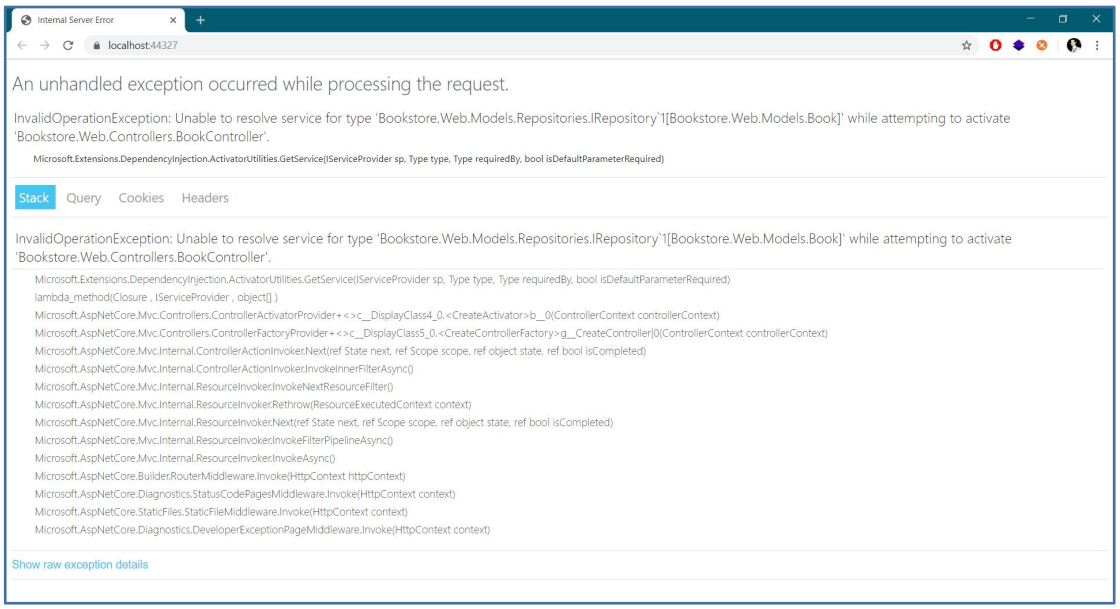
الصورة 68 - عرض رقم الخطأ

في بعض الأحيان، تحدث أخطاء برمجية نحتاج معها إلى المزيد من التفاصيل التقنية، لعمل ذلك، يمكننا استدعاء Middleware الذي يقوم بعرض صفحة استثناءات التطوير Developer Exception Page وذلك من خلال إضافة الأمر التالي داخل الوظيفة Configure:

Startup.cs:

```
if (env.IsDevelopment())
{
    app.UseStatusCodePages();
    app.UseDeveloperExceptionPage();
}
```

عند حدوث خطأ برمجي سنجد الصفحة التالية تخبرنا بتفاصيله:



الصورة 69 - عرض المزيد من التفاصيل عن الاستثناءات

وهنا بكل وضوح يظهر الخطأ وتفاصيله، في الخطأ أعلاه:

InvalidOperationException: Unable to resolve service for type [Bookstore.Web.Models.Book]' while attempting to 1'Bookstore.Web.Models.Repositories.IRepository` activate 'Bookstore.Web.Controllers.BookController'.

حدث استثناء من نوع InvalidOperationException يفيد بأننا لم نقم بتسجيل IRepository في IoC container لكي نتمكن من إنشاء نسخ منه، وبالتالي يلزمنا فقط الذهاب إلى الوظيفة ConfigureServices وتسجيل service كما يلي:

Startup.cs:

```
services.AddSingleton<IRepository<Book>, BookRepository>();
```

وبالتالي يتم حل المشكل.

هكذا نكون قد تعرفنا على كيفية التعامل مع الاستثناءات في ASP.NET Core.

محطات المعالجة المتقدمة Middlewares

من بين المزايا القوية في ASP.NET Core نجد مفهوم Midlleware وهي عبارة عن مكونات تشتغل مع كل HTTP request تدخل إلى ASP.NET request pipeline، وبالتالي يمكننا أن ننشئ middleware خاص بنا يتعامل مع هاته HTTP request.

تأتي ASP.NET Core بمجموعة من Middlewares الجاهزة التي تؤدي عدة مهام، وقد رأينا في الحصة الماضية كيف يسمح لنا أحد هذه Middlewares بعرض Developer Exception Page حينما تؤدي HTTP request إلى حدوث Exception معينة، هنالك بعض Middlewares الأخرى من قبيل:

- Authentication: والذي يسمح لنا بإضافة عملية التحقق من هوية المستخدم
- HTTPS redirections: تسمح لنا بتحويل HTTP request إلى الوضع الآمن HTTPS
- MVC: وتسمح لنا بمعالجة requests بواسطة MVC و Razor
- Routing: تسمح لنا بتبويب Requests من خلال تحديد شكل route الذي ينبغي لل URL أن يوافقه.
- Static files: تسمح لنا بالتعامل مع الملفات الثابتة مثل جافاسكريبت و css والصور، والتي ينبغي أن تكون إلزاميا في مجلد wwwroot، وسنتعرف أكثر على static files في الدرس اللاحق إن شاء الله.

توجد أيضا عدة Middlewares أخرى يقصر المقام عن ذكرها كلها، كما يمكننا أن ننشئ Middlewares خاصة بنا حسب حاجتنا، وهذا ما سنراه في المثال العملي، لكن قبل ذلك ينبغي أن نعلم أنه للتعامل مع Middlewares فالواجهة التي تسمح لنا بذلك هي IApplicationBuilder.
لكن قبل ذلك، يجب أن تكون على دراية تامة بمفهوم المفوضات delegates و عبارات لامدا Lambda Expressions، لأن Middlewares تعتمد على هذا الأسلوب من أجل كتابتها، كما يبين المثال التالي:

Startup.cs:

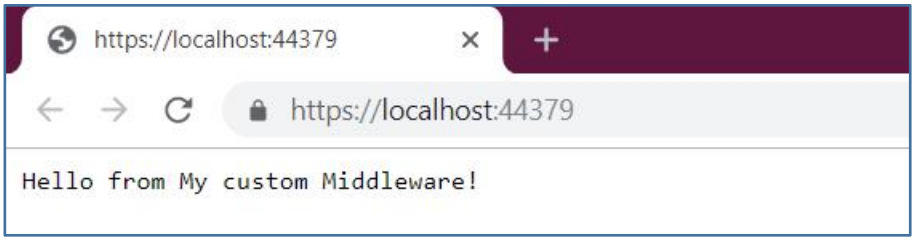
```
        private RequestDelegate HellowWorldMiddleware(RequestDelegate arg)
        {
            return async context =>
            {
                await context.Response.WriteAsync("Hello from My custom
Middleware!");
            };
        }
```

جعلنا الوظيفة HellowWorldMiddleware من نوع RequestDelegate، ثم بداخلها قمنا بكتابة Response التي نريد في HTTP context، الآن سنعود إلى الوظيفة Configure من أجل استدعاء Middleware الخاص بنا، وذلك على الشكل التالي:

Startup.cs:

```
            app.Use(HellowWorldMiddleware);
```

عند التنفيذ سنحصل على النتيجة التالية:



الصورة 70 - تنفيذ Custom Middleware

مما يعني أن Middleware الخاص بنا دخل إلى ASP.NET Request pipeline واستقبل request ثم نفذ الكود الذي نريد.

عودة سريعة إلى الملفات الثابتة Static Files

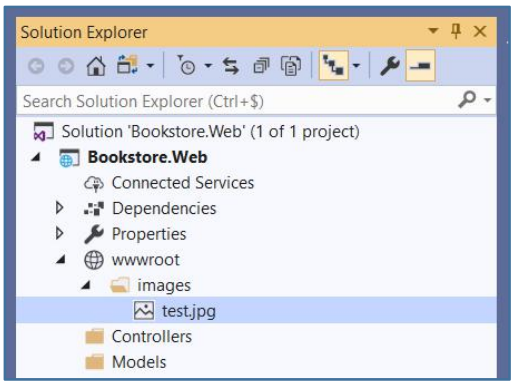
رأينا في الفقرة الماضية مفهوم Middlewares والتي تسمح لنا بالقيام بعمليات معينة على HTTP request الداخلة إلى pipeline الخاصة بمشروعنا، وكان من ضمن Middlewares التي ذكرنا واحدا يقوم بتمكيننا من التعامل مع الملفات الثابتة static files، حيث أنه لا يمكننا القيام بذلك دون استدعاء هذا Middleware في Configure method، لذلك تعالوا بنا نضيف السطر التالي داخل هذه الوظيفة:

Startup.cs:

```
app.UseStaticFiles();
```

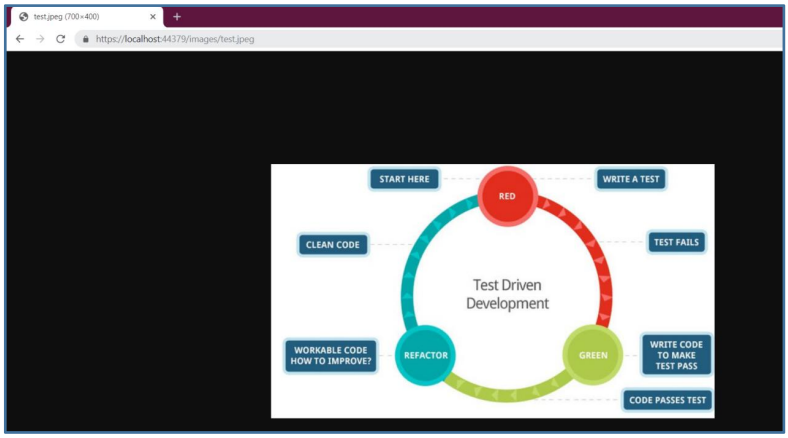
هكذا نكون قد قمنا بتفعيل هذا Middleware وبالتالي بإمكاننا التعامل مع الملفات الثابتة من قبيل جافسكريبت، صور، ملفات css وغيرها.

الآن لو دخلنا إلى مجلد wwwroot وأضفنا ملفا ثابتا، وليكن مثلا صورة:



الصورة 71 - مجلد wwwroot

فبإمكاننا طلبه وعرضه كما يلي:

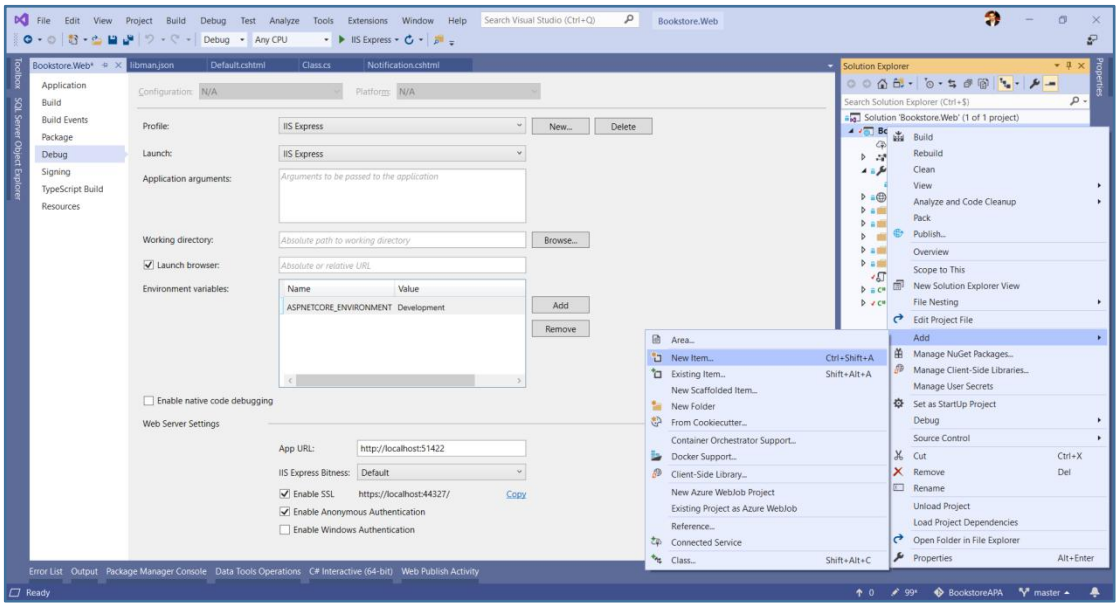


الصورة 72 - عرض الملف الثابت

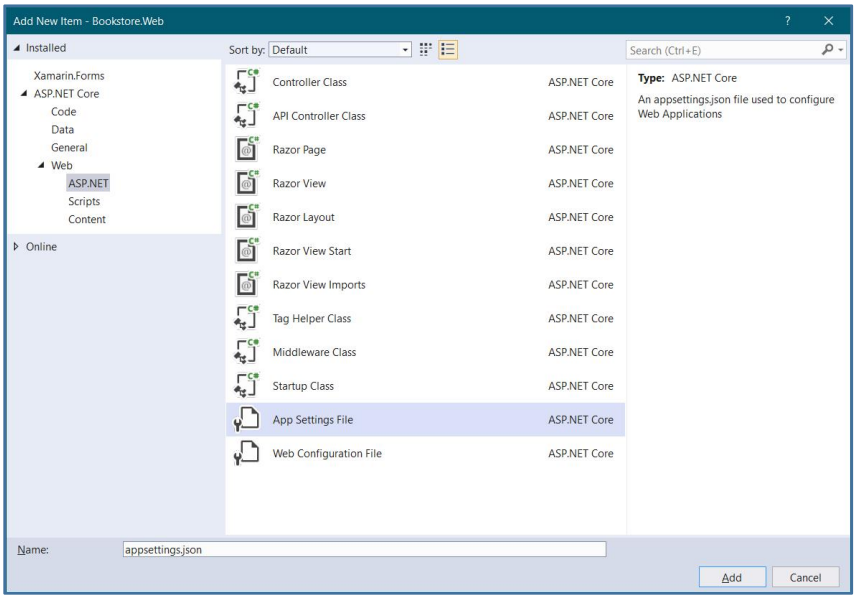
هكذا نكون قد تعرفنا على كيفية تفعيل Middleware الخاص بالملفات الثابتة، واستعمالها في تطبيقنا.

الوصول إلى البيانات المخزنة في ملف الإعدادات AppSettings.json

أثناء بناء مشاريعنا، سنحتاج إلى القيام ببعض الإعدادات في ملف appSettings.json، أو تخزين بعض القيم عليه من أجل الوصول إليها من كلاس معين، كأن نقوم بتخزين نص الاتصال بقاعدة بيانات من نوع SQL Server ثم استدعائها في كلاس معين، وما إلى ذلك، وهذا الملف قد تجده مسبقا في مشروعك، وإلا فإنه بإمكانك إضافته يدويا من خلال الضغط بيمين الماوس على اسم المشروع ثم اختيار Add new item ومن القائمة قم باختيار الملف كما يلي:



الصورة 73 - إضافة ملف الإعدادات appSettings.json



الصورة 74 - اختيار ملف الإعدادات

يمكننا التعامل بكل سهولة مع ملف appSettings.json بواسطة الواجهة IConfiguration التي تمكننا من الوصول إلى كل عنصر من عناصر هذا الملف من خلال اسم العنصر، لنفترض مثلا أننا أضفنا الخاصية WelcomeMessage كما يلي:

appSettings.json:

```
{
  "WelcomeMessage": "Welcome to ASP.NET Core Course"
}
```

الآن تعالوا بنا ندخل إلى Controller ثم نعمل Injection ل object من نوع IConfiguration بالاعتماد على DI Built-in System الذي شرحناه سابقا:

BookController.cs:

```
        private readonly IConfiguration _configuration;

        public BookController(IConfiguration configuration)
        {
            _configuration = configuration;
        }
```

بعد ذلك، تعالوا بنا ندخل إلى Action معينة من أجل جلب قيمة العنصر WelcomeMessage المخزنة في ملف appSettigs.json، وذلك كالآتي:

BookController.cs:

```
        public IActionResult Index()
        {
            var message = _configuration["WelcomeMessage"];

            return View("Index",message);
        }
```

الآن لو نفذنا سنحصل على النتيجة التالية:

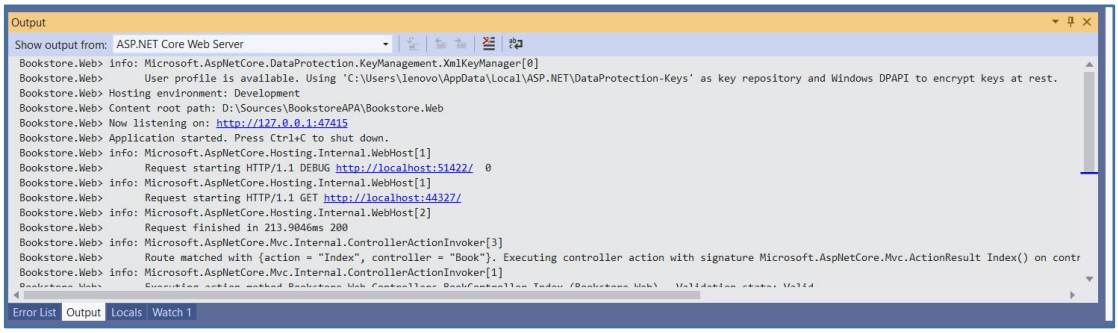
الصورة 75 - عرض البيانات من ملف الإعدادات

وبالتالي تمت عملية القراءة بنجاح من ملف الإعدادات appSettings.json.

آليات تتبع وتسجيل أنشطة التطبيق: Logging

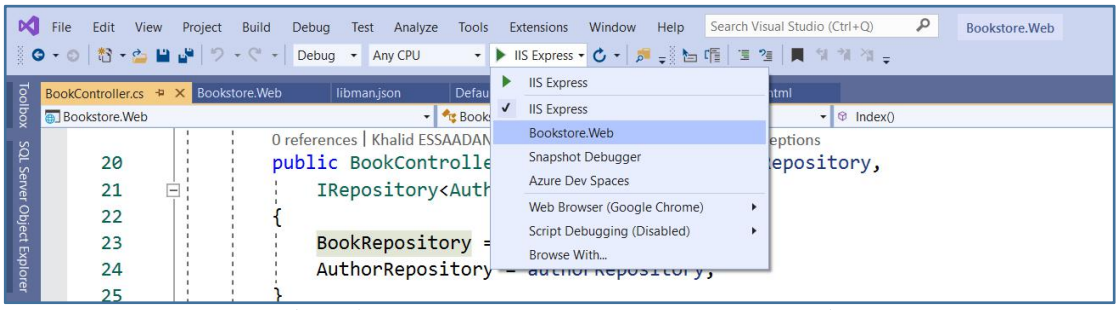
حينما نريد معرفة ما يقع بالتفصيل أثناء تنفيذ تطبيقنا، وتسجيل كافة الأحداث التي تجري، من أجل أن نتعرف على التنبيهات التي تحدث، أو الأخطاء التي تقع، أو المعلومات التي تعطى، أو نريد أن ندقق لنعرف سبب ضعف أداء التطبيق، فنحن عندئذ مطالبون باستعمال آلية Logging والتي تسمح لنا بتسجيل كافة الأنشطة التي يقوم بها تطبيقنا.

تسمح لنا تقنية ASP.NET Core بالاستفادة من آلية Logging بطريقة سهلة، حيث تقوم بتسجيل كافة أنشطة التطبيق وعرضها في واجهة output الخاصة بالفيجوال ستوديو كما يلي:



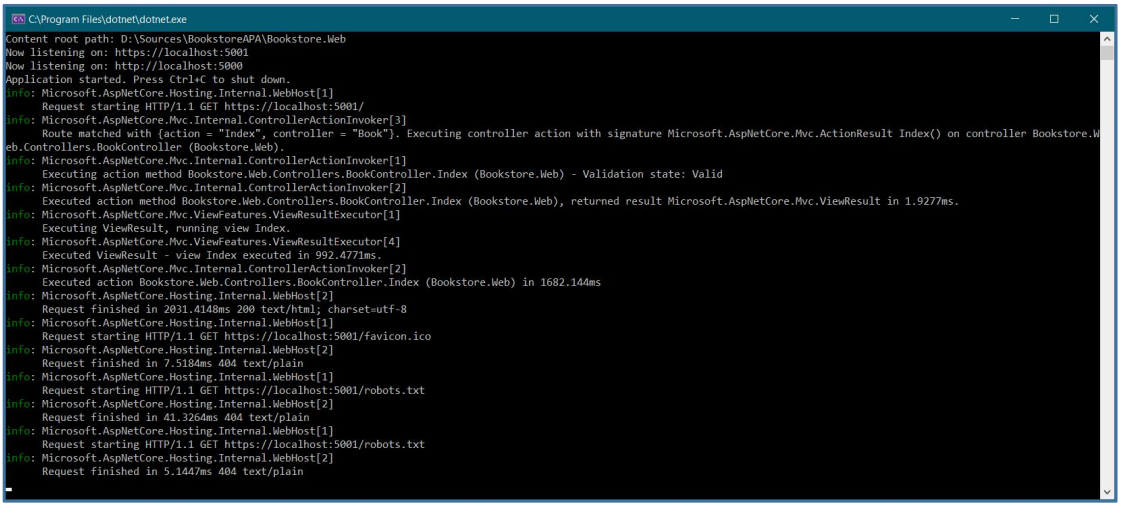
الصورة 76 - واجهة Output لعرض Logging details

يمكننا عرض نتائج Logging في شاشة console، من خلال تنفيذ التطبيق على Kestrel، حيث نذهب إلى زر التنفيذ ونختار اسم المشروع بدل IIS Express:



الصورة 77 - عرض Logging details في واجهة الكونسول

عند التنفيذ ستلاحظ ظهور شاشة الكونسول تعرض نفاصيل Logging كما يلي:



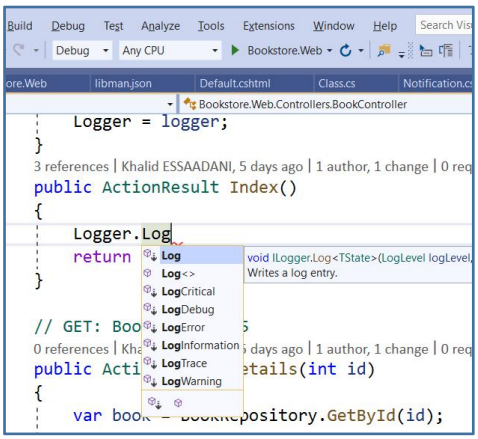
الصورة 78 - واجهة الكونسول

يمكننا أن نعمل Logging في الأماكن التي نكتب فيها الكود، بغرض تتبع الكود أو الحصول على بعض المعلومات، أو عرض بعض التنبيهات وما إلى ذلك، لعمل ذلك سنتعامل مع الواجهة ILogger والتي توجد في مجال الأسماء Microsoft.Extensions.Logging، والتي نمرر لها اسم الكلاس الذي نريد عمل Logging له، فعلى فرض أننا نريد عمل Logging ل Controller معين، سنأتي ونعمل Injection ل object من هذه الواجهة عبر مشيد controller كما يلي:

BookController.cs:

```
public ILogger<BookController> Logger { get; private set; }

public BookController(ILogger<BookController> logger)
{
    Logger = logger;
}
```

بعد ذلك نستطيع استعمال الكائن Logger من أجل تسجيل ما نشاء، علما أنه يمكننا أن نسجل معلومة من خلال الوظيفة LogInformation أو نسجل خطأ من خلال LogError، وغير ذلك كما تبين الصورة التالية:



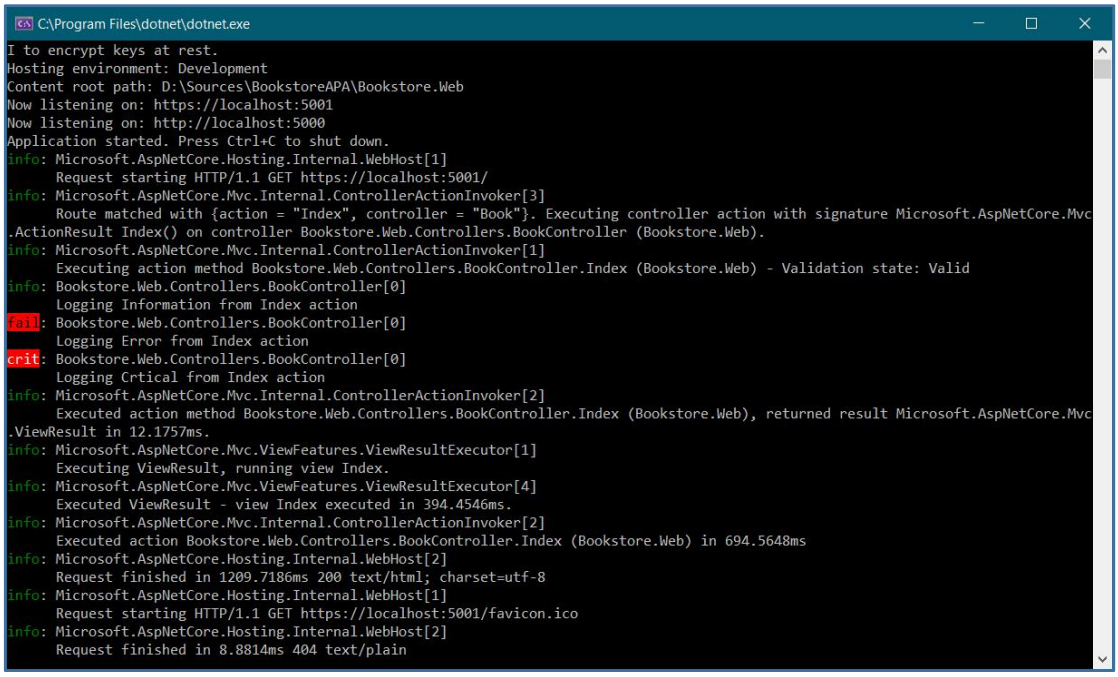
الصورة 79 - أهمية Intellisense

لنكتب الأوامر التالية من أجل عمل Logging:

BookController.cs:

```
        public IActionResult Index()
        {
            Logger.LogInformation("Logging Information from Index action");
            Logger.LogError("Logging Error from Index action");
            Logger.LogCritical("Logging Crtical from Index action");

            return View();
        }
```

عند التنفيذ والدخول إلى View المرتبطة ب Action التي عملنا فيها Logging، ستلاحظ تسجيل المعطيات كما وضعنا على شاشة الكونسول كما يلي:



الصورة 80 - عرض Custom Logging

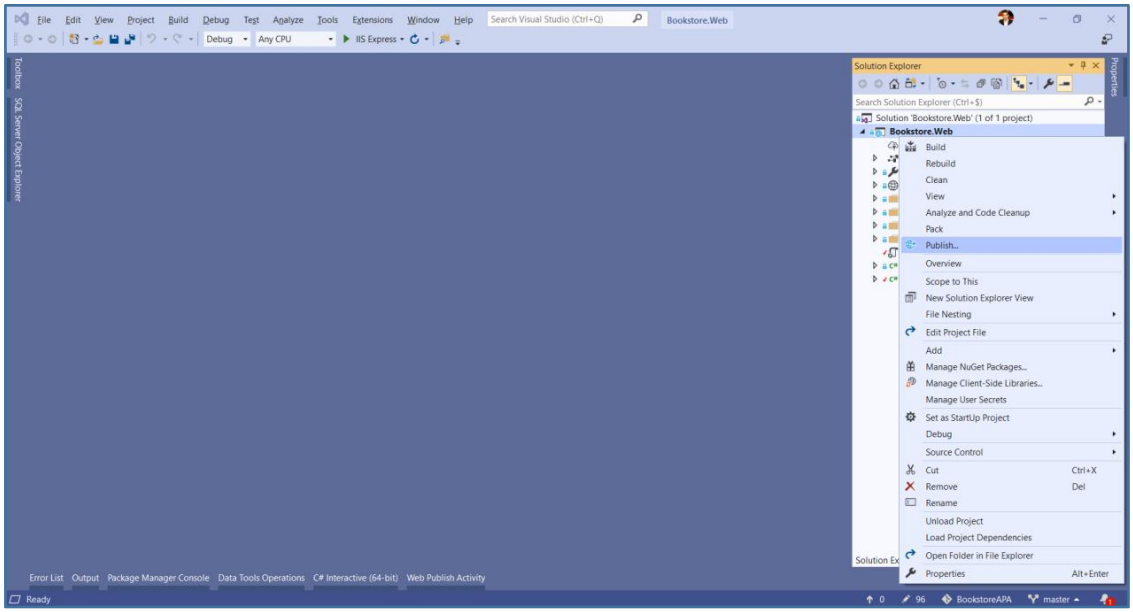
عند النهاية يكون النشر Publish، ثم الرفع Deploy !

حينما ننتهي من تطوير تطبيقنا، سنحتاج من غير شك إلى إعداده ليعمل على ويب سيرفر آخر من خلال استضافته على الانترنت أو رفعه على شبكة داخلية لمؤسسة لكي يصبح قابلا للاستعمال من عدة مستخدمين.

تسمح لنا تقنية ASP.NET Core بنشر الموقع ليعمل على عدة أنظمة سواء على نظام الويندوز من خلال الويب سيرفر IIS أو نظام لينكس على الويب سيرفر Apache أو Nginx.

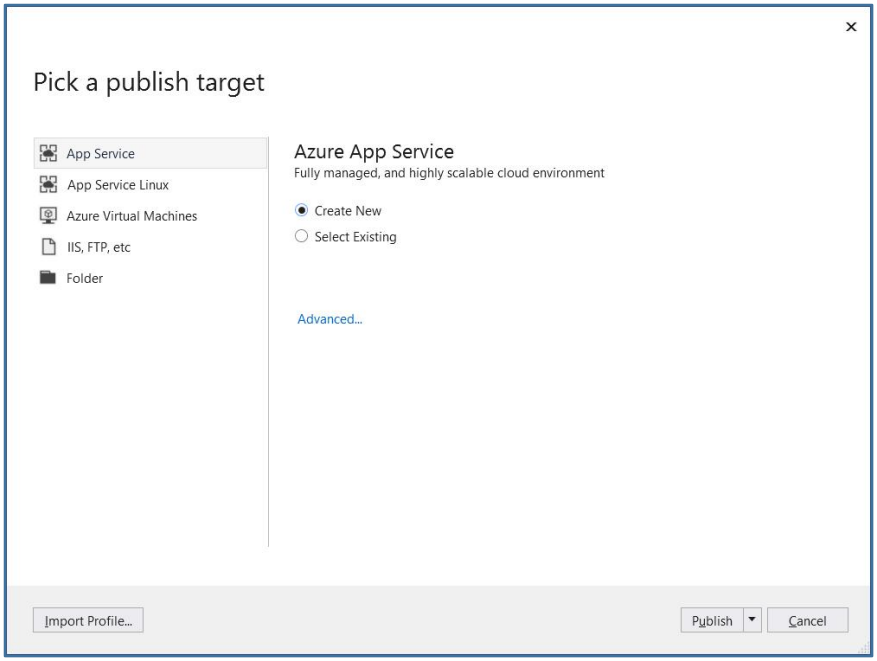
عملية نشر الموقع Publishing تعني أن تقوم بتجميع كل الصفحات والملفات والمكتبات التي يحتوي عليها موقعك في مجلد واحد بغرض رفعها على Host معين وتسمى عملية الرفع ب Deployment.

للقيام بعملية النشر فالأمر بسيط على الفيجوال ستوديو، إذ يكفي أن نضغط بيمين الماوس على اسم المشروع ثم نختار الأمر Publish كما يلي:



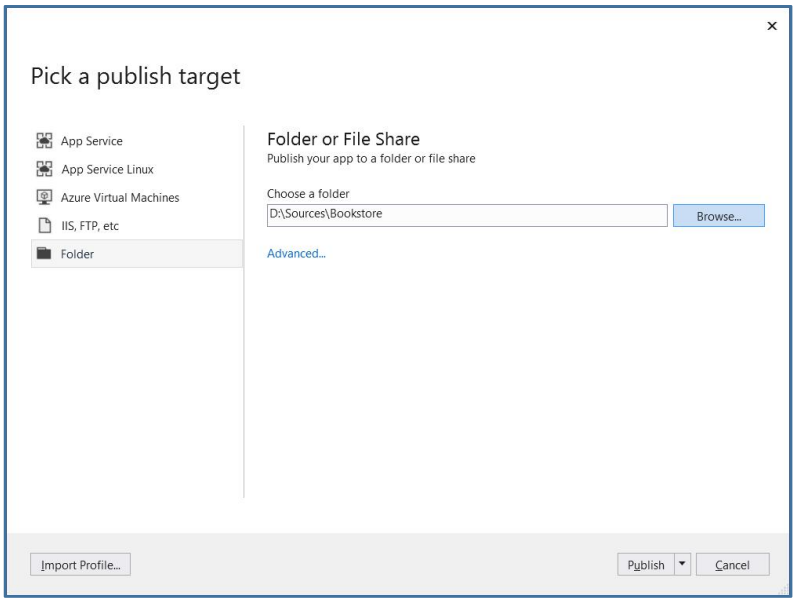
الصورة 81 - نشر الموقع

بعد ذلك ستطالعنا الواجهة التالية:



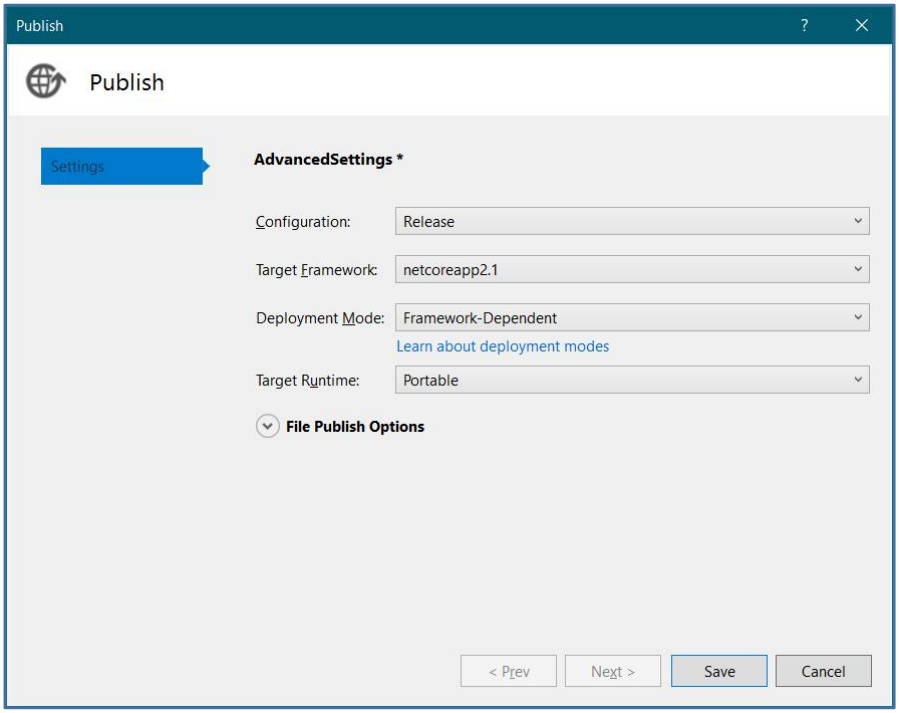
الصورة 82 - اختيار طريقة النشر

الواجهة أعلاه تسمح لنا بالاختيار بين عدة طرق لنشر الموقع حسب البيئة المستهدفة إما Azure App Service، أو App Service Linux، أو IIS في حالة استهداف نظام ويندوز، أو جمع ملفات التطبيق المنشور في مجلد واحد Folder، وهذا الخيار الأخير هو الذي سنتعامل معه في حالتنا، لذلك دعونا نختار الأمر Folder ثم نحدد مسار المجلد الذي نريد نشر التطبيق عليه كما توضح الصورة أسفله:



الصورة 83 - تحديد مسار مجلد النشر

يمكننا تغيير بعض الإعدادات الافتراضية من خلال الضغط على الزر Advanced، لتظهر لنا الواجهة التالية:



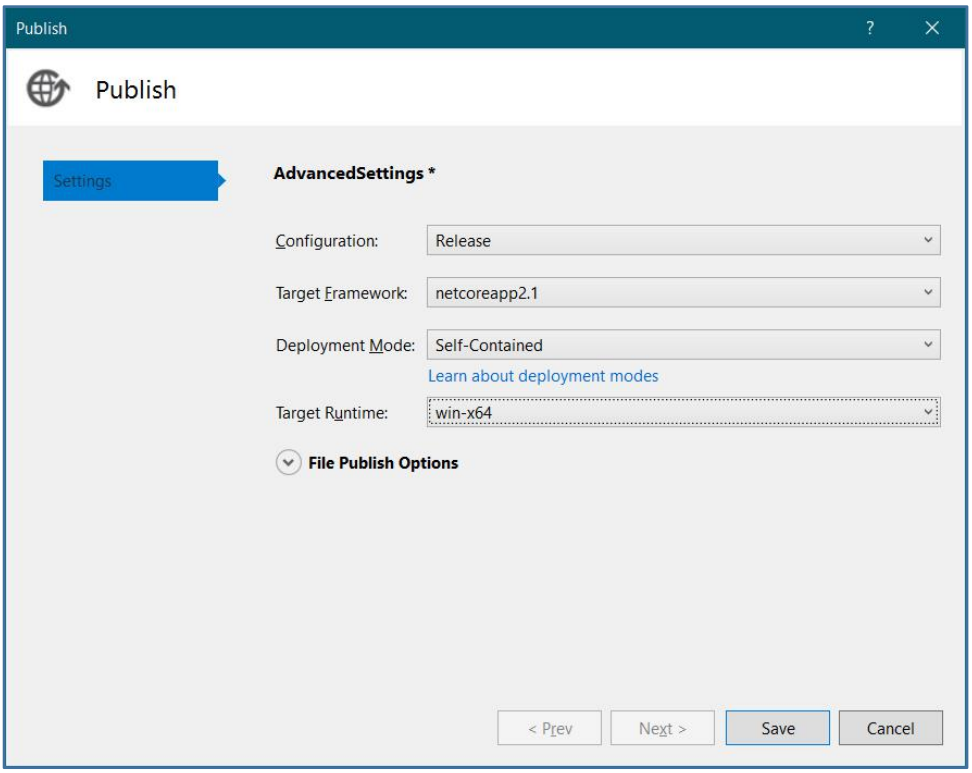
الصورة 84 - تغيير الإعدادات الافتراضية

نستطيع أن نختار نوع الإعدادات Configuration إما باختيار Release أو Debug، ويمكننا كذلك أن نختار إصدار الدوت نيت كور المراد استهدافه في بيئة الاستضافة، بالإضافة إلى نوع التوزيع Deployment Mode والذي يعطينا خيارين اثنين:

- **Framework-Dependent:** ويكون التطبيق عندئذ محتاج إلى وجود إصدار الدوت نيت كور المستهدف في بيئة الاستضافة لكي يتمكن من الاشتغال.
- **Self-Contained:** ويعني أن التطبيق سيحتوي على كل ما يلزمه للاشتغال وليس في حاجة إلى وجود الدوت نيت على بيئة الاستضافة.

يمكننا أيضا أن نختار نظام التشغيل المستهدف من Target Runtime حيث يمكننا أن نختار نظام الويندوز بإصداريه 32 و 64 بت، نظام لينكس، نظام OSX.

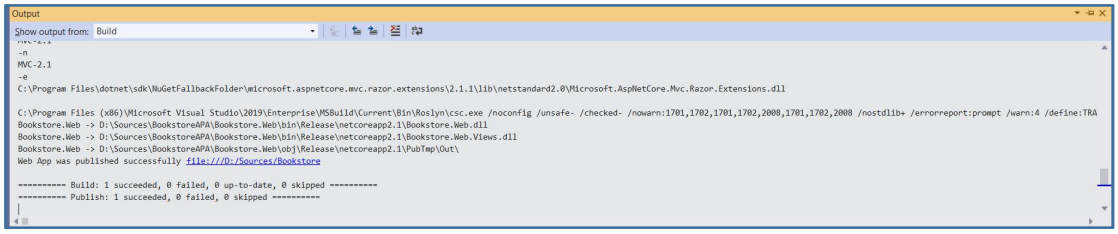
لأن النظام الذي سأجرب عليه هو نظام ويندوز 64بت، فسوف أختار هذا النظام، كما سأختار نوع التوزيع Self-Contained لأضمن أن تطبيقي سيعمل حتى في غياب الدوت نيت كور على الجهاز، وهذه هي الإعدادات التي قمت بتحديدها:



الصورة 85 - اختيار البيئة المستهدفة

بعد ذلك سأعود إلى واجهة النشر وأضغط بكل بساطة على الزر Publish، والذي سيقوم ببدء عملية نشر الموقع والتي ستأخذ وقتا حسب حجم المشروع، لذلك علينا أن ننتظر إلى غاية الانتهاء من عملية النشر.

عند الانتهاء ستظهر لك رسالة في نافذة Output تفيد بحالة النشر كما يلي:

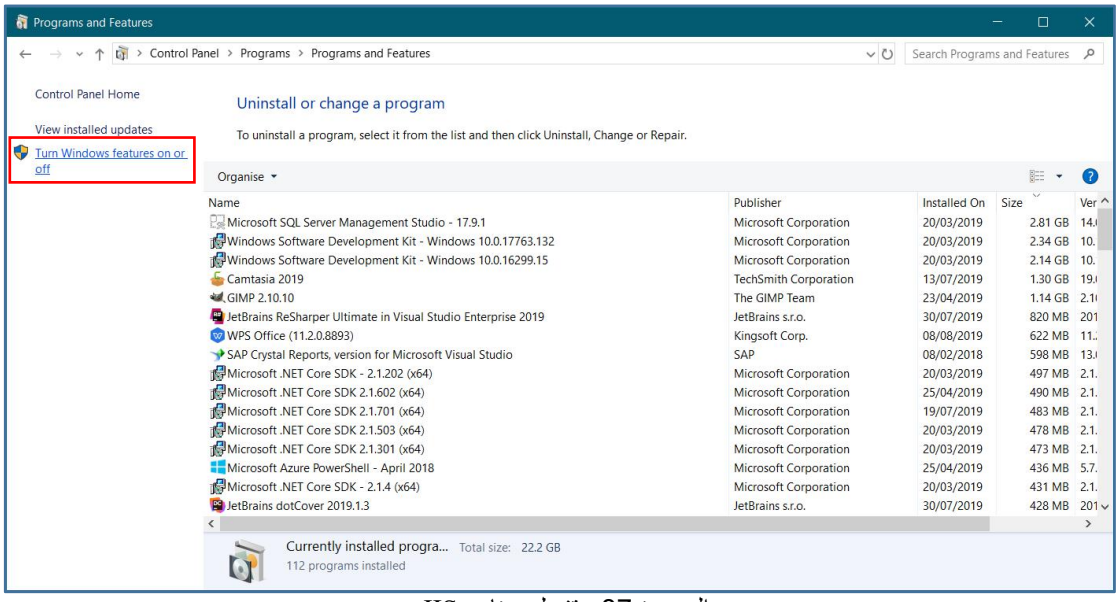


الصورة 86 - نتيجة انتهاء عملية النشر

الآن قمت بأغلب العمل وهي خطوات كما ترون يسيرة جدا، علما أننا نستطيع القيام بها خارج الفيجوال ستوديو بواسطة NET CLI. عبر تنفيذ بضعة أوامر على أحد واجهات الكونسول مثل Power Shell أو Command Prompt.

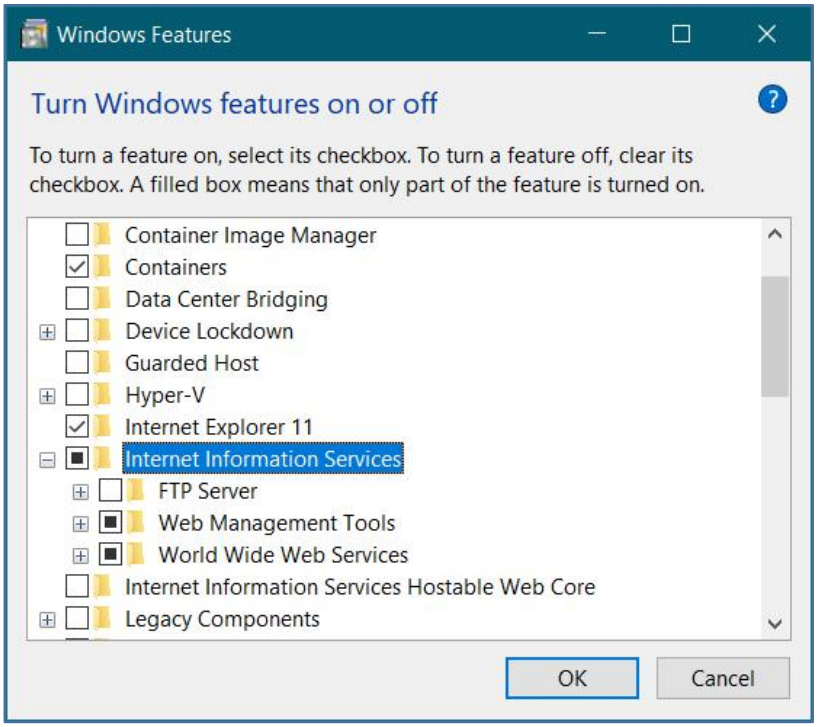
تفعيل IIS وتثبيت Hosting Bundle

الآن سنقوم بفتح IIS، إن لم يكن مثبتا في حاسوبك فالطريقة بسيطة، يكفي أن تدخل إلى واجهة حذف البرامج المثبتة على النظام:



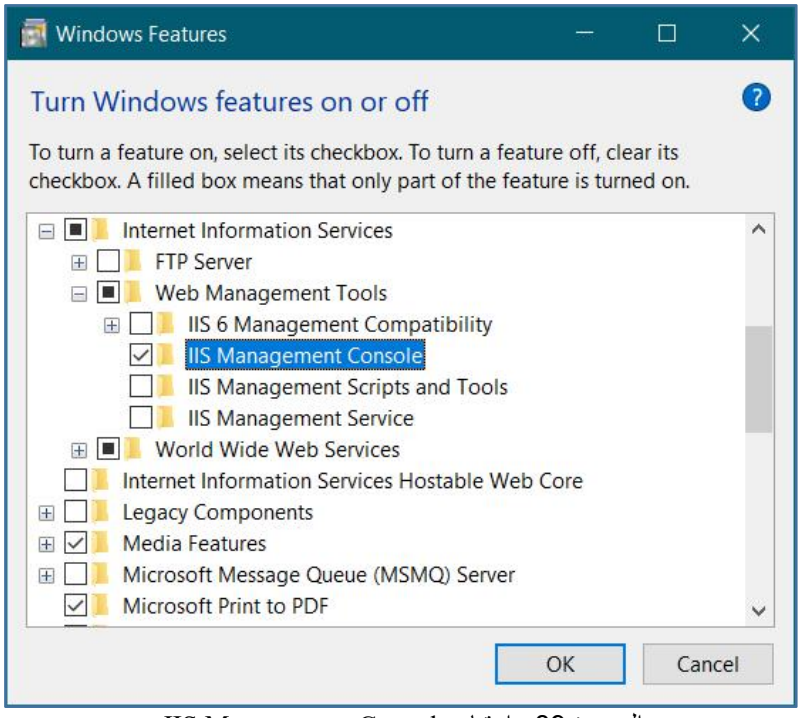
الصورة 87 - تفعيل برنامج IIS

ثم من القائمة الجانبية نختار Turn Windows features on or off وننتظر قليلا ريثما تظهر الواجهة التالية:



الصورة 88 - تحديد مكونات IIS

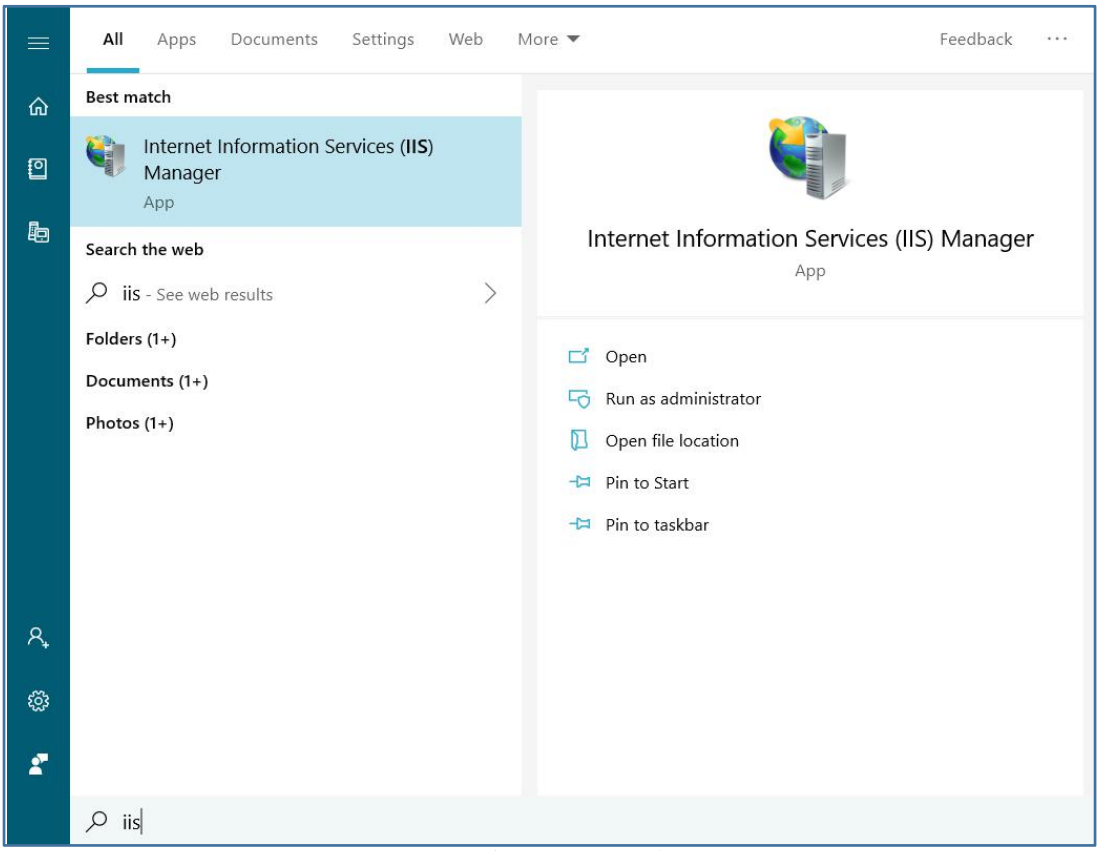
نقوم بتحديد Internet Information Services ونضغط على الزر OK، وينبغي أن تتأكد قبل ذلك من اختيارك لل IIS Management Console الموجودة أسفل Web Management Tools كما يلي:



الصورة 89 - اختيار IIS Management Console

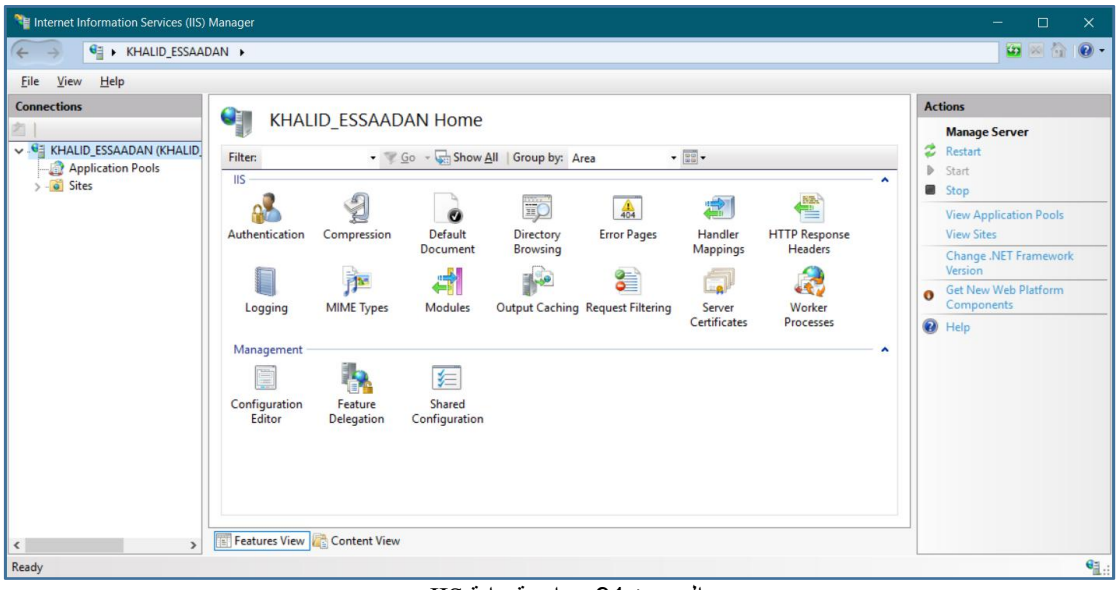
عملية تفعيل IIS ستأخذ بعض الوقت، عند الانتهاء يمكنك إعادة تشغيل الحاسوب إذا طلب منك ذلك.

بعد ذلك قم بالدخول إلى IIS من خلال قائمة Start:



الصورة 90 - تشغيل IIS

عند الدخول إلى IIS ستطالعك الواجهة التالية:



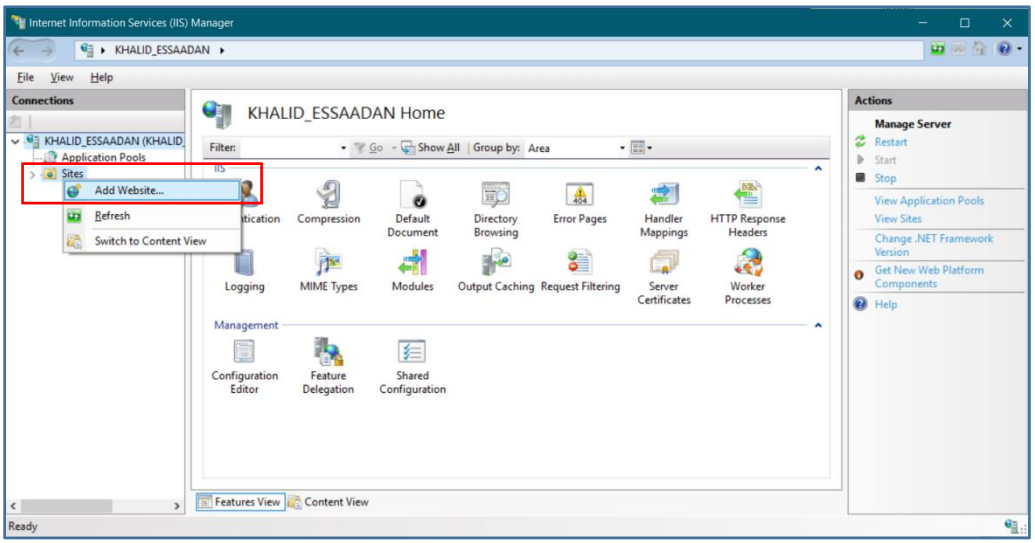
الصورة 91 - واجهة بداية IIS

الآن بقي لنا فقط أن نقوم بتثبيت أحد المكونات اللازمة لل IIS ليتمكن من تشغيل تطبيقات ASP.NET Core، هذا المكون اسمه Net Core Hosting Bundle.، ويمكنك تحميله من الرابط المباشر التالي:

```
Download .NET Core Hosting Bundle:

https://www.microsoft.com/net/permalink/dotnetcore-current-windows-runtime-bundle-installer
```

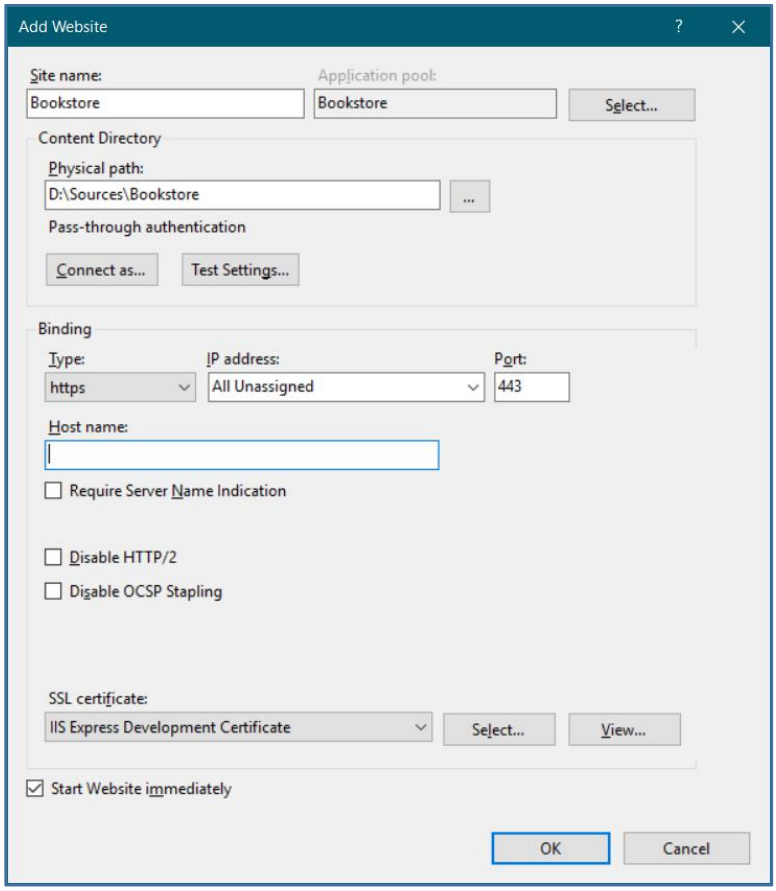
في حال تغير الرابط، يمكنك البحث في محرك البحث غوغل عن: Download ASP.NET Core Hosting Bundle وادخل إلى موقع ميكروسوفت وقم بتحميل وتثبيت البرنامج.

بعد ذلك قم بالدخول مجددا إلى IIS، ثم اضغط بيمين الماوس على Sites كما تبين الصورة أسفله وقم باختيار Add Website:



الصورة 92 - إضافة موقع جديد إلى الويب سيرفر

ستظهر لنا بعد ذلك الواجهة التالية:



الصورة 93 - إعدادات الموقع

في الأول نقوم بإدخال اسم موقعنا مثلا Bookstore، ثم نقوم بإدخال مسار المجلد الذي نتج عن عملية النشر Publishing ونقوم بوضوعه في Physical path، ثم من Binding type نختار الوضع الآمن HTTPS، ثم في الأسفل نقوم باختيار SSL certificate التي ستشتغل في الوضع الآمن ونختار IIS Express Development Certificate، نتأكد من أن الاختيار الموجود في الأسفل Start Website immediately محدد لنستطيع تصفح الموقع، ثم نقوم بالضغط على الزر OK.

الآن لندخل إلى اي متصفح ونكتب الرابط التالي:

```
Localhost Address:

https://localhost
```

ستلاحظ أن موقعك يعمل بنجاح كما لو أنه في Development environment، الآن بعد أن اختبرت موقعك محليا على IIS Web Server، يمكنك رفعه على الانترنت وسيعمل دون مشاكل مع الأخذ بعين الاعتبار أن تنتبه إلى بعض الإعدادات التي عليك تغييرها مثل نص الاتصال بقاعدة البيانات إن كنت قد سجلته في ملف Appsettings.json.

خاتمة

تقنية ASP.NET Core تقنية كبيرة جدا وشرحها يحتاج إلى عدة كتب، لكن أرجو أن أكون بهذا الكتيب قد أخذت بيدك ووضعتك على الطريق الصحيح، والباقي عليك أنت لأنك حزت الأدوات الأساسية، فاحرص أن تنطلق بعد هذا الكتاب باحثا عن مزيد من الشروح حول هذه التقنية، وصاحب هذا البحث بالتطبيق لتترسخ المفاهيم وتحصل الفائدة، ولا تنس الدعاء لصاحب هذا الكتاب ولوالديه ولعامة المسلمين.

وقبل أن أودعك سأزف إليك ببشرى قد تثلج صدرك:

ستجد على قناتنا في اليوتيوب عدة شروحات عن تقنية ASP.NET وعن مجموعة من التقنيات الأخرى التي ستفيدك بإذن الله في حياتك الدراسية والعملية، فلا تتردد في الانكباب على محتواها دارسا ومطبقا:

```
Khalid ESSAADANI Youtube Channel:

https://www.youtube.com/EssaadaniTV
```

بقي فقط أن أستأذنك بالذهاب، دام لك البشر والفرح!