**MWS – S23**

WDC Homework

Student services using microservices

**د. سيرا أستور**

|  |  |  |
| --- | --- | --- |
| الطالب | الرقم | الصف |
| محمد الشامي | Muhammad\_227086 | C3 |
| محمد عارف الشامي | Muhammad\_aaref\_218434 | C3 |
| فاطمة الزهراء الشامي | Fatima\_alzahra\_255920 | C3 |
| بشرى السطلي | Boushra\_252969 | C2 |

MWS\_WDC\_HW1\_S23\_C3\_muhammad\_227086\_C3\_Muhammad\_aaref\_218434\_C3\_Fatima\_alzahra\_255920\_C2\_Boushra\_252969

**جدول المحتويات**

[أولا – سيناريو عمل المشروع وتدفق البيانات 2](#_Toc148038740)

[ثانيا – أنواع المشاريع المستخدمة 3](#_Toc148038741)

[مشاريع الخدمات المصغرة Microservices Projects 3](#_Toc148038742)

[مشروع بوابة العبور/نقطة الربط Gateway Project 4](#_Toc148038743)

[مشروع واجهات المستخدم UI Project 4](#_Toc148038744)

[ثالثا – كيف يعمل النظام ؟ 5](#_Toc148038745)

[رابعا – التصميم والتنفيذ 6](#_Toc148038746)

[بنية الخدمة المصغرة Microservice 6](#_Toc148038747)

[أ - تهيئة الخدمة وتشغيلها 6](#_Toc148038748)

[ب – المتحكمات 7](#_Toc148038749)

[ج – العمليات 8](#_Toc148038750)

[د – تخزين واسترجاع البيانات 9](#_Toc148038751)

[بنية خدمة نقطة الربط Gateway 10](#_Toc148038752)

[أ – تهيئة التوجيه Routing 10](#_Toc148038753)

[ب – تهيئة وتشغيل الخدمة 11](#_Toc148038754)

[بنية واجهة المستخدم Portal 12](#_Toc148038755)

[أ – نوع البرمجية والتهيئة 12](#_Toc148038756)

[ب – الصف HttpCaller 13](#_Toc148038757)

[ج – الصفحات والاستدعاء 13](#_Toc148038758)

[خامسا – التشغيل 16](#_Toc148038759)

[إعدادات الخدمات 16](#_Toc148038760)

[أ – الاتصال بقواعد البيانات 16](#_Toc148038761)

[ب – إعدادات مخدم الخدمة 16](#_Toc148038762)

[إعدادات نقطة العبور 16](#_Toc148038763)

[أ – إعدادات المخدم والمنفذ 16](#_Toc148038764)

[ب – إعدادات التوجيه 16](#_Toc148038765)

[إعدادات الواجهة 17](#_Toc148038766)

[أ – إعداد المخدم والمنفذ 17](#_Toc148038767)

[ب – إعداد عنوان الربط مع Gateway 17](#_Toc148038768)

[تشغيل التجربة 17](#_Toc148038769)

أولا – سيناريو عمل المشروع وتدفق البيانات

* يتم تسجيل بيانات الطالب الأساسية من خلال الخدمة Admission ويمكن أيضا أن يتم رفع مجموعة من الملفات مع البيانات
* تكون حالة الطالب الافتراضية (غير مقبول)
* في نفس الخدمة يمكن قبول بعض الطلاب
* إذا تم قبول أحد الطلاب تظهر بياناته في خدمة Register للتسجيل على المواد
* من خلال الخدمة Register يمكن
  + إنشاء مواد
  + ربط الطالب مع المواد
  + وعرض المسجلين في مادة ما
* عند تسجيل الطالب في إحدى المواد يظهر اسمه في الخدمة الثالثة للتفقد Attendance
* من خلال الخدمة الثالثة يمكن أن يتم
  + إنشاء جلسات لمادة
  + ووضع حالة الطالب (حضور) في الجلسة التي تتبع للمادة التي سجل فيها
  + يمكن أيضا عرض الجلسات حسب المادة وحالة كل طالب

ثانيا – أنواع المشاريع المستخدمة

مشاريع الخدمات المصغرة Microservices Projects

كل خدمة من الخدمات الثلاث المقترحة عبارة عن مشروع Asp.Net Core Web API مبني باستخدام Visual Studio 2022 وباستخدام الإطار .Net 6

يحتوي كل مشروع على المجلدات والملفات الأساسية التالية:

* ملف Program.cs ملف بدء التنفيذ يحتوي على الدالة Main التي تقود تهيئة البرمجية وتشغيلها
* المجلد Models يحتوي على الصفوف التي توصف بيانات التخزين لهذه الخدمة
* المجلد Controller يحتوي صفوف المتحكمات Controllers التي بدورها تحوي العمليات التي تخص كل خدمة والتي يمكن استدعاؤها باستخدام بروتوكول Http
* المجلد Classes يحتوي على صفوف مساعدة حسب المشروع, وفي كل المشاريع يوجد الصف AppSettings الذي يعبر عن الإعدادات الموجودة في ملف appsettings.json
* المجلد Data يحتوي على صف DataContext والتي تعتبر مخزن البيانات عند استخدام تقنية Entity Framework, حيث تتم كل عمليات الاستعلام والإدخال والتعديل عن طريقه.
* المجلد Dtos يحتوي على الصفوف التي تعتبر نماذج مساعدة للإرسال والاستقبال   
  (View Model, Data To Object)
* المجلد Mapping يحوي صف لضبط إعدادات AutoMapper المساعد في التحويل بين Entity Object و صفوف النماذج Dto

مشروع بوابة العبور/نقطة الربط Gateway Project

تم إنشاء المشروع أيضا كمشروع Asp.Net Core Web API مبني باستخدام Visual Studio 2022 وباستخدام الإطار .Net 6 والاستغناء عن كل ما يحتويه باستثناء ملف Program.cs, ويعمل هذا المشروع كبوابة للخدمات المصغرة باستخدام مكتبة Ocelot وملف Ocelot.json الذي يحوي إعدادات التحويل, وسيتم شرحه لاحقا

مشروع واجهات المستخدم UI Project

هذا المشروع يمكن أن يكون أي نوع من المشاريع Windows Forms, Console, Web application حيث يمكن ببساطة استدعاء Web API باستخدام HttpClient في أي مشروع في .Net

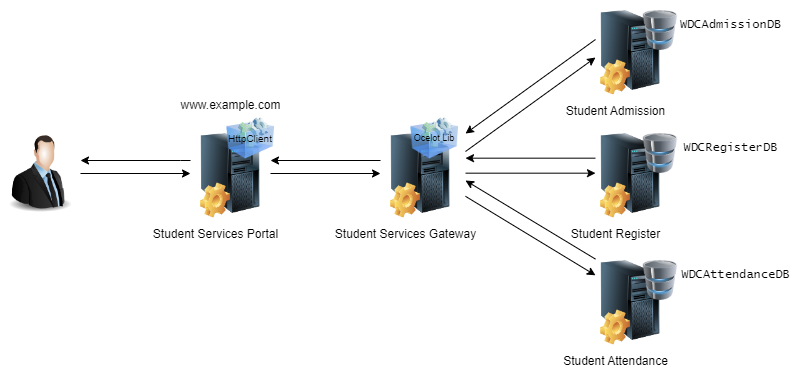
اخترنا العمل بنوع مشروع Asp.Net Core Blazor Server وهي تقنية جديدة من ميكروسوفت لدعم طريقة One Page Application نأتي على شرحها لاحقا

يحتوي المشروع على المجلدات التالية

* مجلد wwwroot الذي يعبر عن المجلد الرئيسي الذي يحوي كافة مستلزمات الصفحات css, js
* مجلد Classes يحتوي على صفوف مساعدة أهمها
  + HttpCaller: وهو صف قمنا بإنشائه لتغليف الصف الرئيسي HttpClient وتسهيل عمليات الاستدعاء و معالجة الأخطاء
  + AppSettings: يمثل الإعدادات المكتوبة في ملف appsettings.json
  + ErrorComponent: يمثل طريقة لمعالجة الأخطاء وعرضها بشكل مناسب في الصفحات
  + غير ذلك
* مجلد Models يحتوي على صفوف مشابهة لصفوف النماذج في الخدمات المصغرة. من أجل قراءة البيانات من الخدمات ومعاملتها كأغراض , مع ملاحظة أنه يمكن إنشاء مشروع مكتبة مشتركة Shared Library لتحوي تعريفات النماذج المشتركة بين المشاريع
* مجلد Pages يحوي الصفحات وعمليات استدعاء الخدمات وعرض النتائج ضمن الصفحات

ثالثا – كيف يعمل النظام ؟

من خلال المخطط التالي يمكن شرح كيفية عمل النظام



كل برمجية يمكنها أن تعمل على مخدم مستقل, ويمكن أن تعمل على نفس الجهاز مع التأكيد على تغيير المنفذ. ولكل خدمة مصغرة قاعدة بيانات مستقلة ولضبط إعدادات عمل البرمجيات (كل واحدة على حدة) يمكن تحرير ملف appsettings.json في مجلد البرمجية.

يقوم المستخدم باستعراض الموقع الذي يحوي الواجهات, والذي بدوره يستخدم المكتبة HttpClient للاتصال بGateway لطلب البيانات أو إرسالها.

تترجم Gateway الطلبات باستخدام المكتبة البرمجية Ocelot بمعاونة الملف Ocelot.json الذي يحوي كل تفاصيل تحويل الطلبات من وإلى الخدمات كنوع الطلب, مساره, منفذه ... الخ. ثم يتم إرسال الطلب إلى الخدمة المطابقة حسب Template إلى الخدمة المناسبة ويتم انتظار الرد, ثم يتم تمرير الرد إلى Portal الذي بدوره يقوم بعرض النتيجة للمستخدم.

كل خدمة يمكن أن تحوي على قاعدة بياناتها الخاصة بدون أن تعتمد على خدمة أخرى.

ينصح الخبراء بعمل تحليل ودراسة للخدمة بحيث تكون مستقلة عن مثيلاتها في كل شيء بحيث لا يؤثر توقفها على الخدمات ولا تتأثر بتوقف غيرها. ينصح الخبراء أيضا باتباع أسلوب DDD في تطوير الأعمال وهو أسلوب لحل المشكلات على أساس مجالها [[1]](#footnote-1).

رابعا – التصميم والتنفيذ

بنية الخدمة المصغرة Microservice

أ - تهيئة الخدمة وتشغيلها

في.Net Core يتم تهيئة الخدمات في ملف Program.cs الذي يحوي التعليمات التالية (على الأقل):



بين سطور هذه التعليمات يمكن أن يتم 1- حقن بعض الخدمات Dependency Injection في حاوية الخدمات container حيث تصبح الخدمات التي تم تكوينها متاحة عبر التطبيق. 2- كما ويتم أيضا تهيئة البرمجيات الوسيطة Middleware [[2]](#footnote-2).

الفرق بين الخدمة والبرمجية الوسيطة أن الخدمة هي مجموعة من العمليات يتم استدعاؤها في البرنامج بحسب اللزوم (بعد حقنها في حاوية الخدمات), أما البرمجية الوسيطة فيتم توصيلها في Pipeline الخاص بمعالجة الطلب Request بعد حقنها فتنفذ تلقائيا على كل طلب (مثل المصادقة والتحقق Auth).

تضاف الخدمات عن طريق السطر builder.Services.Add.. أما البرمجيات الوسيطة فتضاف من خلال السطر app.Use..

في كل خدمة من خدمات المشروع الحالي, يتم الآتي

* حقن الخدمة AppSettings ليتم التعامل معها لأجل قراءة الإعدادات (يهمنا من الإعدادات حاليا سلسلة الاتصال Connection String) وذلك بعد تحميل الملف appsettings.json ثم تهيئة الخدمة
* حقن خدمة AutoMapper وهي برمجية مساعدة الهدف منها (حاليا) نسخ البيانات بين غرضين من نوعين مختلفين
* حقن خدمة المتحكمات Controller ليتم تفسير الطلبات وتحويلها لها
* حقن خدمة مستودع البيانات DataContext ليتم استخدامه في الخدمة والتعامل مع قاعدة البيانات المخصصة لهذه الخدمة

عند حقن خدمة في حاوية الخدمات ... يمكن أن يتم استخدامها مباشرة في أي غرض يتم إنشاؤه من قبل التطبيق بعد وضعها كبارامتر للباني



بعد أن يتم حقن الخدمات يتم إنشاء التطبيق, ثم يتم بناء Pipeline وإضافة Middleware إليه

* برمجية التحقق والمصادقة
* برمجية نقاط الاتصال بالمتحكمات

كما أنه يتم التأكد من بناء قاعدة البيانات وتحديثها من خلال السطرين الموجودين في عملية التهيئة.



يستخدم هذان السطران أدوات مكتبة Entity Framework لإنشاء قاعدة البيانات طبقا للموديل المعتمد. وتكوينها تبعا لتغير الموديل (نمط البيانات المخزنة) الموجود في الخدمة.

ب – المتحكمات

كل خدمة تحوي على متحكم Controller (على الأقل), بدوره يحوي على العمليات التي تستطيع الخدمة إنجازها. يستخدم في المتحكم غرض من كل من DataContext, Mapper, AppSettings .



المتحكم يوسم بـ [ApiController] ليتم معاملته كمتحكم طلبات WebAPI ,أيضا يوسم المتحكم بالوسم Route كما في السطر التالي, لتحديد كيفية شكل مسار الطلب

[Route("api/[controller]/[action]")]

حيث يتم لاحقا الطلب بالشكل التالي

http://{server}:{port}/api/{controller}/{action}/{id}

* + server:port اسم جهاز ومنفذ الخدمة
  + controller اسم المتحكم
  + action اسم العملية في المتحكم
  + id اختياري, يتم تمريره إذا لزم الأمر

نلاحظ أن الباني يحوي على بارامترات وهي الخدمات التي تم حقنها سابقا.

الهدف الرئيسي من Mapper في المشروع هو تحويل البيانات من الصيغة المرسلة RequestModel إلى نموذج تخزين Model. أو تحويل نموذج البيانات المخزنة Model إلى صيغة عرض ViewModel .

مثال:

حيث model هنا من النوع AdmissionModel (نموذج بيانات مرسلة) وسيتم تحويلها إلى Admission (نموذج تخزين البيانات الفعلي) ليتم حفظها لاحقا.

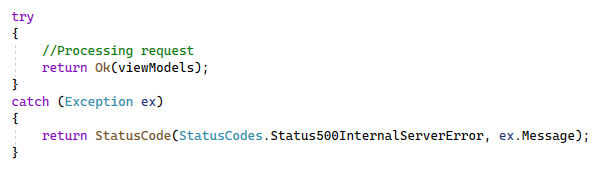
ويتم ضبط إعدادات Mapper والتحكم في عملية التحويل من خلال الصف MappingProfile الموجود في المجلد Mapping, بشكل افتراضي يتم التحويل ونسخ قيم البيانات من غرض إلى آخر من خلال تطابق اسم الخاصية وإمكانية تحويل نوع البيانات بين الخاصيتين في المصدر والهدف.

ج – العمليات

تتلقى الخدمة المصغرة طلبات Http من الأنواع Post, Get, Put (ويمكن إضافة أنواع أخرى مثل Delete و ... الخ). وتنفذ العملية على البيانات المقدمة وتعاد النتيجة من النوع IActionResult حيث يمكن أن تغلف النتيجة بأي StatusCode مع رسالة مناسبة.

يتم وسم العملية بنوع الطلب [HttpGet] أو [HttpPost] أو [HttpPut] حيث يتم تمييز الخدمة عن طريق اسمها ونوع الطلب الذي تتقبله (ويمكن ضبط هذا الإعداد بشكل عام). يمكن أن يوضع وسوم أخرى للعملية مثل الوسم [ActionName("NewName")] في حال رغبنا باستخدام اسم مختلف للعملية عن الاسم الفعلي لها. أما الوسم [Consumes("application/json")] فيستخدم لتحديد نوع البيانات المرسلة لهذه العملية أنها json مثلا. ليس بالضرورة استخدام هذه الوسوم لكل عملية, وإنما تستخدم حسب الحاجة.

كل العمليات لها الشكل العام التالي:



عمليات الاستعلام والتخزين تتم عن طريق Linq2Entity حيث تستخدم الخدمة DataContext في المتحكم ويتم إنشاؤها تلقائيا من قبل مخزن الخدمات عند الطلب (وهناك أنواع إنشاء أخرى). بعد انتهاء العمل مع البيانات (الاستعلام/التخزين) إذا نجحت المعالجة سيتم إعادة OK (لها قيمة 200 التي تعني نجاح الطلب) ويمكن تضمين أي غرض معها (view model / اختياري) ليتم إعادته للزبون. أما إن فشلت العملية أو حدث استثناء ما فيمكن أن نعيد شيفرة خطأ مناسبة (بين 400 – 500) مع شرح المشكلة ex.Message

د – تخزين واسترجاع البيانات

في مجلد Models تملك الخدمة توصيفا لبياناتها التي تريد تخزينها. حيث تم اعتماد استخدام تقنية Entity Framework Code First أي نقوم بإنشاء نموذج اليبانات Model وعلى أساسه وباستخدام الاتصال المخزن في appsettings.json وإعدادات DataContext سيتم إنشاء قاعدة البيانات.

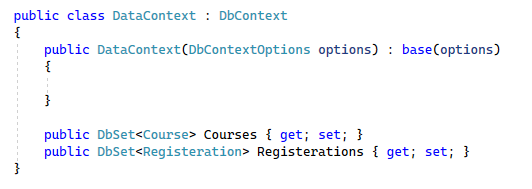
يمكن وخلال التطوير أن يتم إنشاء القاعدة من خلال إدخال الأمر التالي في Nuget-Package manager console:

update-database -project X

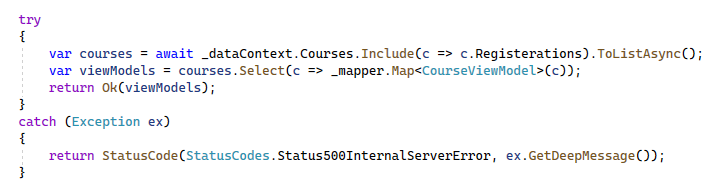
أو عندما يتم تشغيل الخدمة بسبب السطرين في ملف Program.cs المسؤولين عن إنشاء قاعدة البيانات

يحوي الصف DataContext الموجود في المجلد Data على DbSet من أنواع النماذج/Model المستخدمة في المشروع, وتعبر كل DbSet عن جدول في قاعدة البيانات. يتم الاستعلام منه, حشر البيانات فيه وتعديلها.

مثال في خدمة تسجيل الطلاب على المواد:



للاستعلام عن كل المواد وعن الطلاب المسجلين في كل مادة نكتب

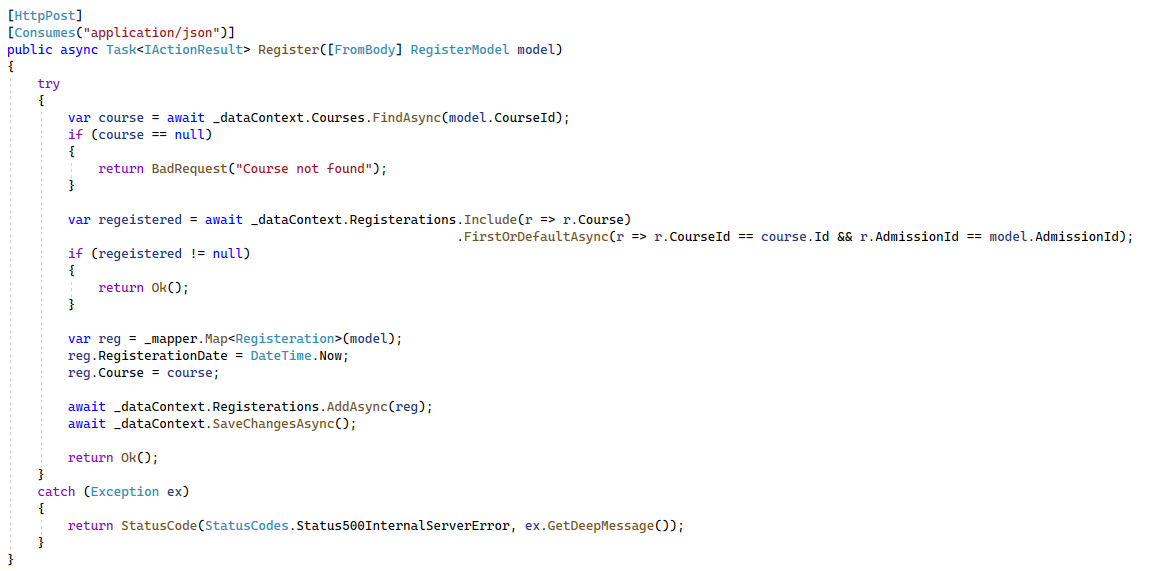


نلاحظ استخدام Mapper لتحويل Model الناتج من الاستعلام إلى ViewModel وذلك لتسهيل العرض لدى المستخدم

أما لتسجيل طالب في مادة ما يمكن أن نكتب الكود التالي:

حيث RegisterModel هي Dto من أجل إرسال البيانات اللازمة للمتحكم والعملية, ثم باستخدام Mapper نحولها إلى Registeration وهي نموذج تخزين Model, ليتم تخزينها في قاعدة البيانات.

تتواجد النماذج المساعدة Dtos, ViewModels في مجلد Dtos

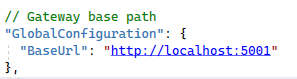


بنية خدمة نقطة الربط Gateway

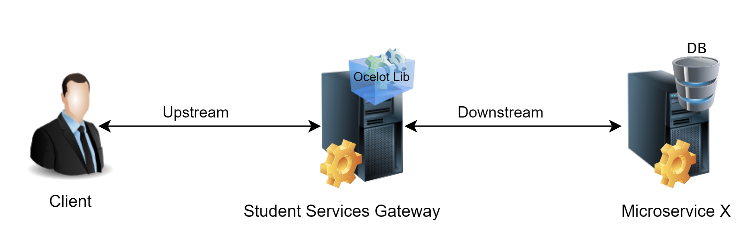
أ – تهيئة التوجيه Routing

يتم إضافة المكتبة Ocelot من Nuget package manager إلى المشروع StudentServiceGateway, يجب مراعاة إصداء المكتبة وتوافقها مع نسخة .Net 6 التي اعتمدناها في إنشاء المشروع. ثم يتم إضافة الملف Ocelot.json إلى المشروع واتباع التعليمات الواردة في الموقع الرسمي للمكتبة [[3]](#footnote-3) اللازمة لتحقيق Routing فقط

تم إعداد ملف Ocelot.json في المشروع بالشكل التالي:

* قسم الإعدادات العامة: فيه تم ضبط المسار الأساسي للخدمة ويجب أن يطابق – تماما – المسار الذي يحدد في الملف appsettings.json
* قسم إعداد التوجيه Route

يتم فيه إعداد خيارات التوجيه الخاصة بكل خدمة حيث يتم توصيف DownStream, UpStream

ولتوضيح المفهومين السابقين نستخدم الرسم التالي:

في كل طلب: يعتبر المسار بين الزبون وخدمة التوجيه هو UpStream, ويعتبر المسار بين خدمة التوجيه وأي خدمة مصغرة أخرى هو DownStream

الخصائص التي يتم ضبطها لل UpStream هي

* UpstreamPathTemplate : يعبر عن المسار وشكل الطلب الجديد, حيث يدمج مع BaseUrl في الإعدادات العامة
* UpstreamHttpMethod : أنواع طلبات Http التي سيسمح بها (Get, Post, …etc)

أما الخصائص التي يتم ضبطها للDowntream فهي

* DownstreamPathTemplate : مسار وشكل الطلب إلى الخدمة المصغرة
* DownstreamScheme: يحدد بروتوكول الاتصال (Http, Https …etc)
* DownstreamHostAndPorts : يحدد مخدم ومنفذ الخدمة المصغرة التي سيتم الاتصال بها.

في كل طلب, تدمج هذه الثلاث خصائص لتشكل مسارا كاملا, يتم تمرير الطلب إليه. وبدلا من تحديد اسم المتحكم, اسم العملية, بارامترات العملية من أجل كل عملية في المتحكم, تتيح لنا المكتبة أن نعيد توجيه كافة عمليات إحدى المتحكمات باستخدام {everything} كما في المثال التالي:

يوضح المثال أن كل ما يطلب من الخدمة الموجودة في

http://localhost:5001/apigateway/AdmissionService

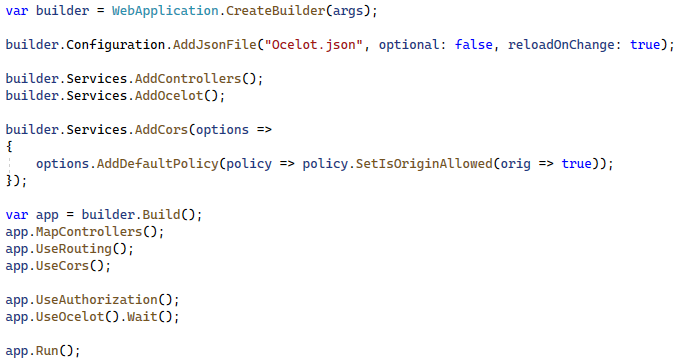
سيتم تمريره كما هو إلى

http://localhost:5002/api/Admission

أيضا تم وضع إعدادات مشابهة للمثال السابق لبقية الخدمات مع الانتباه على المخدم والمنفذ لكل خدمة.

ب – تهيئة وتشغيل الخدمة

كما شرحنا في الخدمات المصغرة, سيتم وضع عمليات التهيئة والتشغيل في ملف Program.cs وسيبدو مشابها للتالي:



في الكود السابق تم إنجاز الأعمال التالية:

* تحميل ملف إعدادات Ocelot
* إضافة خدمات المتحكمات
* إضافة خدمة Ocelot إلى المخزن
* إضافة خدمة السماح CORS وهي اختصار ل(Cross-Origin Resource Sharing) حيث تسمح بالطلب من جهاز آخر غير الجهاز الحالي, مع تحديد سياسة السماح بكل شيء (للتجربة)
* بناء التطبيق
* إضافة البرمجيات الوسيطة Middleware ومنها Ocelot (وبذلك نضمن أن Ocelot ستقوم بمعالجة وتوجيه كل الطلبات الموافقة لملف الإعداد)

بنية واجهة المستخدم Portal

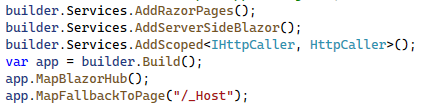
أ – نوع البرمجية والتهيئة

في هذا الجزء يمكن – كما أسلفنا – أن يتم اختيار أي نوع مشروع من مشاريع Visual Studio, حيث يتم استخدام المكتبة HttpClient لإجراء الطلبات بكافة أنواعها (GET, POST, …etc) وتم اختيار نوع المشروع Asp.Net Core Blazor Server

تقنية Blazor هي تقنية جديدة تم إصدارها مع .Net Core لأجل دعم برمجيات الويب ذات الصفحة الواحدة. حيث يتم إرسال الطلبات إلى المخدم واستلام الاستجابة وتعديل عناصر الصفحة بدون أن يتم إعادة التحميل كليا. ويوجد نوعين رئيسيين من أنواع Blazor وهما:

* Blazor Server: حيث يتم تنفيذ الأكواد فعليا على المخدم
* Blazor Web Assembly: حيث تحمل البرمجية كـ WASM إلى جهاز الزبون, ويتم العمل وتنفيذ الأكواد فعليا على جهاز الزبون [[4]](#footnote-4)

في ملف Program.c يتم تهيئة البرمجية ونلحظ بعض السطور الإضافية (لدعم Blazor) كما في التالي



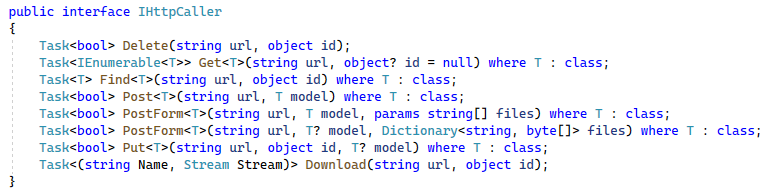
* إضافة خدمة صفحات Razor
* إضافة خدمة ServerSideBlazor دعم ترجمة blazor
* إضافة خدمة HttpCaller (الصف المساعد لطلبات Http)
* إضافة برمجية BlazorHub لمعالجة الطلبات
* تحديد صفحة البدء الفعلية

كما في MVC سابقا فإن Blazor تقوم بتركيب الصفحات والمكونات بطريقة سلسة أكثر. حيث يمكن أن يتم تضمين صفحة في صفحة أخرى أو تصمين مكونات في عدة صفحات, وكل مكون يمكن أن يحوي على طريقته الخاصة في العمل

كل صفحة/مكون في Blazor يكون له الامتداد razor حيث يكتب بطريقة Razor (التي تدعم إمكانية استعمال أكواد C# داخل وسوم Html) التي يتم ترجمتها عند المخدم إلى وسوم Html

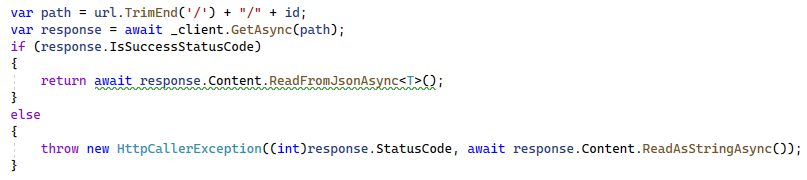
ب – الصف HttpCaller

وهو صف مساعد تم إنشاؤه بهدف تسهيل استدعاءات Http في البرمجية. وله العمليات التالية:



يحقق الصف HttpCaller العمليات المذكورة في الواجهة, ليسهل من استخدام HttpClient وعدم إعداده كل مرة.

مثال: تحقيق عملية Find إيجاد عنصر واحد



يتم استدعاء التابع GetAsync وانتظار النتيجة, ثم التأكد من أن النتيجة عادت بشفرة نجاح ليتم تحويل النتيجة إلى النوع T المطلوب. أما إن فشلت العملية فيتم إعادة خطأ مع رقمه ورسالته

لاحقا في الصفحات ... يكفي أن نكتب الكود التالي لاسترجاع الطالب ذي الرقم x

var std = await Caller.Find<RegisterViewModel>("RegisterService/Find",x);

حيث Caller هو غرض من IHttpCaller والنوع T هو RegisterViewModel والمسار يدمج مع مسار مخدم Gateway ليصبح Url كاملا (تم وضع مخدم ومنفذ Gateway في appsettings.json في حال أردنا تغيير مسارات ومنافذ الخدمات) , أما x فهي رقم التسجيل.

يوضع الاستدعاء السابق ضمن Try, Catch في حال حصول خطأ سيتم عرضه بدون أن تتوقف الصفحة عن العمل

ج – الصفحات والاستدعاء

الصفحات في Blazor تعتمد مبدأ المكونات, أي أن كل صفحة تتألف من عدة مكونات, كل مكون أيضا يمكن أن يتكون من مكونات جزئية أخرى وهكذا. يتم في الصفحة كتابة كود HTML ويمكن تضمين أكواد C# باستخدام @{} حيث يتم ترجمتها في المخدم وتحويل النتيجة إلى وسوم HTML.

تتميز الصفحة من خلال وضع رابط الصفحة في الأعلى باستخدام التوجيه @page أما المكونات فلا داعي لها

مثال: صفحة التسجيل

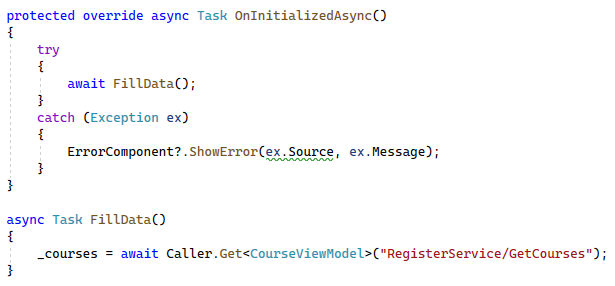


يستخدم التوجيه @Inject لاستدعاء غرض من Dependency Injection Container الذي تم إعداد الخدمات فيه في ملف Program.cs

في الصفحات نقوم بحقن الغرض HttpCaller من أجل القيام بالاستدعاءات اللازمة لعمل الصفحة.

في القسم @code يمكن تعريف متحولات, بارامترات, أحداث وغير ذلك. ويمكن إعادة كتابة override الأحداث الأساسية مثل OnInitilized وغيرها

مثال : في مكون المواد الذي يعتبر جزءا من صفحة التسجيل قمنا بكتابة الكود التالي

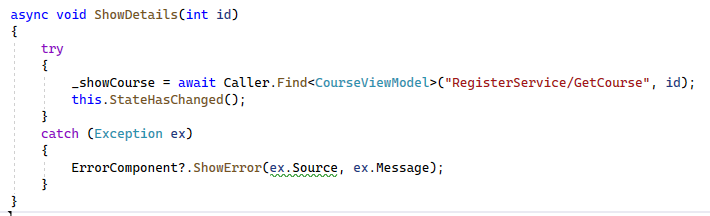


يعبر الكود السابق عن إعادة كتابة للحدث الرئيسي OnInitilized وفيه يتم استدعاء العملية GetCourses من الخدمة RegisterService والتي هي بدورها عبارة عن واجهة عبور إلى الخدمة الفعلية Register

بعد أن يتم تعبئة القائمة \_courses بالبيانات اللازمة يتم تحديث الواجهة تلقائيا وعرض المواد وذلك من خلال الكود التالي:

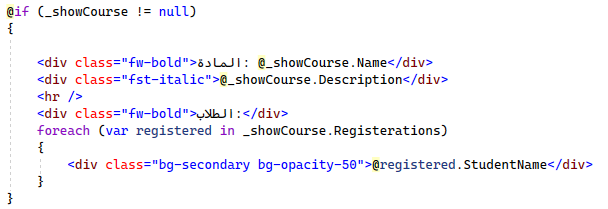


عند النقر على تفاصيل يتم تنفيذ الكود التالي

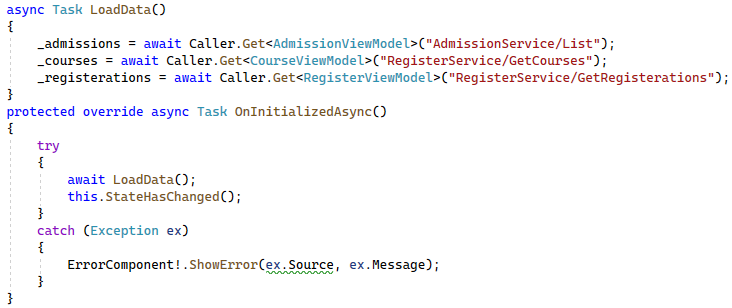


الذي يستدعي العملية GetCourse ويمرر لها id المادة المحددة

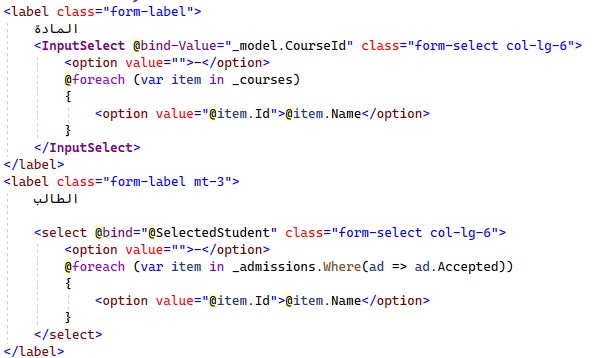
وعندما تعود النتيجة نقوم بعرضها كالتالي:



مثال آخر: في مكون التسجيل على المواد, الجزء الثاني من مكونات الصفحة Register نقوم بالاستعلام من عدة خدمات لملء البيانات في القوائم كما في الكود التالي:



بعد أن يتم انتهاء الاستدعاءات الثلاثة (من خدمات مختلفة) وملئ القوائم بالبيانات نقوم بعرضها كما في الكود التالي:



على نفس المنوال, تمت برمجة الصفحات الأخرى Attendance, Admission حيث تقسم الصفحات لمكونات, وفي كل مكون نضع الأكواد اللازمة للاستدعاء والعرض

خامسا – التشغيل

إعدادات الخدمات

أ – الاتصال بقواعد البيانات

في ملف appsettings.json في مجلد الخدمة المطلوبة يمكن تعديل الاتصال Connection String بحسب إعدادات الأجهزة لديك كما في الصورة المرفقة



ب – إعدادات مخدم الخدمة

يمكن تعديل المخدم والمنفذ للخدمة أيضا من خلال الملف appsettings.json كالتالي:



إعدادات نقطة العبور

أ – إعدادات المخدم والمنفذ

من خلال الملف appsettings.json يمكن أن يتم تعديلها. كما في كل الخدمات يمكن تعديل القسم Kestrel

ب – إعدادات التوجيه

من خلال الملف ocelot.json يمكن تعديل طريقة التحويل لكل خدمة, كما يجب الانتباه أن BaseUrl في هذا الملف يجب أن يتطابق مع المخدم والمنفذ في appsettings.json وإلا سوف تحصل أخطاء غير متوقعة

يحوي الملف على قسمين

* قسم GlobalConfiguration : يحوي على مسار المخدم الرئيسي
* قسم Routes يحوي على إعدادات التوجيه لكل خدمة

كل خدمة لها الإعدادات التالية:

* DownstreamPathTemplate : مسار الخدمة الفعلية ضمن مخدمها (بدون اسم المخدم ومنفذ)
* DownstreamScheme : نوع بروتوكول الاتصال بالخدمة http, https
* DownstreamHostAndPorts : اسم مخدم الخدمة والمنفذ الذي يعمل به فعليا
* UpstreamPathTemplate : مسار الخدمة الجديد (البديل عن مسارها الأصلي) حيث يدمج مع BaseUrl في قسم الإعدادات العامة
* UpstreamHttpMethod: أنواع الطلبات المسموح بعبورها وتحويلها إلى الخدمة

إعدادات الواجهة

أ – إعداد المخدم والمنفذ

كما في الخدمات السابقة من خلال الإعداد Kestrel في ملف appsettings.json

ب – إعداد عنوان الربط مع Gateway

ويفيد في حال قام المستخدم بتغيير مكان وجود Gateway فيتم ضبطه ليوافق اسم المخدم ومنفذه التي تعمل عليه خدمة العبور

يتم ذلك من خلال الإعداد Gateway Address في الملف appsettings.json



تشغيل التجربة

في مجلد Executables المرفق توجد كل الخدمات (كل خدمة بمجلد منفصل) بعد أن قمنا بإصدارها Release بحيث أصبحت جاهزة للاستثمار.

يمكن أن يتم تشغيل كافة الخدمات مباشرة من خلال الملف RunAll.cmd الذي يحوي تعليمات لتشغيل الخدمات عوضا عن الدخول إلى مجلد كل خدمة وتشغيلها يدويا.

--- انتهى ---

1. [Designing a DDD-oriented microservice - .NET | Microsoft Learn](https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice) [↑](#footnote-ref-1)
2. [ASP.NET Core fundamentals overview | Microsoft Learn](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-7.0&tabs=windows#programcs) [↑](#footnote-ref-2)
3. [Getting Started — Ocelot 1.0.0 documentation](https://ocelot.readthedocs.io/en/latest/introduction/gettingstarted.html) [↑](#footnote-ref-3)
4. [ASP.NET Core Blazor | Microsoft Learn](https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0) [↑](#footnote-ref-4)