

Laboratorul 6

Structura aplicatiei ASP.NET Core MVC

O bună înțelegere a ceea ce se află în spatele MVC ne poate ajuta să punem în context caracteristicile aplicațiilor ASP.NET Core MVC. Termenul model-view-controller a fost folosit de la sfârșitul anilor 1970 și a apărut în proiectul Smalltalk de la Xerox PARC, unde a fost conceput ca o modalitate de organizare a unor aplicații GUI timpurii. O parte din detaliile fine ale modelului MVC original a fost legată de concepte specifice Smalltalk, cum ar fi ecrane și instrumente, dar conceptele mai largi sunt încă aplicabile aplicațiilor și sunt deosebit de potrivite pentru aplicațiile web.

În termeni generali, modelul MVC presupune că o aplicație va fi împărțită în cel puțin trei bucăți.

- Models, care conțin sau reprezintă datele cu care utilizatorii lucrează și operațiunile care se pot efectua asupra acestor date.
- Views, care redă parțial sau total modelul ca interfață cu utilizatorii
- Controlerele, care procesează cererile primite, declanșază operațiuni pe model și selectează view-urile corespunzătoare necesare pentru a redă utilizatorului răspunsurile la solicitări.

Fiecare piesă din arhitectura MVC este bine definită și de sine stătătoare, ceea ce conduce la separarea preocupărilor. Logica care manipulează datele din model este conținută doar în model, logica care afișează datele este doar în vizualizare, iar codul care gestionează solicitările utilizatorului este conținută numai în controler. O astfel de împărțire clară între fiecare dintre piesele aplicației conduce la aplicații mai ușor de întreținut și de extins pe parcursul vieții, indiferent de cât de mare va deveni.

Models. Modelele reprezintă definiția universului în care funcționează aplicația dvs. Într-o aplicație bancară, de exemplu, modelul reprezintă toate conceptele acceptate într-o bancă, cum ar fi conturile, evidența generală și limitele de credit pentru clienți, precum și operațiuni care pot fi utilizate pentru manipularea datelor din model, cum ar fi depunerea de fonduri și efectuarea retragerilor din conturi. Modelul este, de asemenea, responsabil pentru păstrarea stării generale și a coerenței datelor - de exemplu, trebuie să asigure faptul că toate tranzacțiile sunt înregistrate corect, că un client nu retrage mai mulți bani decât are dreptul sau mai mulți bani decât are în cont.

Modelul dintr-o aplicație MVC trebuie să:

- Conțină datele de domeniu
- Conțină logica pentru crearea, gestionarea și modificarea datelor de domeniu

Modelul nu ar trebuie sa:

- Expuna detalii despre cum sunt obtinute sau gestionate datele modelului (cu alte cuvinte, detaliile mecanismului de stocare a datelor nu trebuie expuse controlerelor si View-urilor)
- Conțina logică pentru interacțiunea cu utilizatorii (aceasta este sarcina View-urilor)

Mulți dezvoltatori neavizati pentru modelul MVC, considera că obiectivul modelului MVC este de a separa datele de logică. Aceasta este o înțelegere greșită. Scopul modelului MVC este divizarea unei aplicații în trei zone funcționale, fiecare dintre ele putând conține atât logică cât și date. Scopul nu este eliminarea logicii din model. Mai degrabă, trebuie să ne asigurăm că modelul conține doar logica pentru crearea și gestionarea datelor modelului.

Controllers. Controlerele sunt țesutul conjunctiv în modelul MVC, acționând ca un mediator între model și View-uri. Controlerele definesc acțiunile care oferă logica de afaceri care operează pe modelul de date și care furnizează datele pe care View-urile le afișează utilizatorului.

Un controler construit folosind modelul MVC ar trebui sa:

- Conțina acțiunile necesare pentru efectuarea de operatii asupra modelului bazate pe solicitarile utilizatorilor.

Controlerul nu ar trebui sa:

- Conțina logică care gestionează expunerea datelor pentru utilizatori (aceasta este treaba View-urilor)
- Conțina logică care gestionează persistența datelor (aceasta este treaba modelului)

Views. View-urile conțin logica necesară pentru afișarea datelor către utilizator sau pentru capturarea datelor de la acesta, astfel încât acestea să poată fi procesate printr-o acțiune a controlorului.

Vizualizările ar trebui sa:

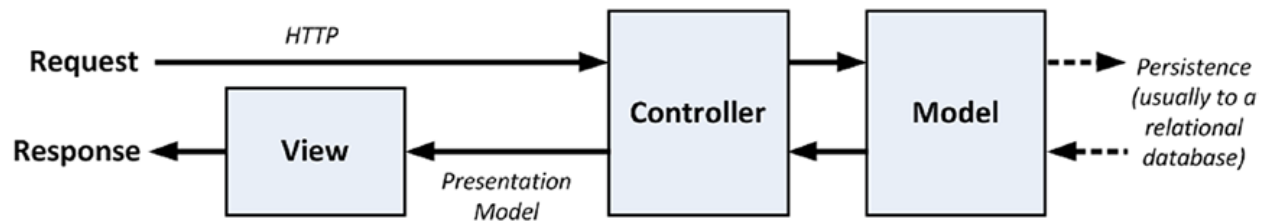
- Conțina logica și mecanismele necesare pentru a prezenta datele utilizatorului

Vizualizările nu ar trebui sa:

- Conțina logică complexă (aceasta trebuie plasată într-un controler)
- Conțina logică care creează, stochează sau manipulează modelul de domeniu. Vizualizările pot conține logică, dar ar trebui să fie simplă și ușor de utilizat.

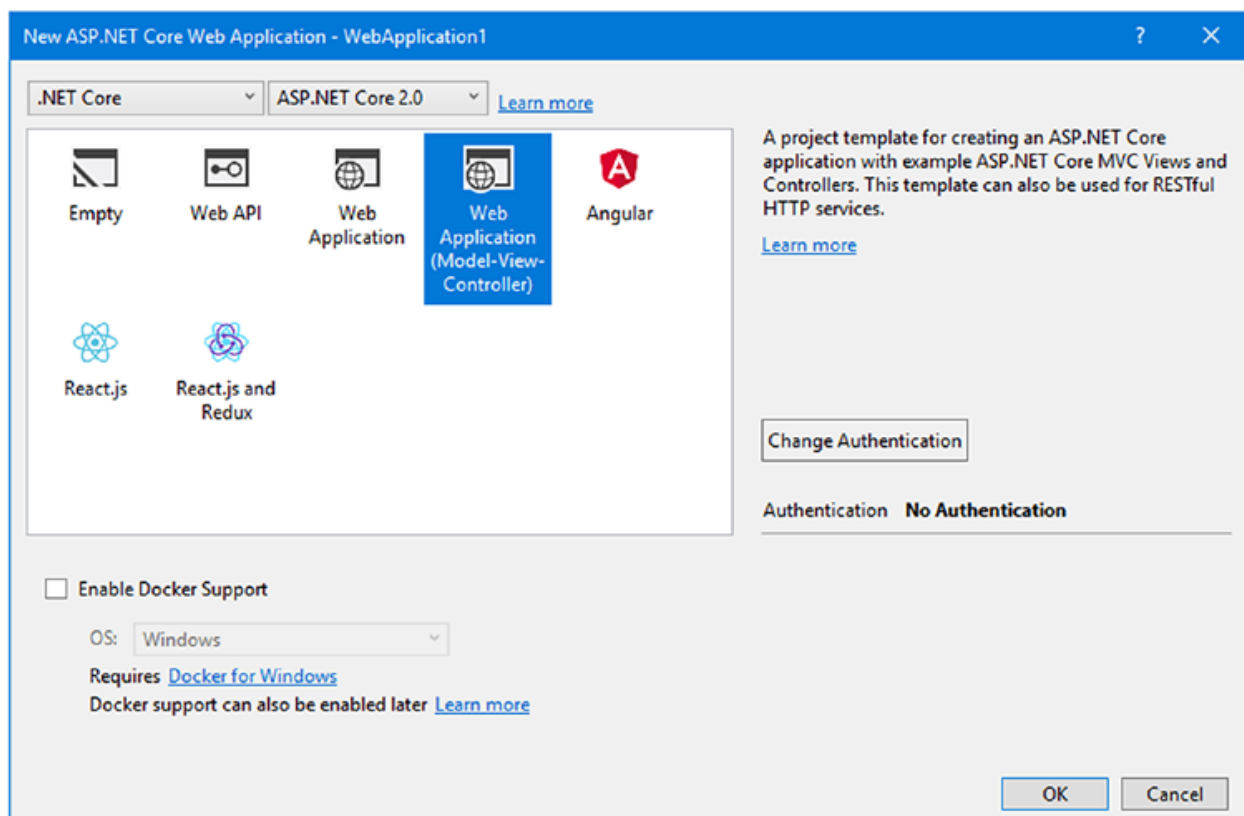
Implementarea MVC in ASP.NET. ASP.NET Core MVC adaptează modelul abstract MVC la lumea dezvoltatorilor ASP.NET și C#. În ASP.NET Core MVC, controlerele sunt clase C#, de obicei derivate de la `Microsoft.AspNetCore.Mvc.Controller`.

Fiecare metodă publică dintr-o clasă derivată din Controller este o acțiune, care este asociată cu o adresă URL. Când o cerere este trimisă la adresa URL asociată unei metode de acțiune, declarațiile din acea metodă de acțiune sunt executate pentru a efectua o anumită operație pe modelul de domeniu și apoi pentru a selecta o vizualizare pentru a fi afișată clientului. Figura de mai jos arată interacțiunile dintre Controller, Model și View.



ASP.NET Core MVC utilizează un motor de vizualizare, cunoscut sub numele de Razor, care este componenta responsabilă de procesarea unui View pentru a genera un răspuns pentru browser. View-urile Razor sunt șabloane HTML care conțin logica C#, care este utilizată pentru a prelucra datele modelului cu scopul de a genera conținut dinamic care răspunde modificărilor modelului.

Proiectul ASP.NET Core MVC. Când creați un nou proiect ASP.NET Core MVC, Visual Studio vă oferă câteva opțiuni despre conținutul inițial pe care îl doriți în proiect așa cum putem vedea în imaginea de mai jos. Ideea este de a ușura procesul de învățare pentru noii dezvoltatori și de a aplica unele bune practici de economisire a timpului pentru funcții și sarcini comune.



Modelul proiectului Empty conține instalări pentru ASP.NET Core, dar nu include bibliotecile sau configurația necesară pentru o aplicație MVC. Șablonul de proiect Web API include ASP.NET Core și MVC, cu o aplicație de exemplu care arată cum să primiți și să procesați solicitările de date de la clienți, folosind un controler API.

Șablonul de proiect Web Application (Model-View-Controller) include ASP.NET Core și MVC, cu o aplicație de exemplu care demonstrează modul de generare a conținutului HTML. Șabloanele API și Web Application (Model-View-Controller) pot fi configurate cu scheme diferite pentru autentificarea utilizatorilor și autorizarea accesului acestora la aplicație.

Celelalte șabloane oferă conținut inițial potrivit pentru a lucra cu cadre de aplicație cu o singură pagină (Angular și React) și cu Pagini Razor (care permite amestecarea de cod și markup în același fișier)

Adevărata diferență între șabloanele proiectului este setul inițial de biblioteci, fișiere de configurare, cod și conținut pe care Visual Studio le adaugă atunci când creează proiectul. Indiferent de șablonul pe care îl utilizați pentru a crea un proiect, vor apărea unele foldere și fișiere comune.

Unele dintre elementele unui proiect au roluri speciale care sunt codate în ASP.NET Core sau MVC sau într-unul dintre instrumentele pentru care Visual Studio oferă suport. Altele sunt doar

convenții de denumire utilizate în majoritatea proiectelor ASP.NET Core sau MVC. Descriem pe scurt fișierele și folderele importante pe care le veți întâlni într-un proiect ASP.NET Core MVC.

/Areas - Areas sunt un mod de a împărți o aplicație mare în bucăți mai mici.

/Dependencies - Elementul Dependencies oferă detalii despre toate pachetele pe care se bazează un proiect.

/Components - Aici sunt definite clase componente de vizualizare, care sunt utilizate pentru a afișa caracteristici auto-conținute, cum ar fi coșurile de cumpărături.

/Controllers - Aici se vor pune clasele de Controller. Aceasta este o convenție. Putem pune clasele Controller oriunde dorim, deoarece toate sunt compilate în același ansamblu.

/Data - Aici sunt definite clasele context ale bazei de date.

/Data/Migrations - Aici sunt stocate mecanismele de migrare ale bazei de date Entity Framework Core, astfel încât bazele de date pot fi pregătite pentru a stoca datele aplicației.

/Models - Acesta este locul în care punem clasele de model. Aceasta este o convenție. Putem defini clasele model oriunde în proiect sau într-un proiect separat.

/Views - Acest director conține View-urile, de obicei grupate în foldere numite după controlerul cu care sunt asociate.

/Views/Shared - Acest director conține machete și View-uri care nu sunt specifice unui singur controller.

/Views/_ViewImports.cshtml - Acest fișier este utilizat pentru a specifica spațiile de nume care vor fi incluse în fișierele de vizualizare Razor.

/Views/_ViewStart.cshtml - Acest fișier este utilizat pentru a specifica un layout implicit pentru motorul de vizualizare Razor.

/appsettings.json - Acest fișier conține setări de configurare care pot fi adaptate pentru diferite medii, cum ar fi dezvoltarea, testarea și producția. Cele mai frecvente utilizări pentru acest fișier sunt definirea șirurilor de conexiune la serverul de baze de date și setările de logare / debug.

/bower.json - Acest fișier conține lista de pachete gestionate de managerul de pachete Bower.

/<project>**.csproj** - Acest fișier conține configurația pentru proiect, incluzând pachetele NuGet necesare aplicației.

/Program.cs - Această clasă configurează platforma de găzduire pentru aplicație.

/Startup.cs - Această clasă configurează aplicația.

/wwwroot - Aici punem conținut static, cum ar fi fișiere CSS și imagini. Este, de asemenea, locul în care managerul de pachete Bower instalează pachete JavaScript și CSS.

Convențiile MVC. Există două tipuri de convenții într-un proiect MVC. Primul tip este constituit doar din sugestii cu privire la modul în care am putea dori să ne structurăm proiectul. De exemplu, este convențional să punem pachetele JavaScript și CSS în folderul `wwwroot / lib`. Acesta este locul în care alți dezvoltatori MVC s-ar aștepta să le găsească și unde le va instala managerul de pachete. Dar sunteți liber să redenumiți folderul `lib` sau să îl eliminați în întregime și să puneți pachetele în altă parte.

Celălalt tip de convenție decurge din principiul convenției asupra configurației. Convenția privind configurația înseamnă că nu este necesar să configurați în mod explicit asocierile între Controller și View-urile acestora, de exemplu, urmăm doar o anumită convenție de denumire pentru fișiere și totul funcționează. Există mai puțină flexibilitate în schimbarea structurii proiectului atunci când aveți de-a face cu acest tip de convenție.

Convenții pentru clasele Controller. Clasele Controller au nume care se termină cu Controller, cum ar fi `ProductController`, `AdminController` și `HomeController`. Când facem referire la un controler din altă parte a proiectului, specificăm prima parte a numelui (cum ar fi `Prodct`), iar MVC adaugă automat Controller la nume și începe să caute clasa `ProductController`.

Convențiile pentru View. View-urile sunt în folderul numit `/Views / <Numele controlului>`. De exemplu, un View asociat cu clasa `ProductController` ar fi în folderul `/ Views / Product`.

MVC se așteaptă ca View-ul implicit pentru o metodă de acțiune să fie numit după acea metodă. De exemplu, vizualizarea implicită asociată unei metode de acțiune numită `List` ar trebui numită `List.cshtml`.

Astfel, pentru metoda de acțiune `List` din clasa `ProductController`, vizualizarea implicită este de așteptat să fie `/ Views / Product / List.cshtml`. Vizualizarea implicită este utilizată când returnați rezultatul apelării metodei `View` într-o metodă de acțiune, astfel:

```
...  
return View();  
...
```

Putem specifica un View diferit după nume, astfel:

```
...  
return View("MyOtherView");  
...
```

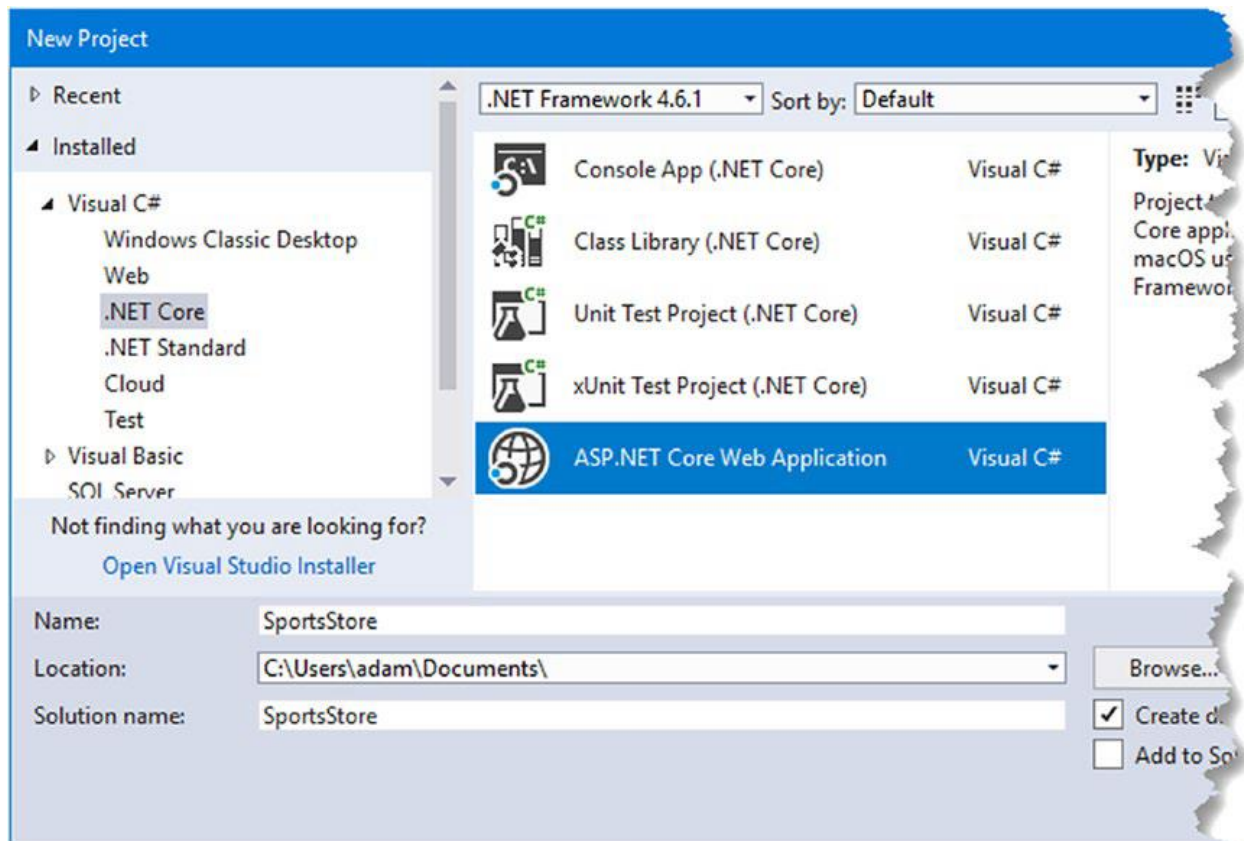
Convenții pentru Layouts. Convenția de denumire pentru Layouts este de a prefixa fișierul cu un caracter de subliniere (`_`), iar fișierele de Layouts sunt plasate în folderul `/Views /Shared`.

Acest layout este aplicat în mod implicit tuturor View-urilor prin fișierul `/Views/_ViewStart.cshtml`.

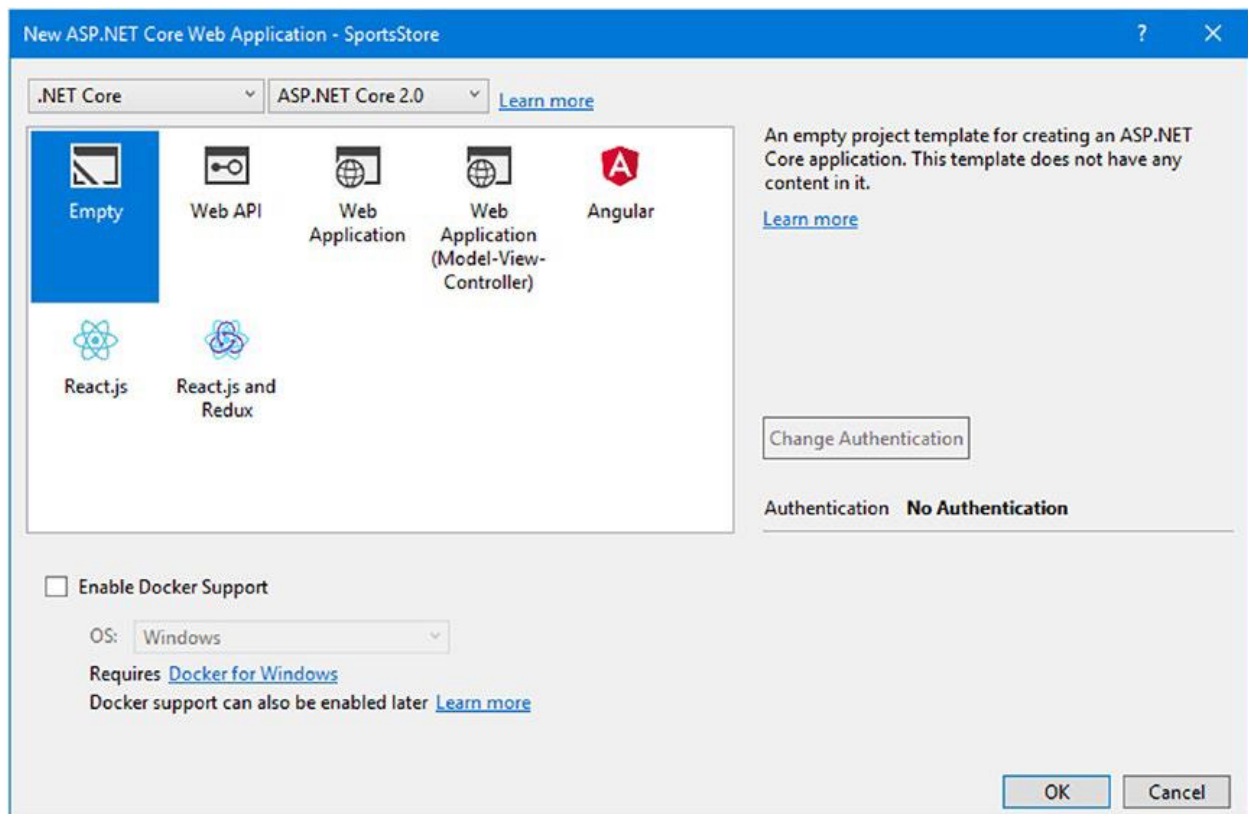
SportsStore: A Real Application. Acum vom construi o aplicație de comerț electronic simplă, numită SportsStore, care va urma abordarea clasică adoptată de magazinele online de pretutindeni.

Vom crea un catalog de produse online pe care clienții îl pot accesa după categorie și pagină. De asemenea, vom crea o zonă de administrare care include facilități de creare, citire, actualizare și ștergere (CRUD) pentru gestionarea catalogului.

Crearea proiectului MVC. În scop didactic vom crea un proiect Empty și vom adauga toate fișierele de configurare și componentele de care avem nevoie. Am început selectând New -> Project din meniul Visual Studio File și selectând șablonul de proiect web Application ASP.NET Core, așa cum se arată în figura de mai jos. Am setat numele proiectului ca SportsStore și am dat clic pe butonul OK.



Am selectat șablonul Empty, așa cum se arată în figura de mai jos. M-am asigurat că .NET Core și ASP.NET Core au fost selectate în meniurile din partea de sus a ferestrei de dialog și că opțiunea Enable Docker Support a fost debifată înainte de a face clic pe butonul OK pentru a crea proiectul SportsStore.



Crearea structurii de foldere. Următorul pas este să adăugați folderele care vor conține componentele aplicației necesare pentru o aplicație MVC: Models, Controllers și View. Fișiere suplimentare vor fi necesare mai târziu, dar acestea reflectă principalele părți ale aplicației MVC și sunt suficiente pentru a începe.

Models- Acest folder va conține clasele de model.

Controllers - Acest folder va conține clase de controler.

Views - Acest folder conține tot ce se referă la View-uri, inclusiv fișiere Razor individuale.

Configurarea aplicației. Clasa Startup este responsabilă pentru configurarea aplicației Core ASP.NET. Exemplul 1 arată modificările pe care le-am făcut la clasa Startup pentru a activa Framework-ul MVC și unele caracteristici conexe care sunt utile pentru dezvoltare. Clasa Startup este o caracteristică importantă a aplicațiilor ASP.NET Core.

Exemplul 1. Enabling Features in the Startup.cs File in the SportsStore Folder

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Threading.Tasks;  
using Microsoft.AspNetCore.Builder;  
using Microsoft.AspNetCore.Hosting;  
using Microsoft.AspNetCore.Http;
```



```

using Microsoft.Extensions.DependencyInjection;
namespace SportsStore {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                });
        }
    }
}

```

Metoda **ConfigureServices** este utilizată pentru a configura obiecte partajate care pot fi utilizate în întreaga aplicație prin funcția de injecție de dependență.

Metoda **AddMvc** pe care o apelăm în metoda ConfigureServices este o metodă de extensie care configurează obiectele partajate utilizat în aplicațiile MVC.

Metoda **Configure** este utilizată pentru a configura funcțiile care primesc și procesează cererile HTTP. Fiecare metodă pe care o apelăm în metoda Configure este o metodă de extensie care stabilește un process legat de solicitarea HTTP, așa cum este descris mai jos.

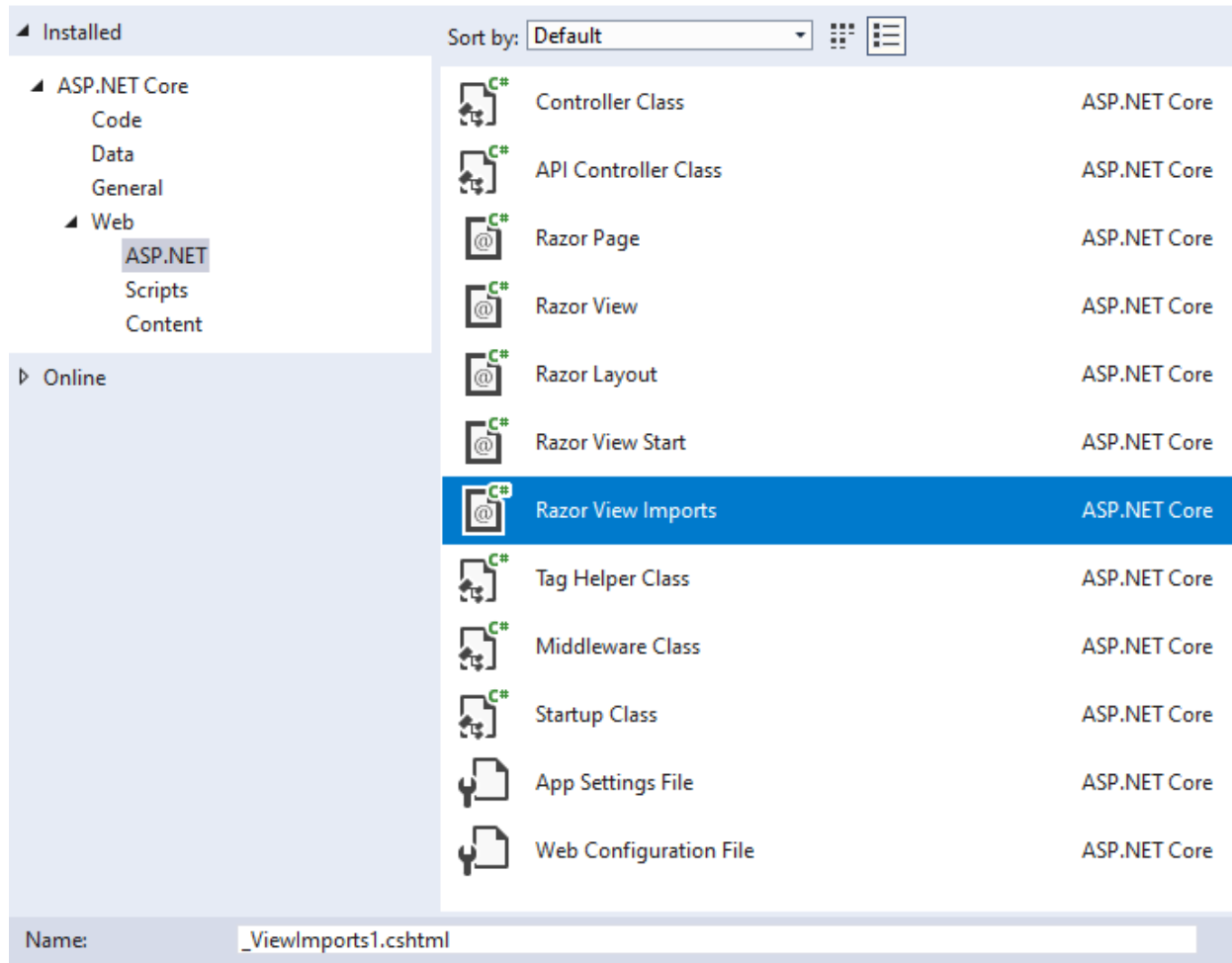
UseDeveloperExceptionPage() - Această metodă de extensie afișează detalii despre excepțiile care apar în aplicație, care sunt utile în timpul procesului de dezvoltare. Nu trebuie activată în aplicațiile implementate.

UseStatusCodePages() - Această metodă de extensie adaugă un mesaj simplu la răspunsurile HTTP care altfel nu ar avea un body, cum ar fi 404 - Not Found responses.

UseStaticFiles() - Această metodă de extensie permite asistență pentru servirea conținutului static din folderul wwwroot.

UseMvc() - Această metodă de extensie activează ASP.NET MVC Core.

În continuare, trebuie să pregătim aplicația pentru View-uri Razor. Faceți clic dreapta pe folderul Views, selectați Add -> New Item din meniul pop-up și selectați articolul Razor View Imports din categoria ASP.NET Core, așa cum se arată mai jos.



Faceți clic pe butonul Add pentru a crea fișierul `_ViewImports.cshtml` și modificați conținutul noului fișier ca în Exemplul 2.

Exemplul 2. The Contents of the `_ViewImports.cshtml` File in the Views Folder

@using SportsStore.Models

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

Instrucțiunea `@using` îmi va permite să folosesc tipurile din spațiul de nume `SportsStore.Models` în View-uri, fără a fi necesar să mă refer la spațiul de nume. Instrucțiunea `@addTagHelper` permite tag_helpers încorporate, pe care le vom folosi ulterior pentru a crea elemente HTML care reflectă configurația aplicației `SportsStore`.

Definirea modelului de domeniu. Toate proiectele încep cu modelul de domeniu, care este inima unei aplicații MVC. Întrucât aceasta este o aplicație de comerț electronic, cel mai evident model de care am nevoie este un produs. Am adăugat un fișier de clasă numit `Product.cs` în folderul `Models` și l-am folosit pentru a defini clasa afișată în Exemplul 3.

Exemplul 3. The Contents of the Product.cs File in the Models Folder

```
namespace SportsStore.Models {  
    public class Product {  
        public int ProductID { get; set; }  
        public string Name { get; set; }  
        public string Description { get; set; }  
        public decimal Price { get; set; }  
        public string Category { get; set; }  
    }  
}
```

Creating a Repository

Crearea unui Repository. Am nevoie de un mecanism de a obține obiecte Product dintr-o bază de date. Modelul include logica de stocare și preluare a datelor din depozitul de date persistente. Nu ne vom îngriji cu privire la modul în care vom implementa persistența datelor pentru în prima fază, dar vom începe procesul de definire a unei interfețe pentru aceasta. Am adăugat un nou fișier de interfață C#, numit IProductRepository.cs în folderul Models și l-am folosit pentru a defini interfața prezentată în Exemplul 4.

Exemplul 4. The Contents of the IProductRepository.cs File in the Models Folder

```
using System.Linq;  
namespace SportsStore.Models {  
    public interface IProductRepository {  
        IQueryable<Product> Products { get; }  
    }  
}
```

Această interfață folosește IQueryable <T> pentru a permite unui apelant să obțină o secvență de obiecte Product. Interfața IQueryable <T> este derivată din interfața IEnumerable <T> mai familiară și reprezintă o colecție de obiecte care pot fi interogate, precum cele gestionate de o bază de date. O clasă care depinde de interfața IProductRepository poate obține obiecte Product fără a fi nevoie să cunoască detaliile modului în care sunt stocate sau cum le va livra clasa de implementare.

Interfața IQueryable <T> este utilă, deoarece permite interogarea eficientă a unei colecții de obiecte. Interfeței IQueryable <T> ne permite să selectăm din baza de date doar obiectele de care avem nevoie folosind instrucțiuni LINQ standard și fără a fi nevoie să știm ce server de baze de date stochează datele sau modul în care procesează interogarea. Fără interfața IQueryable <T>, ar trebui să extrag toate obiectele Product din baza de date și apoi să le eliminăm pe cele pe care nu le dorim, ceea ce devine o operație scumpă pe măsură ce cantitatea de date utilizată de o aplicație crește. Din acest motiv, interfața IQueryable <T> este utilizată de obicei în loc de IEnumerable <T> în interfețele și clasele depozitelor de baze de date.

Crearea unui depozit fals. Acum că am definit o interfață, am putea să implementăm mecanismul de persistență și să îl conectăm la o bază de date, dar vrem să adăugăm mai întâi câteva dintre celelalte părți ale aplicației. Pentru a face acest lucru, vom crea o implementare falsă a interfeței `IProductRepository`, care va rămâne valabilă până când vom crea adevăratul mecanism de stocare a datelor. Pentru a crea depozitul fals, am adăugat un fișier de clasă numit `FakeProductRepository.cs` în folderul `Models` și l-am folosit pentru a defini clasa prezentată în Exemplul 5.

Exemplul 5. The Contents of `FakeProductRepository.cs` File in the `Models` Folder

```
using System.Collections.Generic;
using System.Linq;
namespace SportsStore.Models {
    public class FakeProductRepository : IProductRepository {
        public IQueryable<Product> Products => new List<Product> {
            new Product { Name = "Football", Price = 25 },
            new Product { Name = "Surf board", Price = 179 },
            new Product { Name = "Running shoes", Price = 95 }
        }.AsQueryable<Product>();
    }
}
```

Clasa `FakeProductRepository` implementează interfața `IProductRepository` returnând o colecție fixă de obiecte de produs ca valoare a proprietății `Products`. Metoda `AsQueryable` este folosită pentru a converti colecția fixă de obiecte într-un `<Product> IQueryable`, care este necesar pentru a implementa interfața `IProductRepository` și ne permite să cream un depozit fals.

Înregistrarea Repository Service. MVC subliniază utilizarea unor componente cuplate ușor, ceea ce înseamnă că puteți face o schimbare într-o parte a aplicației fără a fi nevoie să faceți modificări corespunzătoare în altă parte. Această abordare clasifică părțile din aplicație drept servicii, care oferă caracteristici pe care le utilizează alte părți ale aplicației. Clasa care furnizează un serviciu poate fi apoi modificată sau înlocuită fără a necesita modificări în clasele care îl utilizează.

Pentru aplicația `SportsStore`, vrem să cream un serviciu **Repository**, care să permită controlerelor să obțină obiecte care implementează interfața `IProductRepository` fără a ști ce clasă este utilizată. Acest lucru îmi va permite să încep dezvoltarea aplicației folosind clasa

`FakeProductRepository` simplă creată și apoi să o înlocuiesc cu un depozit real ulterior, fără a fi nevoie să fac modificări în toate clasele care au nevoie de acces la depozit.

Serviciile sunt înregistrate în metoda `ConfigureServices` din clasa `Startup`, iar în Exemplul 6, am definit un nou serviciu `Repository`.

Exemplul 6. Creating the Repository Service in the `Startup.cs` File in the `SportsStore` Folder

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
namespace SportsStore {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IProductRepository, FakeProductRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
            });
        }
    }
}

```

Declarația pe care am adăugat-o la metoda ConfigureServices spune ASP.NET Core că atunci când o componentă, cum ar fi un controler, are nevoie de o implementare a interfeței IProductRepository, ar trebui să primească o instanță a clasei FakeProductRepository. Metoda AddTransient specifică faptul că un nou obiect FakeProductRepository trebuie creat de fiecare dată când este nevoie de interfața IProductRepository.

Adăugarea unui controller. Pentru a crea primul controler din aplicație, am adăugat un fișier de clasă numit ProductController.cs în folderul Controllers și am definit clasa prezentată în Exemplul 7.

Exemplul 7. The Contents of the ProductController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
namespace SportsStore.Controllers {
    public class ProductController : Controller {
        private IProductRepository repository;
        public ProductController(IProductRepository repo) {
            repository = repo;
        }
    }
}

```

```
}
```

Când MVC trebuie să creeze o nouă instanță a clasei `ProductController` pentru a gestiona o solicitare HTTP, va inspecta constructorul și va vedea că necesită un obiect care implementează interfața `IProductRepository`. Pentru a determina ce clasă de implementare ar trebui utilizată, MVC consultă configurația din clasa `Startup`, care îi spune că ar trebui să fie utilizată clasa `FakeRepository` și că o nouă instanță ar trebui creată de fiecare dată. MVC creează un nou obiect `FakeRepository` și îl folosește pentru a invoca constructorul `ProductController` pentru a crea obiectul controler care va procesa cererea HTTP.

Acest mecanism este cunoscut sub numele de **injecție de dependență**, iar abordarea sa permite constructorului `ProductController` să acceseze depozitul aplicației prin interfața `IProductRepository`, fără a fi nevoie să știe ce clasă de implementare a fost configurată. Mai târziu, vom înlocui depozitul fals cu unul real, iar injecția de dependență face ca controller-ul să continue să funcționeze fără modificări.

În continuare, am adăugat o metodă de acțiune, numită `List`, care va oferi o vizualizare care arată lista completă a produselor din depozit, așa cum se arată în Exemplul 8.

Exemplul 8. Adding an Action Method in the `ProductController.cs` File in the `Controllers` Folder using `Microsoft.AspNetCore.Mvc`;

using `SportsStore.Models`;

```
namespace SportsStore.Controllers {  
    public class ProductController : Controller {  
        private IProductRepository repository;  
        public ProductController(IProductRepository repo) {  
            repository = repo;  
        }  
        public IActionResult List() => View(repository.Products);  
    }  
}
```

Apelarea la această metodă `View` (fără a specifica un nume de `View`) spune MVC să redea `View`-ul implicit pentru metoda de acțiune. Pasarea colecției de obiecte `Product` din depozit la metoda `View` oferă Framework-ului datele cu care va popula obiectul `Model`.

Adăugarea și configurarea `View`-ului. Trebuie să creez o vizualizare pentru a prezenta conținutul utilizatorului, dar sunt necesari câțiva pași pregătitori care vor simplifica acest lucru. Prima este crearea unui layout partajat care va defini conținut comun care va fi inclus în toate răspunsurile HTML trimise clienților.

Layout-urile partajate sunt o modalitate utilă de a vă asigura că `View`-urile sunt consecvente și conțin fișiere JavaScript importante și fișiere de stil CSS.

Am creat folderul Views/Shared și i-am adăugat o nouă MVC View Layout Page numită `_Layout.cshtml`, care este numele implicit pe care Visual Studio îl atribuie acestui tip de element. Exemplul 9 arată fișierul `_Layout.cshtml`. Am făcut o modificare la conținutul implicit, care specifica conținutul elementului titlu SportsStore.

Exemplul 9. The Contents of the `_Layout.cshtml` File in the Views/Shared Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>

</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

În continuare, trebuie să configurez aplicația astfel încât fișierul `_Layout.cshtml` să fie aplicat în mod implicit. Acest lucru se face prin adăugarea unui fișier MVC View Page Start, numit `_ViewStart.cshtml` în folderul Views. Conținutul implicit adăugat de Visual Studio, afișat în Exemplul 10, selectează un layout numit `_Layout.cshtml`, care corespunde fișierului afișat în Exemplul 9.

Exemplul 10. The Contents of the `_ViewStart.cshtml` File in the Views Folder

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

Acum trebuie să adaugăm View-ul care va fi afișat atunci când se folosește metoda de acțiune List pentru a gestiona o solicitare. Am creat folderul Views / Product și i-am adăugat un fișier View Razor numit `List.cshtml`. Apoi am adăugat conținutul afișat în Exemplul 11.

Exemplul 11. The Contents of the `List.cshtml` File in the Views/Product Folder

```
@model IEnumerable<Product>
@foreach (var p in Model) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

AICI

Expresia @model din prima linie a fișierului specifică faptul că View-ul va primi o secvență de obiecte Product din metoda de acțiune ca date ale modelului său. Folosim apoi o expresie @foreach pentru a genera un set simplu de elemente HTML pentru fiecare obiect Product primit.

Setarea rutei implicit. Trebuie să-i spunem MVC că ar trebui să trimită solicitări care sosesc pentru adresa URL rădăcină a aplicației noastre (http: // mysite /) la metoda de acțiune List din clasa ProductController. Facem acest lucru prin editarea declarației din clasa Startup care stabilește clasele MVC care gestionează solicitările HTTP, așa cum se arată în Exemplul 12.

Exemplul 12. Changing the Default Route in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
namespace SportsStore {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IProductRepository, FakeProductRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}
```

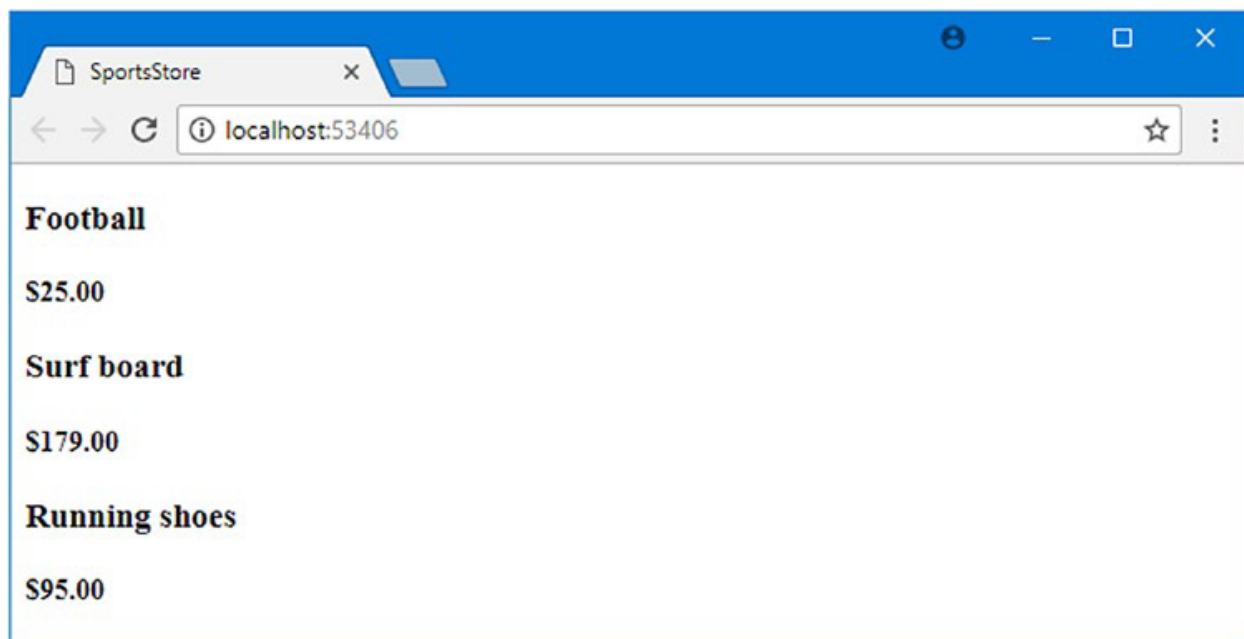
Metoda Configure a clasei de Startup este utilizată pentru a configura secvența de solicitare, care constă din clase (cunoscute sub numele de middleware) care vor inspecta cererile HTTP și vor genera răspunsuri. Metoda UseMvc configurează middleware-ul MVC, iar una dintre opțiunile de configurare este schema care va fi utilizată pentru maparea adreselor URL către

controlere și metode de acțiune. Opțiunea de rutare din Exemplul 12 face ca MVC să trimită implicit solicitările către metoda de acțiune List a controlorului Product, cu excepția cazului în care URL-ul cererii specifică altceva.

Observați că am setat numele controlerului în Exemplul 12 ca Product și nu ProductController, care este numele clasei. Aceasta face parte din convenția de denumire MVC, în care numele claselor controlerului se termină în general în Controller, dar se omite această parte a numelui atunci când se face referire la clasă.

Rularea aplicației. Toate elementele de bază ale aplicației sunt construite. Avem un controler cu o metodă de acțiune pe care MVC o va folosi atunci când se solicită adresa URL implicită pentru aplicație. MVC va crea o instanță a clasei FakeRepository și o va folosi pentru a crea un nou obiect controler pentru a gestiona cererea. FakeRepository va furniza controlerului câteva date de testare simple, pe care metoda de acțiune a acestuia le pasează la View Razor, astfel încât răspunsul HTML la browser va include detalii pentru fiecare obiect Product.

La generarea răspunsului HTML, MVC va combina datele din View-ul selectat prin metoda de acțiune cu conținutul din aspectul partajat, producând un document HTML complet pe care browserul îl poate analiza și afișa. Puteți vedea rezultatul pornind aplicația, așa cum se arată în figura de mai jos. Acesta este modelul tipic de dezvoltare pentru ASP.NET Core MVC.



Pregătirea unei baze de date. Acum putem afișa o vizualizare simplă care conține detalii despre produse, dar folosește datele de test pe care le conține depozitul fals. Înainte de a putea implementa un depozit real, trebuie să configurăm o bază de date și să o populăm cu date.

Vom folosi SQL Server ca bază de date și vom accesa baza de date utilizând Entity Framework Core (EF Core), care este framework-ul ORM (Microsoft .NET object-relational mapping). Un framework ORM prezintă tabelele, coloanele și rândurile unei baze de date relaționale prin obiecte C# obișnuite.

O caracteristică a SQL Server este LocalDB, care este o implementare gratuită de administrare a caracteristicilor SQL Server de bază special concepute pentru dezvoltatori. Folosind această caracteristică, putem sari peste procesul de configurare a unei baze de date în timp ce construim proiectul și apoi să mă implementăm aplicația într-o instanță completă SQL Server.

Dacă nu ați selectat LocalDB când ați instalat Visual Studio, trebuie să faceți acest lucru acum. Poate fi selectată în secțiunea Componente individuale a programului de instalare Visual Studio.

Crearea claselor de baze de date. Database context class este puntea de legătură între aplicație și Entity Framework Core și oferă acces la datele aplicației folosind obiecte model. Pentru a crea clasa context a bazei de date pentru aplicația SportsStore, am adăugat un fișier de clasă numit ApplicationDbContextContext.cs în folderul Models și am definit clasa prezentată în Exemplul 13.

Exemplul 13. The Contents of the ApplicationDbContextContext.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.DependencyInjection;
namespace SportsStore.Models {
    public class ApplicationDbContext : DbContext {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options) { }
        public DbSet<Product> Products { get; set; }
    }
}
```

Clasa de bază DbContext oferă acces la funcționalitatea de bază a Entity Framework Core și proprietatea Products va oferi acces la obiectele Product din baza de date. Clasa ApplicationDbContext este derivată din DbContext și adaugă proprietățile care vor fi utilizate pentru a citi și scrie datele aplicației. Există o singură proprietate în acest moment, care va oferi acces la obiectele Product.

Crearea clasei Repository. Următorul pas este crearea unei clase care să implementeze interfața IProductRepository și să-și obțină datele utilizând Entity Framework Core. Am adăugat un fișier de clasă numit EFProductRepository.cs în folderul Models și l-am folosit pentru a defini clasa Repository prezentată în Exemplul 14.

Exemplul 14. The Contents of the EFProductRepository.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;
namespace SportsStore.Models {
```

```

public class EFProductRepository : IProductRepository {
    private ApplicationDbContext context;

    public EFProductRepository(ApplicationDbContext ctx) {
        context = ctx;
    }

    public IQueryable<Product> Products => context.Products;
}

```

Vom adăuga funcționalități pe măsură ce adaug funcții în aplicație, dar, pentru moment, implementarea depozitului nu face decât să mapeze proprietatea Products definită de interfața IProductRepository pe proprietatea Products definită de clasa ApplicationDbContext. Proprietatea Products din clasa context returnează un obiect DbSet <Product>, care implementează interfața IQueryable <T> și face ușoară implementarea interfeței IProductRepository atunci când folosim Entity Framework Core. Acest lucru asigură că interogările la baza de date vor prelua doar obiectele care sunt necesare.

Definirea stringului de conexiune. Un string de conexiune specifică locația și numele bazei de date și oferă setări de configurare pentru modul în care aplicația trebuie să se conecteze la serverul de baze de date. Stringurile de conexiune sunt stocate într-un fișier JSON numit appsettings.json, pe care l-am creat în proiectul SportsStore folosind șablonul de element de fișier de configurare ASP.NET din secțiunea Generală a ferestrei Add New Item.

Visual Studio adaugă un placeholder connection string în fișierul appsettings.json atunci când creează fișierul, pe care l-am înlocuit în Exemplul 15. Stringurile de conexiune trebuie să fie specificate într-o singură linie neîntreruptă.

Exemplul 15. Editing the Connection String in the appsettings.json File in the SportsStore Folder

```

{
  "Data": {
    "SportStoreProducts": {
      "ConnectionString": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStoreMVC;Trusted_Connection=True;MultipleActiveResultSets=true"
    }
  }
}

```

În secțiunea Date din fișierul de configurare, am setat numele șirului de conexiune la SportsStoreProducts. Valoarea elementului ConnectionString specifică faptul că funcția LocalDB trebuie utilizată pentru o bază de date numită SportsStoreMVC.

Configurarea aplicației. Următorii pași sunt să citim stringul de conexiune și să configurăm aplicația pentru a-l utiliza în vederea conectării la baza de date. Exemplul 16 arată modificările necesare clasei Startup pentru a accesa detalii despre datele de configurare conținute în fișierul

appsettings.json și utilizarea acestora pentru a configura Entity Framework Core. Partea de citire a fișierului JSON este gestionată de clasa Program.

Exemplul 16. Configuring the Application in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
namespace SportsStore {
    public class Startup {
        public Startup(IConfiguration configuration) =>
            Configuration = configuration;
        public IConfiguration Configuration { get; }
        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseMvc(routes => {
                routes.MapRoute(
                    name: "default",
                    template: "{controller=Product}/{action=List}/{id?}");
            });
        }
    }
}
```

Constructorul adăugat la clasa Startup primește datele de configurare încărcate din fișierul appsettings.json, care este prezentat printr-un obiect care implementează interfața IConfiguration. Constructorul atribuie obiectul IConfiguration unei proprietăți numite Configuration, astfel încât să poată fi folosit de restul clasei Startup.

Pentru aplicația SportsStore, am adăugat o secvență de apeluri de metodă care setează Entity Framework Core în metoda ConfigureServices.

```
...
services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(Configuration["Data:SportStoreProducts:ConnectionString"]));
...
```

Metoda de extensie AddDbContext stabilește serviciile furnizate de Entity Framework Core pentru clasa I de context a bazei de date create în Exemplul 13. Multe dintre metodele utilizate în clasa Startup permit configurarea serviciilor și a caracteristicilor de middleware folosind argumente opționale.

Argumentul la metoda AddDbContext este o expresie lambda care primește un obiect de options care configurează baza de date pentru clasa context. În acest caz, am configurat baza de date cu metoda UseSqlServer și am specificat stringul de conexiune, care este obținut din proprietatea Configuration.

Următoarea modificare pe care am făcut-o în clasa Startup a fost să înlocuim depozitul fals cu unul real, astfel:

```
...
services.AddTransient<IProductRepository, EFProductRepository>();
...
```

Componentele din aplicația care utilizează interfața IProductRepository, care este doar controlerul Product în acest moment, vor primi un obiect EFProductRepository atunci când sunt create, ceea ce le va oferi acces la datele din baza de date. Efectul este că datele false vor fi înlocuite perfect de datele reale din baza de date, fără a fi nevoie să schimbam clasa ProductController.

Dezactivarea verificării domeniului de aplicare. Clasa Program este responsabilă pentru pornirea și configurarea ASP.NET Core înainte de a preda controlul la clasa Startup, iar Exemplul 17 arată modificarea necesară în această clasă. Fără această modificare, o excepție va fi aruncată când încercați să creați schema bazei de date.

Exemplul 17. Preparing for Entity Framework Core in the Program.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
```

```

using Microsoft.Extensions.Logging;

namespace SportsStore {
    public class Program {
        public static void Main(string[] args) {
            CreateWebHostBuilder(args).Build().Run();
        }
        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .UseDefaultServiceProvider(options => options.ValidateScopes = false);
    }
}

```

Aceasta este singura modificare la clasa Program necesară aplicației SportsStore.

Crearea migrației bazei de date. Entity Framework Core este capabil să genereze schema bazei de date folosind clasele de model printr-o caracteristică numită migrations. Când pregătim o migrare, EF Core creează o clasă C# care conține comenzile SQL necesare pentru pregătirea bazei de date. Dacă trebuie să modificăm clasele model, atunci putem crea o nouă migrare care conține comenzile SQL necesare pentru a reflecta modificările. În acest fel, nu trebuie să ne facem griji despre scrierea și testarea manuală a comenzilor SQL ne putem concentra doar pe clasele model C# din aplicație.

Comenzile Core Entity Framework Core sunt efectuate din linia de comandă. Deschidem un nou prompt de comandă sau fereastra PowerShell, navigăm în folderul proiectului SportsStore (cel care conține fișierele Startup.cs și appsettings.json) și executăm următoarea comandă pentru a crea clasa de migrare care va pregăti baza de date pentru prima utilizare. :

dotnet ef migrations add Initial

Când această comandă s-a încheiat, vom vedea un folder Migrations în fereastra Visual Studio Solution Explorer. Acesta este locul în care Entity Framework Core își păstrează clasele de migrație. Unul dintre numele fișierelor va fi o valoare timestamp urmată de _Initial.cs, iar aceasta este clasa care va fi utilizată pentru a crea schema inițială pentru baza de date. Dacă examinați conținutul acestui fișier, puteți vedea cum a fost utilizată clasa de model Product pentru a crea schema.

Popularea initiala a bazei de date. Pentru a popula baza de date și a furniza câteva exemple de date, am adăugat un fișier de clasă numit SeedData.cs în folderul Models și am definit clasa prezentată în Exemplul 18.

Exemplul 18. The Contents of the SeedData.cs File in the Models Folder

```

using System.Linq;
using Microsoft.AspNetCore.Builder;

```

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
namespace SportsStore.Models {
    public static class SeedData {
        public static void EnsurePopulated(IApplicationBuilder app) {
            ApplicationDbContext context = app.ApplicationServices
                .GetRequiredService<ApplicationDbContext>();
            context.Database.Migrate();
            if (!context.Products.Any()) {
                context.Products.AddRange(
                    new Product {
                        Name = "Kayak", Description = "A boat for one person",
                        Category = "Watersports", Price = 275 },
                    new Product {
                        Name = "Lifejacket",
                        Description = "Protective and fashionable",
                        Category = "Watersports", Price = 48.95m },
                    new Product {
                        Name = "Soccer Ball",
                        Description = "FIFA-approved size and weight",
                        Category = "Soccer", Price = 19.50m },
                    new Product {
                        Name = "Corner Flags",
                        Description = "Give your playing field a professional touch",
                        Category = "Soccer", Price = 34.95m },
                    new Product {
                        Name = "Stadium",
                        Description = "Flat-packed 35,000-seat stadium",
                        Category = "Soccer", Price = 79500 },
                    new Product {
                        Name = "Thinking Cap",
                        Description = "Improve brain efficiency by 75%",
                        Category = "Chess", Price = 16 },
                    new Product {
                        Name = "Unsteady Chair",
                        Description = "Secretly give your opponent a disadvantage",
                        Category = "Chess", Price = 29.95m },
                    new Product {
                        Name = "Human Chess Board",
                        Description = "A fun game for the family",

                        Category = "Chess", Price = 75 },
                    new Product {
                        Name = "Bling-Bling King",

```

```

        Description = "Gold-plated, diamond-studded King",
        Category = "Chess", Price = 1200
    }
};
context.SaveChanges();
}
}
}
}
}

```

Metoda statică `EnsurePopulated` primește un argument `IApplicationBuilder`, care este interfața folosită în metoda `Configure` a clasei `Startup` pentru a înregistra componente middleware pentru a gestiona cererile HTTP și aici ne asigurăm că baza de date are conținut.

Metoda `EnsurePopulated` obține un obiect `ApplicationDbContext` prin interfața `IApplicationBuilder` și apelează la metoda `Database.Migrate` pentru a se asigura că migrarea a fost aplicată, ceea ce înseamnă că baza de date va fi creată și pregătită astfel încât să poată stoca obiecte `Product`. În continuare, este verificat numărul de obiecte `Product` din baza de date. Dacă nu există obiecte în baza de date, atunci baza de date este populată folosind o colecție de obiecte `Product` folosind metoda `AddRange` și apoi scrisă în baza de date folosind metoda `SaveChanges`.

Ultima modificare este să adăugăm un apel la metoda `EnsurePopulated` din clasa `Startup`, așa cum se arată în Exemplul 19.

Exemplul 19. Seeding the Database in the Startup.cs File in the SportsStore Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
namespace SportsStore {
    public class Startup {
        public Startup(IConfiguration configuration) =>
            Configuration = configuration;
        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>

```



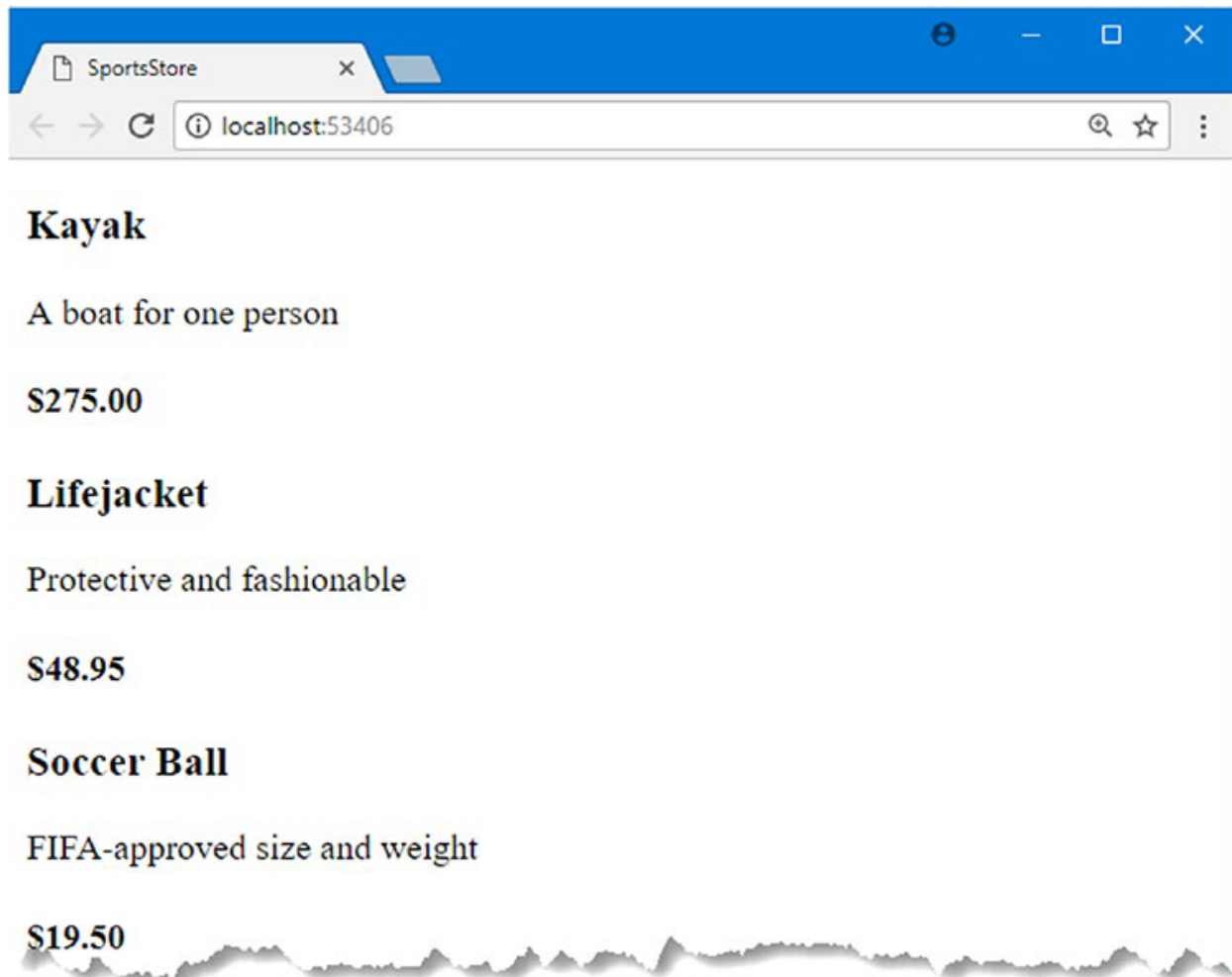
```

        options.UseSqlServer(
            Configuration["Data:SportStoreProducts:ConnectionString"]);
        services.AddTransient<IProductRepository, EFProductRepository>();
        services.AddMvc();
    }
    public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseMvc(routes => {
            routes.MapRoute(
                name: "default",
                template: "{controller=Product}/{action=List}/{id?}");
        });
        SeedData.EnsurePopulated(app);
    }
}
}
}

```

Porniți aplicația și baza de date va fi creată, populată și folosită pentru a furniza datele sale. (Aveți răbdare; poate fi nevoie de un timp suplimentar pentru crearea bazei de date).

Când browserul solicită adresa URL implicită pentru aplicație, configurația aplicației îi comunică MVC că trebuie să creeze un controler Product pentru a gestiona cererea. Crearea unui nou controler Product înseamnă invocarea constructorului ProductController, care necesită un obiect care implementează interfața IProductRepository, iar noua configurație spune MVC că un obiect EFProductRepository trebuie creat și utilizat pentru aceasta. Obiectul EFProductRepository atinge funcționalitatea Entity Framework Core care încarcă date de la SQL Server și le transformă în obiecte Product. Toate acestea sunt ascunse de clasa ProductController, care primește doar un obiect care implementează interfața IProductRepository și funcționează cu datele pe care le furnizează. Rezultatul este că fereastra browserului arată datele de probă din baza de date, așa cum este ilustrat în figura de mai jos.



Adăugarea elementelor de gestionare a catalogului de produse. Convenția pentru gestionarea colecțiilor de articole mai complexe este de a prezenta utilizatorului două tipuri de pagini: o pagină de listă și o pagină de editare, așa cum se arată în figura mai jos. Împreună, aceste pagini permit utilizatorului să creeze, să citească, să actualizeze și să șteargă articole din colecție. Împreună, aceste acțiuni sunt cunoscute sub numele de CRUD. Dezvoltatorii trebuie să implementeze CRUD atât de des, încât instrumentele de dezvoltare includ scenarii pentru crearea de controlere CRUD cu metode de acțiune predefinite.

Index

[Create New](#)

Name	Description	Price	Category	
Kayak	A boat for one person	275.00	Watersports	Edit Details Delete
Lifejacket	Protective and fashionable	48.95	Watersports	Edit Details Delete
Soccer Ball	FIFA-approved size and weight	19.50	Soccer	Edit Details Delete
Corner Flags	Give your playing field a professional touch	34.95	Soccer	Edit Details Delete
Stadium	Flat-packed 35,000-seat stadium	79500.00	Soccer	Edit Details Delete
Thinking Cap	Improve brain efficiency by 75%	16.00	Chess	Edit Details Delete
Unsteady Chair	Secretly give your opponent a disadvantage	29.95	Chess	Edit Details Delete
Human Chess Board	A fun game for the family	75.00	Chess	Edit Details Delete
Bling-Bling King	Gold-plated, diamond-studded King	1200.00	Chess	Edit Details Delete
Alfa	My Product	100.00	Chess	Edit Details Delete

Details

Edit

Product

Product

Name

Description

Price

Category

[Back to List](#)

Name
Kayak

Description
A boat for one person

Price
275.00

Category
Watersports

[Edit](#) | [Back to List](#)

Pentru a implementa un CRUD standard in ASP.NET Core MVC vom face clic dreapta pe folderul Controllers apoi selectam Add/ New Scaffolded item. Selectam MVC Controller with View, using Entity Framework asa cum se poate vedea mai jos, selectam clasa Product ca si model si actionam butonul Add. Acum, daca pornim aplicatia o sa putem sa actualizam si sa vizualizam tabela product.

tion;

ng[

s).t

r C

Iden

()

tExe












tExe

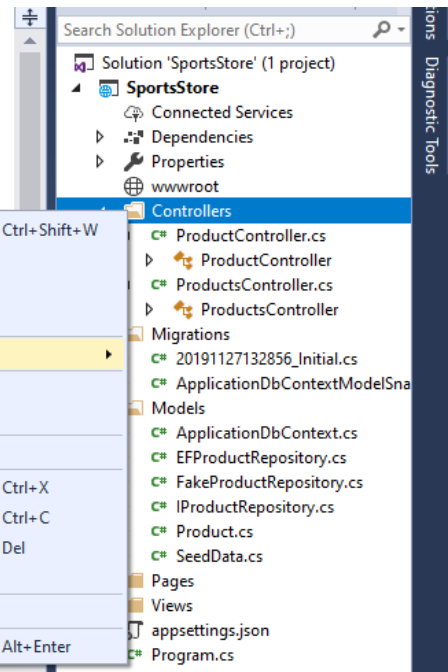
Add Scaffold

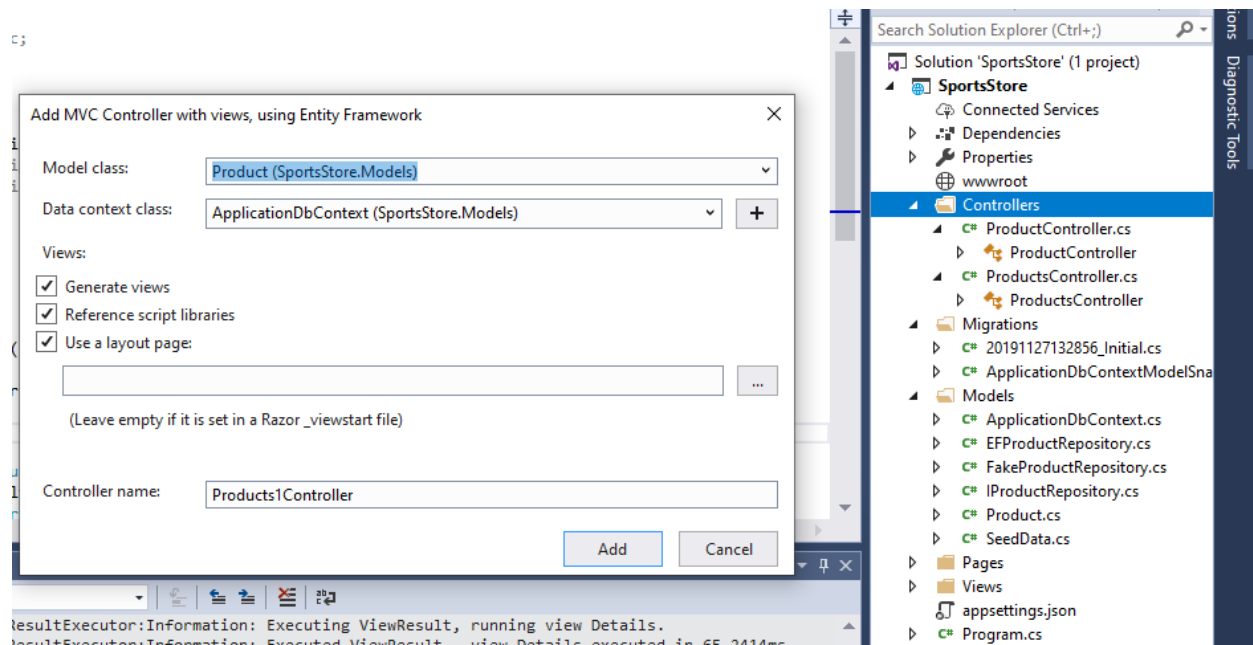
Installed

Common

- API
- MVC
- Razor Pages
- Identity

-  MVC Area
-  MVC Controller - Empty
-  MVC Controller with read/write actions
-  MVC Controller with views, using Entity Framework
-  API Controller - Empty
-  API Controller with read/write actions
-  API Controller with actions, using Entity Framework
-  MVC View
-  Razor Page
-  Razor Page using Entity Framework
-  Razor Pages using Entity Framework (CRUD)





Exercitii.

1. Studiați și testați exemplele din textul de mai sus.
2. Să se creeze o aplicație web ASP.NET Core MVC care să creeze și să actualizeze tabela CATALOG din baza de date TESTDBMVC. Tabela CATALOG are câmpurile Id, marca, nume, prenume, nota1, nota2.