Laboratorul 5

Dezvoltarea unei aplicații ASP.NET cu C#

Ce ne propunem astăzi?

În acest laborator ne propunem să implementăm o aplicație web folosind ASP.NET și Visual C#. Aplicația este destinată unei biblioteci și permite evidența cărților, a cititorilor și a împrumuturilor efectuate. Toate datele vor fi păstrate într-o bază de date SQL Server, interacțiunea cu aceasta fiind efectuată cu ajutorul Entity Framework..

ASP.NET pune la dispoziție trei framework-uri pentru crearea aplicațiilor web: Web Forms, ASP.NET MVC și ASP.NET Web Pages. Toate acestea trei sunt framework-uri stabile și mature, care permit crearea de aplicații web complexe. Indiferent de cel pe care îl vom alege vom beneficia de toate caracteristicile ASP.NET. Mai mult, cele trei framework-uri nu sunt complet independente, iar alegerea unuia nu va exclude utilizarea altuia. Deoarece framework-urile pot coexista în cadrul aceleiași aplicații, nu este ceva neobișnuit să întâlnim diferite componente ale aplicațiilor web dezvoltate folosind framework-uri diferite.

Aplicația din acest laborator (Figura 1) o vom dezvolta folosind ASP.NET MVC, iar pentru lucrul cu datele din baza de date vom folosi Entity Framework.

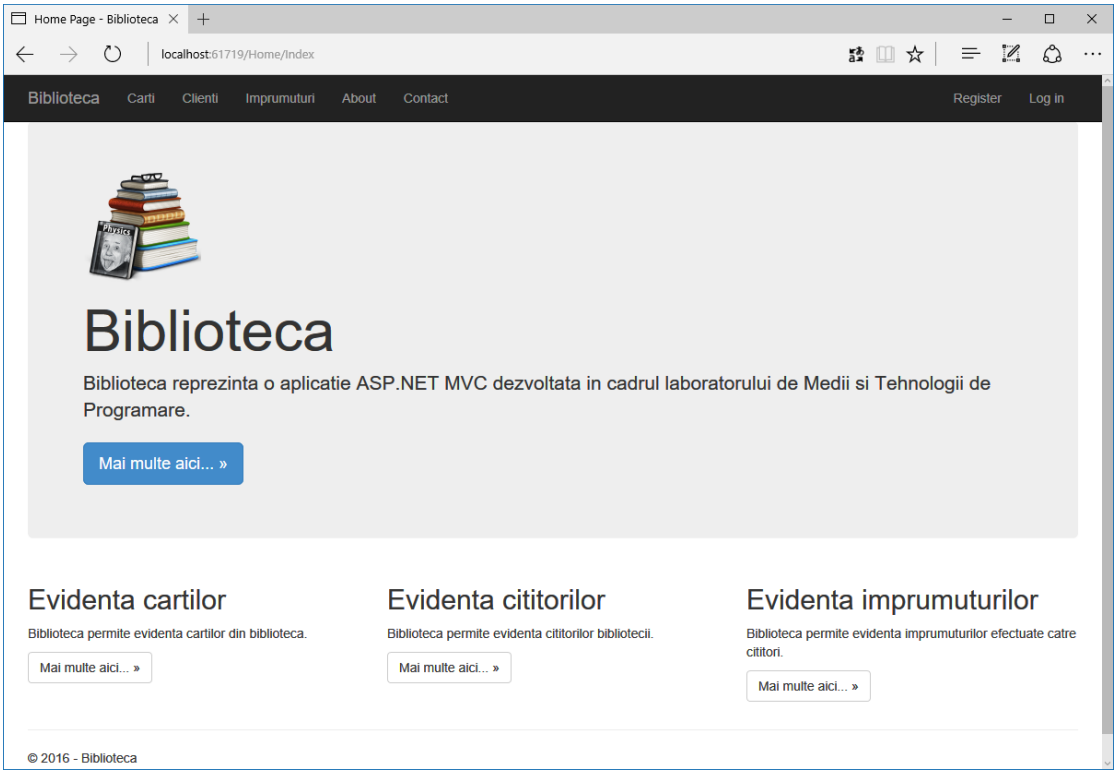

***Figura 1**. Pagina de start a aplicației Biblioteca*

Mai pe larg, vom proceda astfel…

Vom crea un proiect nou de tip ASP.NET Web Application (Figura 2), apoi, din șabloanele predefinite vom alege MVC (Figura 3).

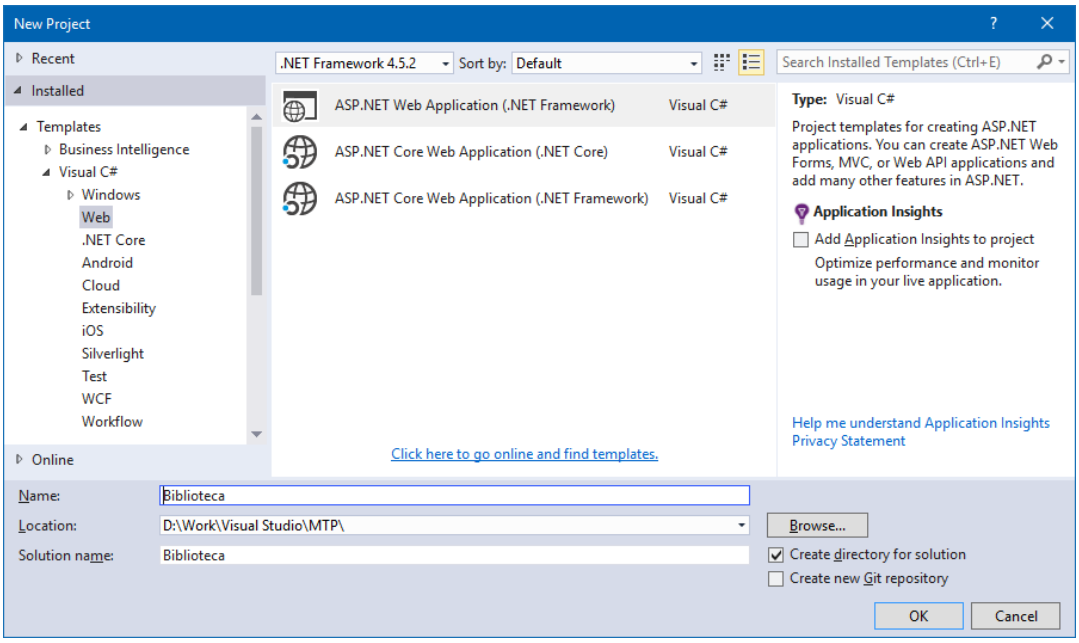

Figura 2. *Crearea unui proiect nou*

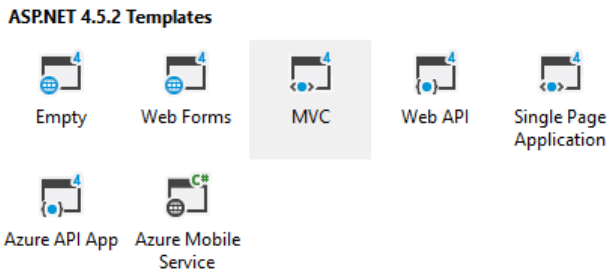

Figura 3. *Alegerea unei aplicații de tip ASP.NET MVC*

Visual Studio va crea automat scheletul unei aplicații web funcționale. Printre alte funcții gata implementate, aplicația va permite înregistrarea și autentificarea utilizatorilor aplicației, datele acestora fiind stocate local, într-o bază de date separată. Puteți rula în acest moment aplicația pentru a examina funcțiile implementate automat. Componentele paginii de start le puteți găsi în Solution Explorer în directorul Views/Home.

Șablonul MVC

Înainte de a trece mai departe haideți să vedem ce este o aplicație MVC. Șablonul arhitectural MVC (Model-View-Controller) separă o aplicație în trei categorii principale de componente: Models, Views, Controllers. Acest șablon ajută la realizarea principiului software denumit „separation of concerns”. Acesta implică, în linii mari, modularizarea, sau separarea unei aplicații software în secțiuni distincte, fiecare fiind responsabilă cu îndeplinirea unei sarcini clare.

În cadrul MVC, cererile utilizatorului sunt prelucrate de către un **Controller**, acesta fiind responsabil de operarea împreună cu **Modelul** (setul de date cu care operează aplicația)

pentru îndeplinirea acțiunilor cerute și/sau extragerea rezultatelor. Controller-ul alege **View-ul** care să îi fie afișat utilizatorului și îi pune la dispoziție datele cerute din **Model**.

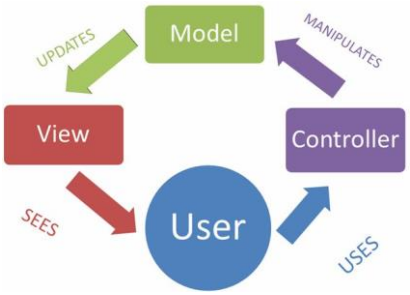

Figura 4. *Legătura dintre componentele MVC*

Legătura la baza de date

În continuare va trebui să ne construim propriile pagini pentru interacțiunea cu datele aplicației. Dacă în laboratorul trecut am construit manual o bază de date pe care am accesat-o din cadrul aplicației, în lucrarea de față vom utiliza o altă modalitate de operare cu datele.

Entity Framework (EF) reprezintă un set de tehnologii din cadrul ADO.NET care ajută la dezvoltarea aplicațiilor software care operează cu baze de date, permițându-le dezvoltatorilor un nivel superior de abstractizare atunci când operează cu aceste date, fără a fi nevoie să se îngrijească de tabelele și câmpurile în care aceste date sunt stocate. Entity Framework oferă o modalitate de mapare a obiectelor relaționale, având avantajul că elimină nevoia scrierii de cod pentru conectare la o bază de date.

Entity Framework suportă două modalități prin care putem interacționa cu o bază de date:

- *Entity Framework Code First*: EF generează automat baza de date pe baza unei structuri clase definite de programator (Models) care mapează structura unor tabele.
- *Entity Framework Database First*: EF generează automat o structură de clase (Models) pornind de la o structura unei baze de date existente.

În ambele cazuri legătura cu datele din baza de date se face în mod programatic, prin clasele care implementează modelul de date.

Pentru lucrarea de față vom alege să modelăm datele folosind Entity Framework Code First. Astfel, vom construi un model al datelor conform cu diagrama din Figura 5.

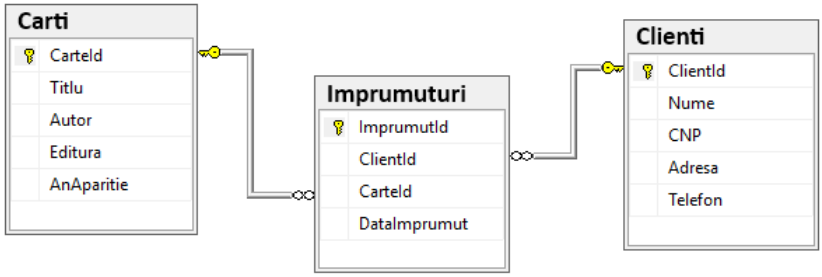

Figura 5. *Structura tabelelor din baza de date*

Înainte de utilizarea Entity Framework este bine să vă asigurați că această componentă este instalată (componentele pot fi instalate individual, pentru fiecare soluție în parte). Astfel, selectați din meniu comanda Tools → NuGet Package Manager → Manage NuGet Packages for Solution și alegeți pentru instalare componenta EntityFramework (vezi Figura 6).

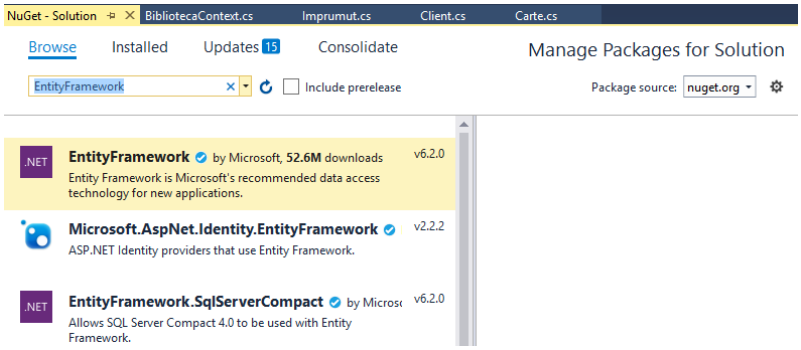

Figura 6. *Instalarea componentei EntityFramework.*

În continuare vom adăuga, rând pe rând, trei clase noi care implementează structura datelor aplicației, conform celor trei tabele. În Solution Explorer dați click dreapta pe directorul *Models* -> Add -> Class (Figura 7).

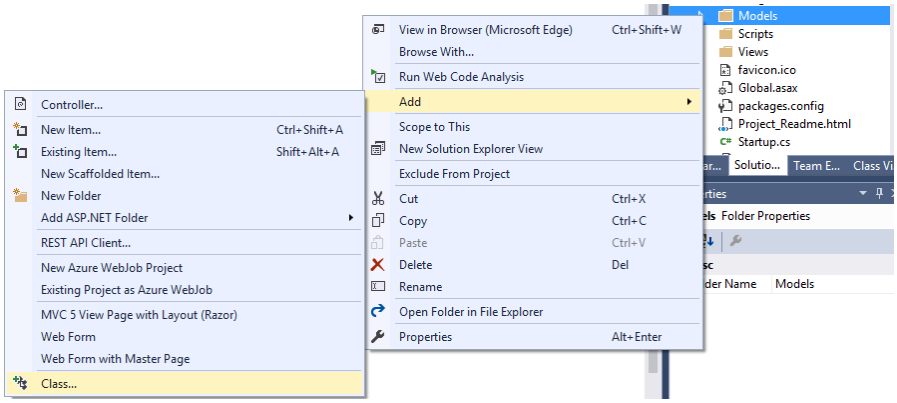

Figura 7. *Adăugarea unei clase model.*

Vom adăuga trei clase: `Carte`, `Client` și `Imprumut`.

```
using System.ComponentModel.DataAnnotations;

namespace Biblioteca.Models
{
    public class Carte
    {
        public int CarteId { get; set; }
        public string Titlu { get; set; }
        public string Autor { get; set; }
        public string Editura { get; set; }
        [Range(1900,2016), Display(Name = "Anul aparitiei")]
        public int AnAparitie { get; set; }
    }
}
```

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Biblioteca.Models
{
    public class Client
    {
        public int ClientId { get; set; }
        public string Nume { get; set; }
        [MaxLength(13), MinLength(13)]
        [RegularExpression("^[0-9]*$", ErrorMessage = "CNP invalid")]
        public string CNP { get; set; }
        public string Adresa { get; set; }
        public string Telefon { get; set; }
```

```
        public virtual ICollection<Imprumut> Imprumuturi { get; set; }
    }
}
```

```
using System;
using System.ComponentModel.DataAnnotations;

namespace Biblioteca.Models
{
    public class Imprumut
    {
        public int ImprumutId { get; set; }
        public int ClientId { get; set; }
        public int CarteId { get; set; }
        [Display(Name = "Data imprumut"), DataType(DataType.Date)]
        public DateTime DataImprumut { get; set; }

        public virtual Client Client { get; set; }
        public virtual Carte Carte { get; set; }
    }
}
```

Puteți observa la nivelul claselor Model o serie de adnotări care vor oferi componentelor View informații privitoare la modul de afișare a datelor și la validările câmpurilor.

În continuare vom defini (tot în directorul *Models*) o clasă derivată din clasa `DbContext` care va expune proprietăți colecție de tip DbSet. Această clasă coordonează funcționalitatea Entity Framework pentru un anumit model de date. În codul acestei clase vom specifica acele entități care vor fi incluse în modelul de date.

```
using System.Data.Entity;

namespace Biblioteca.Models
{
    public class BibliotecaContext:DbContext
    {
        public BibliotecaContext() : base("BibliotecaContext")
        {

        }

        public virtual DbSet<Client> Clienti { get; set; }
        public virtual DbSet<Carte> Carti { get; set; }
        public virtual DbSet<Imprumut> Imprumuturi { get; set; }
    }
}
```

Sfaturi utile

*Dacă lucrați cu Entity Framework Database First, clasa de tip **database context** va fi generată automat, însă în cazul în care alegeți Entity Framework Code First va trebui să definiți manual această clasă.*

Pentru ca aplicația să cunoască locația și parametrii de conectare la baza de date va trebui să adăugăm stringul de conectare în fișierul Web.config în secțiunea *configuration -> connectionStrings*, la finalul fișierului. Stringul de conectare adăugat este marcat cu galben în secvența de mai jos:

```
...
  <connectionStrings>
    <add name="BibliotecaContext" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;
AttachDbFilename=|DataDirectory|\Biblioteca.mdf; Integrated Security=True"
providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Baza de date (de tip SQL Server Local DB) va fi creată la prima accesare a unui View care efectuează operații de citire sau scriere în/din baza de date. Putem verifica acest lucru selectând în Solution Explorer opțiunea *Show All Files*, apoi examinând conținutul directorului App_Data (vezi Figura 8).

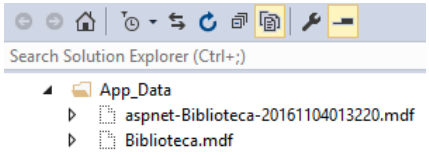

Figura 8. *Vizualizarea bazei de date a aplicației*

Până aici am definit componenta Model a aplicației. Se recomandă să dați comanda *Rebuild Solution* pentru a forța Visual Studio să recunoască componentele Model definite până acum.

Adăugarea componentelor Controller și View

În continuare vom defini componentele Controller și View. În Solution Explorer, dați click dreapta pe directorul Controllers -> Add -> Controller (Figura 9).

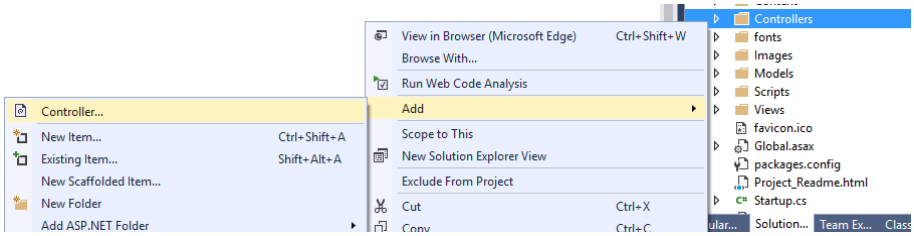

Figura 9. *Adăugarea unei componente Controller*

Din dialogul care se va deschide vom alege opțiunea *MVC 5 Controller with views, using Entity Framework* (Figura 10). În acest fel, odată cu componenta Controller vor fi create o serie de View-uri corespunzătoare operațiilor de tip CRUD (Create, Read, Update, Delete) asupra componentei Model asociate.

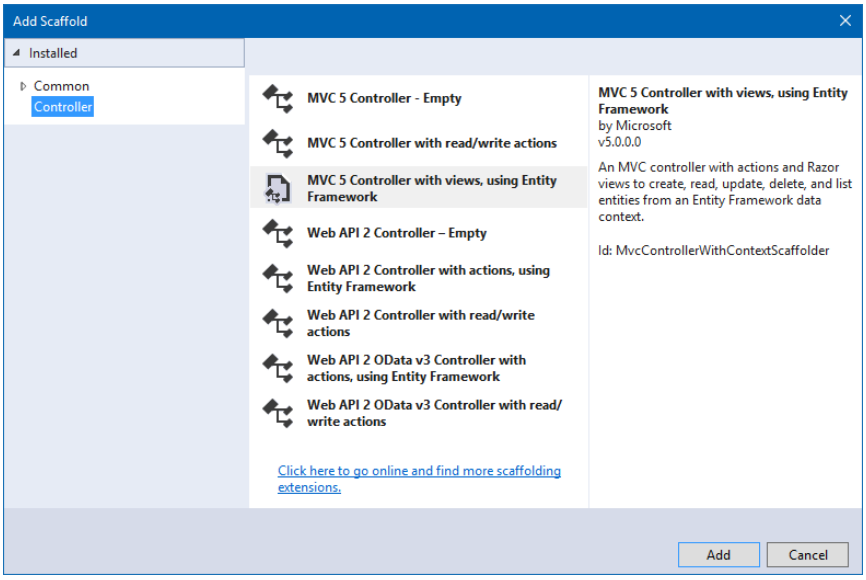

Figura 10. *Alegerea tipului componentei Controller*

În continuare, în fereastra dialog ce se va deschide (Figura 11), vom indica clasa de tip Model, clasa data context și vom avea grijă ca opțiunea *Generate views* să fie selectată. De asemenea vom avea grijă ca numele clasei Controller să fie cel dorit de noi.

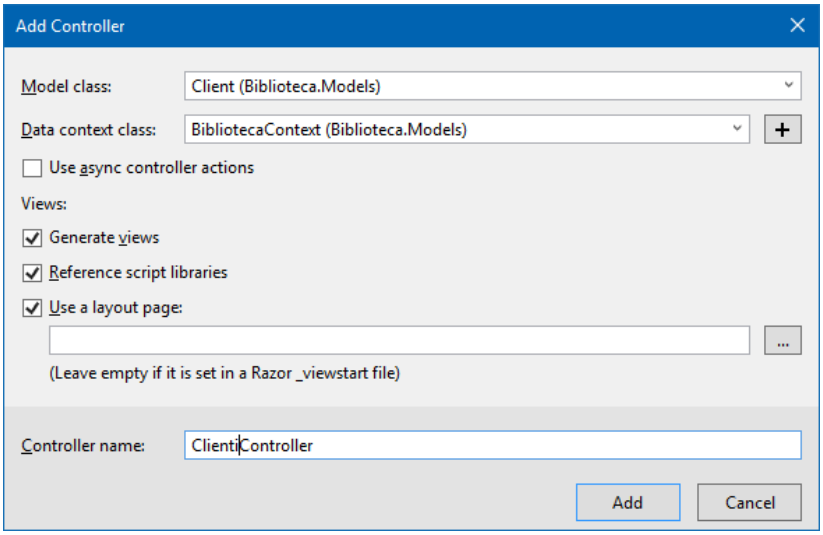

Figura 11. *Definirea opțiunilor și componentelor asociate unui Controller*

Urmând acești pași vom defini trei clase Controller: `ClientiController`, `CartiController` și `ImprumuturiController`.

Până în acest punct am creat componentele Model, View și Controller pentru accesarea datelor aplicației. Pentru fiecare Controller au fost create o serie de cinci View-uri diferite, care corespund la cinci acțiuni specifice (Figura 12):

- **Create**: crearea unei înregistrări noi;
- **Delete**: ștergerea unei înregistrări existente cu afișarea unui dialog de confirmare;
- **Details**: vizualizare detaliată a unei înregistrări;
- **Edit**: modificarea datelor unei înregistrări;
- **Index**: afișarea listei cu toate înregistrările.

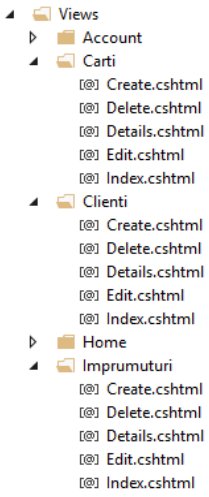

Figura 12. *View-urile create pentru cele trei componente Controller*

Singurul lucru pe care trebuie să îl mai facem, înainte de rularea aplicației, este să adăugăm legături la aceste View-uri din pagina principală a aplicației.

Aspectul View-urilor aplicației este influențat de pagina `/Views/Shared/_Layout.cshtml`. Aceasta definește, printre altele, antetul și subsolul comune tuturor paginilor aplicației. În meniul aplicației vom adăuga legături la paginile pentru evidența cărților, a clienților și a împrumuturilor. Modificările efectuate în pagina `_Layout.cshtml` sunt marcate cu galben:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Biblioteca</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")

</head>
<body>
    <div class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse" data-
target=".navbar-collapse">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("Biblioteca", "Index", "Home", new { area = "" }, new { @class
= "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Carti", "Index", "Carti")</li>
                    <li>@Html.ActionLink("Clienti", "Index", "Clienti")</li>
                    <li>@Html.ActionLink("Imprumuturi", "Index", "Imprumuturi")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
                @Html.Partial("_LoginPartial")
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - Biblioteca</p>
        </footer>
    </div>
```

```
    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

Observați faptul că este vorba de o pagină care conține cod HTML, însă, pe alocuri se face uz de operatorul @. Acesta specifică faptul că, împreună cu codul HTML, pot fi inserate inline secvențe de cod C# (în cazul nostru este vorba de un proiect C#, însă în cazul unui proiect Visual Basic .NET codul inserat va fi VB). Mai multe despre folosirea *sintaxei Razor* puteți citi în articolul [5] din secțiunea Bibliografie a lucrării de față.

Legăturile indicate prin `@Html.ActionLink` indică pagina la care se face referire. De exemplu, `@Html.ActionLink("Carti", "Index", "Carti")` va construi un link cu textul „Carti” care indică spre View-ul `Index` asociat Controller-ului `Carti`.

Sfaturi utile

Înainte de a rula aplicația asigurați-vă că nu aveți deschisă în editorul de cod o pagină de tip CSHTML, deoarece Visual Studio va lansa aplicația și va forța încărcarea acelei pagini deschise în fereastra curentă a editorului. În cazul anumitor pagini care au nevoie de un parametru index (Delete, Details, Edit) încărcarea lor în browser fără indicarea parametrului va duce la afișarea unei pagini de eroare.

Dacă doriți ca aplicația să nu afișeze la start conținutul implicit ASP.NET, creați un conținut personalizat pentru aplicația voastră, modificând View-urile afișate în pagina de pornire (/Views/Home/*).

În acest moment puteți rula aplicația și să navigați printre paginile acesteia adăugând date despre cărți, clienți și împrumuturi. Observați faptul că View-urile conțin toate acțiunile necesare (*Create New*, *Edit*, *Details*, *Delete*, *Back to List* etc.), iar câmpurile sunt afișate și, după caz, formatate automat în funcție de tipul lor sau de indicațiile din adnotările conținute în clasele Model. De asemenea, datele sunt validate înainte de salvarea lor în baza de date.

Adăugarea funcției de căutare a unui cititor

Înainte de implementarea funcției de căutare examinați conținutul clasei `ClientiController` pentru a vedea modul în care sunt implementate acțiunile (cererile din partea utilizatorului). De exemplu, metoda `Index()` nu primește niciun parametru și returnează un View căruia îi transmite ca argument un obiect Model reprezentând lista tuturor clienților (observați utilizarea obiectului `db` de tip `BibliotecaContext`).

Vom modifica metoda `Index()` astfel încât să primească ca parametru numele după care se va efectua căutarea.

```
        public ActionResult Index(string nume)
        {
            var clienti = from c in db.Clienti
                          select c;
            if (!String.IsNullOrEmpty(nume))
            {
                clienti = clienti.Where(c => c.Nume.Contains(nume));
            }
            return View(clienti);

        }
```

Pentru o căutare mai facilă am folosit LINQ (Language Integrated Query – componentă a .NET Framework care adaugă capabilități de interogare nativă a datelor limbajelor .NET) și expresii lambda.

În continuare vom adăuga în View-ul *Index* asociat Controller-ului *Client* codul marcat cu galben din secvența de mai jos.

```
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm())
    {
    <p>
        Nume: @Html.TextBox("nume") <br />
        <input type="submit" value="Cauta" />
    </p>
    }

</p>
```

Va fi adăugată o casetă text (în care utilizatorul va putea introduce numele căutat) și un buton pentru efectuarea operației de căutare. `Html.BeginForm()` va utiliza metoda POST adresată paginii curente, cererea fiind procesată în Controller de metoda asociată View-ului. În cazul nostru, fiind vorba de View-ul *Index*, va fi apelată metoda `Index()` din `ClientiController.cs` căreia îi va fi transmis argumentul `nume`.

Adăugarea de detalii suplimentare la componenta View

În continuare dorim ca View-ul Details a Controller-ului Client să afișeze suplimentar detalii privitoare la data nașterii și vârsta clientului (obținute din CNP-ul acestuia) și un istoric al tuturor împrumuturilor efectuate de acesta (vezi Figura 13).

Figura 13. *Afișarea detaliilor suplimentare pentru un client*

Lista împrumuturilor o putem afla ușor din obiectul Client curent, datorită proprietăților de tip colecție oferite de clasele Model. Pentru afișarea informațiilor despre împrumuturi ne putem inspira din modul în care sunt afișate în View-ul Index toți clienții.

Visual Studio ne oferă un ajutor considerabil prin aceea că Intellisense este disponibil nu doar în fișierele sursă C# ci și în cadrul paginilor ASP.

Calcularea datei de naștere și a vârstei

Pentru calcularea vârstei unui client va trebui întâi să aflați data de naștere a acestuia din CNP (anul, luna și ziua) folosind metoda `Substring()` a unui string. Teoretic, totul ar trebui să fie simplu din momentul în care ați aflat data de naștere, deoarece vârsta o puteți afla scăzând data nașterii din data curentă (`DateTime.Today`, sau `DateTime.Now`). Deși tipul DateTime, folosit pentru lucrul cu date calendaristice, pune la dispoziție metoda `Subtract()` pentru efectuarea diferenței a două date calendaristice, rezultatul va fi stocat, contrar așteptărilor, într-o structură de tip TimeSpan. Aceasta nu oferă o metodă pentru extragerea componentei an din intervalul de timp (deși clasa DateTime oferă!), astfel că va trebui să găsim o altă modalitate de calculare a diferenței dintre două date calendaristice exprimată în ani.

Atât DateTime cât și TimeSpan pun la dispoziție lucrul cu ticks. Un tick reprezintă 10^{-7} secunde (1 secundă = 10.000.000 ticks). Astfel, pe de-o parte, orice dată calendaristică și orice interval de timp pot fi exprimate în ticks, iar pe de altă parte, pornind de la un număr de ticks putem construi un obiect DateTime sau TimeSpan prin intermediul constructorilor acestora. Știind toate aceste lucruri, aflarea vârstei unei persoane nu ar trebui să mai fie o problemă.

Data de naștere și vârsta pentru clientul curent le puteți calcula în metoda `Details` din controllerul `ClientiController.cs`, transmițându-le mai apoi către componenta View prin intermediul ViewBag. ViewBag este o proprietate folosită în cadrul aplicațiilor ASP.NET MVC care vă permite să partajați în mod dinamic diferite valori între un controller și un view. De exemplu, în controller puteți scrie:

```
ViewBag.Varsta = 25;
```

Iar în view puteți face referire la valoarea setată în controller astfel:

```
@ViewBag.Varsta
```

Opțional, cei doritori ar putea să implementeze în continuare funcția de filtrare a cărților în funcție de anul de apariție (exemplu: cărțile care au apărut între anii 2000-2010).

Cu ce ne-am ales?

Prin aplicația dezvoltată în cadrul laboratorului de astăzi am reușit să facem primii pași în realizarea aplicațiilor web ASP.NET MVC și lucrul cu datele dintr-o bază de date folosind Entity Framework Code First.

Bibliografie

[1] Building Websites in ASP.NET, *https://www.asp.net/get-started/websites*

[2] Getting Started with ASP.NET MVC 5, *https://www.asp.net/mvc/overview/getting-started/introduction/getting-started*

[3] MVC, *https://docs.asp.net/en/latest/mvc/index.html*

[4] Getting Started with Entity Framework 6 Code First using MVC 5, *https://www.asp.net/mvc/overview/getting-started/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application*

[5] Introduction to ASP.NET Web Programming Using the Razor Syntax (C#), *https://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-c*