Dezvoltarea aplicațiilor Web

Andrei Sipoș

Facultatea de Matematică și Informatică, DL Info, Anul III
Semestrul I, 2020/2021

Aspecte organizatorice

Pagina de web a cursului

Informații actualizate despre curs, incluzând acest suport, se pot găsi la pagina:

https://cs.unibuc.ro/~asipos/daw.html

Desfășurare

Atât cursul, cât și laboratoarele vor fi ținute pe Microsoft Teams, mai exact de către următorii oameni:

- cursul (seriile 33 și 34): Andrei Sipoș
- laboratoare
  - grupele 331, 333: Ștefan Buzoianu
  - grupele 332, 334: Ștefan Iordache
  - grupele 341, 342, 343, 344: Andreea Guștер

Notare

Conform tradiției cursului, nota va avea două componente:

- 50% examen final (de fapt verificare)
- 50% proiect

Despre examen

Examenul se va desfășura în ultimul laborator și va dura două ore.

Despre proiect

Proiectul va fi realizat individual și va consta într-un site web ce va folosi (cel puțin la bază) tehnologie ASP.NET MVC 5 sau ASP.NET Core MVC și eventual și alte tehnologii web.

Se va aprecia în special complexitatea algoritmică și funcțională.

Generalități

Abstractizare

În istoria limbajelor și paradigmelor de programare, de orice fel ar fi ele, se întrezărește o temă ce reapare constant, anume *abstractizarea* într-o măsură cât mai mare și cât mai potrivită diferitelor aplicații.

Abstractizarea conduce la un cod care se poate întreține mai ușor și despre care se poate judeca mai ușor.

Teorema fundamentală a ingineriei software – sintagmă introdusă de Andrew Koenig pentru a descrie o remarcă a lui Butler Lampson, atribuită lui David J. Wheeler:

"We can solve any problem by introducing an extra level of indirection."

The Ballad of Software Development (2016)

Add a level of indirection
Don't hardcode
Modularize
Don't repeat yourself
Parametrize
Abstract

...And that's the magic.

Pericolele abstractizării

Trebuie să avem în vedere că abstractizarea poate conduce la:

- Inversiunea ei
  - funcționalitatea se duplică în mod involuntar pe diferite niveluri
  - a se vedea exemple la: https://en.wikipedia.org/wiki/Abstraction_inversion
- Leaky abstraction
  - fenomen inevitabil ce apare ca urmare a faptului că abstractizarea nu poate surprinde toate detaliile fenomenului pe care dorim să-l ascundem
  - concept introdus în articolul de blog https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

POO

Un mod de abstractizare relevant acestui curs este *programarea orientată pe obiecte*.

Conceptele de clasă și obiect au apărut pentru prima dată în limbajul Simula, dezvoltat de Ole-Johan Dahl și Kristen Nygaard la Norsk Regnesentral în anii '60 (din acest limbaj s-a inspirat în mod direct Bjarne Stroustrup în anii '80 în crearea C++).

Popularizarea POO s-a datorat mai ales limbajului Smalltalk, introdus în principal de Alan Kay și Adele Goldberg, în anii '70, la laboratoarele Xerox PARC.

Smalltalk este un limbaj OO *par excellence*. Concepte problematice nu sunt prezente în Smalltalk, cum ar fi moștenirea multiplă din C++ (ce nu este, de altfel, prezentă, exceptând forme rudimentare, nici în Java sau C#, ale căror modele de obiecte au fost concepute ca fiind variante simplificate ale celui din C++).

MVC

În cadrul dezvoltării Smalltalk, în 1979, a apărut pentru prima dată ideea de MVC – concept introdus de Trygve Reenskaug și numit astfel în urma discuțiilor sale cu Adele Goldberg.

MVC reprezintă un mod de a organiza codul cu scopul inițial declarat de a dezvolta aplicații GUI pe desktop.

Codul programului este împărțit în genere în trei componente, denumite convențional **M**odel, **V**iew, **C**ontroller.

Descrierea MVC

Modelul reprezintă datele specifice domeniului aplicației, i.e. *logica business*.

"A software solves a business problem."
(Tu, Do Huang, *Operating Systems: From 0 to 1*, capitolul 1)

Viewul reprezintă modul în care informația este afișată.

Controllerul, în funcție de inputul dat de utilizator, execută transformările necesare asupra modelului.

MVC pe Web

În general, bibliotecile concepute pentru a descrie site-uri web, așa-numitele *web frameworks*, au de efectuat o abstractizare foarte dură, deoarece au de interpretat un protocol eminamente *stateless* cum este HTTP (v. cursul de Rețele de calculatoare).

MVC și-a făcut debutul pe scena web sub o formă netradițională, anume arhitectura **JSP Model 2** (1998). Diferența majoră este că view-ul nu mai afișează în mod actualizat conținutul modelului, acest fenomen fiind considerat nenatural pe web.

Creșterea masivă în popularitate a Model 2 a fost dată în principal de framework-ul **Ruby on Rails** (2004).

Express.js (2010) este un web framework MVC pentru Node.js (studiat la sfârșitul cursului de Tehnici web).

Menționez și un web framework actual pentru Smalltalk – anume **Seaside** – acesta fiind bazat însă pe *continuations* și nu pe MVC.

Microsoft

La acest curs vom studia tehnologie Microsoft.

Mai precis, limbajul folosit va fi C#, introdus de Anders Hejlsberg în 2000 ca parte din inițiativa .NET a Microsoft.

- John C. Dvorak, *Microsoft Dot Nyet*, PCMag Dec. 2000.

În forma sa cea mai răspândită (e.g. Visual Studio), limbajul (ca și Java) se situează între interpretare și compilare, fiind mai întâi tradus în Common Intermediate Language (bytecode) și apoi compilat just-in-time în cod-mașină.

Un limbaj foarte iubit de programatori:

cartman82 Aug 23, 2014, 7:59 PM

The "Which language is the least bad" thread has predictably turned into a C# love fest.

Vom învăța C# **din mers** (prin referire la C++ și Java).

Bibliografie C#

Referințe detaliate despre C#:

- Mark Michaelis with Eric Lippert and Kevin Bost, *Essential C# 8.0*, Seventh Edition, Addison-Wesley Professional, 2020.
- Jon Skeet, *C# in Depth*, Fourth Edition, Manning Publications, 2019.

Microsoft pe Web

Enumerăm frameworkurile web Microsoft:

- **A**ctive **S**erver **P**ages (1996) – unde codul putea fi scris în VBScript, JScript, PerlScript
- ASP.NET Web Forms (2002) – unde se poate folosi orice limbaj .NET (deci inclusiv C#)

ASP.NET Web Forms încearcă pe cât posibil să simuleze experiența *stateful* familiară din Windows Forms: un *leaky abstraction* sever, ce conduce la o cantitate mare de date ce trebuie interschimbată permanent.

Pentru a ameliora asemenea probleme și pentru a oferi un răspuns industriei la Ruby on Rails, Scott Guthrie a introdus o variantă MVC a ASP.NET – versiunea preliminară a fost lansată în 2007, iar cea 1.0 în 2009. Codul a fost lansat sub licență Apache License 2.0 în 2012.

ASP.NET MVC

Vom studia la acest curs ASP.NET MVC 5.

ASP.NET MVC, ca și celelalte web framework-uri MVC pomenite (Express.js, Ruby on Rails) induc o arhitectură thin-client, în care tot codul MVC rulează pe server. A se compara cu tendința actuală de a folosi un back-end minimal (ce doar expune un API și comunică cu baza de date, de exemplu ASP.NET Core), în vreme ce codul MVC este rulat de client (cu un framework ca AngularJS).

Vom fi așadar "în contra direcției de astăzi".

Vom lucra cu Visual Studio Community Edition, care trebuie instalat (din câte văd, puteți să vă înregistrați cu contul FMI) și care include un server web.

Bibliografie ASP.NET MVC 5

Mai multe informații despre subiectul cursului pot fi găsite în următoarele surse:

- Jon Galloway, *ASP.NET MVC Music Store Tutorial*, Version 3.0b, Free Online, 2011.
- Nimit Joshi, *Programming ASP.NET MVC 5: A Problem Solution Approach*, 2013.
- Adam Freeman, *Pro ASP.NET MVC 5*, Fifth Edition, Apress, 2013.
- Jon Galloway, Brad Wilson, K. Scott Allen, David Matson, *Professional ASP.NET MVC 5*, John Wiley & Sons, 2014.

- Mosh Hamedani, *The Complete ASP.NET MVC 5 Course* (curs video)

Debut rapid în ASP.NET MVC 5

MVC în ASP.NET

Arhitectura MVC în ASP.NET mai conține și o a patra componentă, **routerul**, care transformă URL-ul în HTTP Request și apelează metoda (funcția) corespunzătoare din controller.

Modelul (care, după cum am zis, exprimă logica business) este format din clase ce depind de scopul aplicației și ale căror instanțieri sunt Plain Old CLR Objects (POCOs).

Viewul reprezintă codul HTML al unei pagini împreună cu eventuale elemente dinamice ce sunt preluate din model.

Controllerul procesează un HTTP Request prin metodele sale; fiecare asemenea metodă operează de obicei cu obiecte din clasele-model și apoi apelează un view corespunzător cu argumentul instanțiat cu un asemenea obiect-model.

Un exemplu minimal I: Crearea proiectului

Vom vedea acum modul cum se poate obține un site ce ilustrează minimal legăturile dintre componente. Voi demonstra procesul în Visual Studio; totuși, voi include aici pașii comentați pentru a putea fi eventual reprodus.

- Creați un nou proiect cu template-ul ASP.NET Web Application (.NET Framework) – varianta C#.
- Denumiți proiectul DebutRapid.
- În următorul meniu, selectați MVC.
  Vom vedea mai târziu cum putem obține cod MVC și fără a selecta acest șablon.
- Puteți explora proiectul nou-creat în Solution Explorer; de pildă, în `App_Start\RouteConfig.cs` puteți vedea configurația acelui router despre care am vorbit, ce interpretează URL-urile (dar nu vom modifica acum fișierul).

Un exemplu minimal II: Modelul

- Dați drept-click pe Models și selectați Add, Class. Numiți fișierul `Figura.cs` (clasa se va denumi automat `Figura`).
- Observați namespace-ul `DebutRapid.Models` în care se află clasa.
- Înăuntrul clasei, introduceți codul:

```
private string nume;
public string Nume
{
    get { return nume; }
    set { nume = value; }
}
```

Primul rând introduce o variabilă privată (ca în C++). Restul codului introduce o **proprietate** *ce îmbracă variabila într-o formă public accesibilă. Acest tip de cod este destul de răspândit – el admite și următoarea prescurtare:*

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

Un exemplu minimal III: Controllerul

- Dați drept-click pe Controllers și selectați Add, Controller, Empty. Denumiți controllerul `FiguriController`.
- Adăugați directiva

  ```
  using DebutRapid.Models;
  ```

  sub restul directivelor `using`, ca să putem accesa direct, de pildă, clasa `Figura`.
- Copiați metoda `Index` (ce va fi la final accesibilă via URL-ul `/Figuri/`) și redenumiți copia în `Prima` (ce va fi accesibilă via URL-ul `/Figuri/Prima`).
- Modificați codul metodei `Prima` în următorul:

  ```
  Figura f = new Figura();
  f.Nume = "cerc";
  return View(f);
  ```

  După cum se vede, metoda apelează acum view-ul corespunzător controllerului cu argumentul `f`*. Acest view trebuie însă creat.*

Un exemplu minimal IV: Viewul

- Dați drept-click pe numele metodei `Prima` și selectați Add View. Apăsați Add. Observați că viewul a fost creat în `Views\Figuri`.
- Adăugați la începutul fișierului directiva

  ```
  @model DebutRapid.Models.Figura
  ```

  Astfel, compilatorul va ști ce fel de obiect este transmis viewului ca argument.
- Înlocuiți conținutul headerului de tip `h2` cu

  ```
  @Model.Nume
  ```

  Acest cod Razor (cod C# în HTML) va afișa valoarea proprietății `Nume` *a obiectului transmis.*
- Având viewul la vedere, apăsați Ctrl+F5 pentru a compila și afișa pagina ce se obține apelând metoda corespunzătoare din controller.

Introducere în Controller, View și rutare

Despre controllere

Controllerul este singura componentă ce poate funcționa fără celelalte două din MVC. De aceea, expunerile despre framework-ul ASP.NET MVC încep de obicei cu el.

Metodele sale se numesc *acțiuni* și returnează un obiect de tip `ActionResult` (clasă abstractă din care derivă mai multe clase concrete, precum `ViewResult`, returnată de metoda helper `View` folosită în exemplul minimal).

Rutarea implicită

În mod implicit, așa cum ați văzut în fișierul `RouteConfig.cs` din `App_Start`, URL-urile sunt interpretate de către router în modul următor:

`{controller}/{action}/{id}`

unde controllerul implicit este `Home`, acțiunea implicită este `Index`, iar parametrul `id` este opțional.

Clase derivate din ActionResult

Prezentăm cele mai frecvente clase și pentru fiecare clasă, comportamentele uzuale ale metodei helper asociate (lista completă a supraîncărcărilor se poate consulta prin IntelliSense).

- `ViewResult` $\rightsquigarrow$ `View`
  - fără parametri: afișează view-ul corespunzător acțiunii
  - cu parametru string: afișează view-ul specificat de parametru
    - cale relativă: view din folderul aceluiași controller
      ex. `View("Prima")`
    - cale absolută
      ex. `View("~/Views/Home/Contact.cshtml")`
  - cu parametru obiect oarecare: model transmis view-ului (v. exemplul minimal)
  - simultan cu cei doi parametri de mai sus
    ex. `View("Prima", f)`
- `HttpNotFoundResult` $\rightsquigarrow$ `HttpNotFound` (fără parametri)
- `ContentResult` $\rightsquigarrow$ `Content`
  - cu parametru string: creează o pagină având acel string pe post de cod HTML
    ex. `Content("<b>Good night</b>")`

Clase derivate din ActionResult

- `RedirectResult` $\rightsquigarrow$ `Redirect`
  - cu parametru string: redirecționare la URL-ul indicat de string ex. `Redirect("http://fmi.unibuc.ro")`
- `RedirectToRouteResult` $\rightsquigarrow$ `RedirectToAction`
  - cu următorii parametri: un string ce desemnează acțiunea, un string opțional ce desemnează controllerul și un obiect anonim (tot opțional) ce conține parametrii ce vor fi transmiși acțiunii respective

Exemplu pentru ultima clasă:

```
RedirectToAction("Index", "Figuri",
  new { coded = 48, start = "yes" })
```

Acest obiect va produce o redirecționare către URL-ul

`/Figuri?coded=48&start=yes`

Transmiterea parametrilor către acțiuni

Dacă adăugăm parametri unei acțiuni, atunci ei vor fi instanțiați cu parametrii transmiși via URL în felul ilustrat mai devreme (*query string*). Continuând exemplul anterior, acțiunea `Index` din controllerul `Figuri` va putea avea antetul

```
public ActionResult Index(int coded, string start)
```

Putem face anumiți parametri opționali, folosindu-ne de tipurile *nullable* din C#:

```
public ActionResult Index(int? coded)
{
    if (!coded.HasValue) coded = 7;
    return Content(coded.ToString());
}
```

Transmiterea parametrilor în alte moduri

Se observă că în rutarea implicită parametrul `id` poate fi transmis via URL. Așadar, dacă adăugăm unei acțiuni un parametru numit id, iar apoi introducem valoarea dorită în URL, spre exemplu:

`Figuri/Index/5`

codul se va comporta exact ca în cazul când parametrul este transmis prin *query string*.

Pentru a transmite și alți parametri, nu numai `id`, prin URL, avem nevoie de rutări personalizate.

Există și un al treilea mod de a transmite parametri, anume prin forme, vom vedea mai târziu.

Rutări convenționale (tradiționale)

Pentru ca site-ul să admită noi rute, se adaugă în metoda `RegisterRoutes` din `RouteConfig.cs` o nouă apelare a metodei `MapRoute` a obiectului routes. În supraîncărcarea sa cea mai comună, metoda are următorii parametri:

- un string ce denotă numele rutei
- un string ce denotă șablonul de URL, unde parametrii ce se doresc a fi instanțiați apar cu numele lor între acolade
- un obiect anonim ce conține valori implicite pentru parametri

Spre exemplu:

```
routes.MapRoute(
        "Route1",
        "RouteCustom/{coded}/{start}",
        new { controller = "Figuri", action = "Edit" });
```

Avertisment

Ordinea rutelor este importantă: dacă un URL se potrivește mai multor rute, va fi selectată ruta mapată mai devreme. De pildă, dacă în exemplul anterior introducem apelarea lui `MapRoute` după cea deja existentă în fișier (`Default`), routerul va selecta pentru URL-ul

`RouteCustom/23/yes`

ruta `Default` (pentru că URL-ul se potrivește șablonului respectiv), care va produce o eroare, dat fiind că nu avem un controller denumit `RouteCustom`.

Constrângeri în rute convenționale

Pentru a restricționa mai mult rutele posibile, putem introduce un al patrulea parametru, tot un obiect anonim, ce va conține expresii regulate C# pentru parametri. De exemplu:

```
new { coded = "48|49" }
```

Aici, orice apelare a unui URL ce se potrivește șablonului din `Route1` unde `coded` ia alte valori decât 48 sau 49 va produce selectarea rutei `Default`.

Sintaxa și semantica expresiilor regulate sunt cele ale clasei .NET denumite `Regex`, iar o documentație se poate găsi aici.

Avertisment II

Expresiile regulate .NET (ca și cele din alte limbaje, de exemplu PHP sau Java) nu coincid cu expresiile regulate așa cum sunt ele înțelese în teoria limbajelor formale. De exemplu, limbajul

$$\{a^n b^n \mid n > 0\},$$

despre care s-a demonstrat că nu este regulat, poate fi descris folosind regex-uri .NET (exercițiu de laborator!).

Rutări atribuite

Rutările convenționale au dezavantajul că sunt greu de întreținut – se pot multiplica incontrolabil, și, de asemenea, informația este împărțită în fișiere diferite și trebuie actualizată simultan: codul este fragmentat, deci fragil.

Rutările atribuite rezolvă problema: ele sunt scrise sub forma atributului `Route` pus pe rândul dinaintea antetului acțiunii corespunzătoare, având ca parametru șablonul de URL, spre exemplu:

```
[Route("RouteCustom/{coded}/{start}")]
```

Pentru a mapa aceste rute atribuite, mai trebuie adăugată în `RegisterRoutes`, înainte de mapările obișnuite, linia

```
routes.MapMvcAttributeRoutes();
```

Parametrii pot fi aici făcuți opționali adăugând semnul `?` după numele lor.

Constrângeri în rute atributate

În acest tip de rute, constrângerile sunt indicate în cadrul șablonului; de pildă, cea anterioară se poate scrie sub forma:

```
[Route("RouteCustom/{coded:regex(^(48|49)$)}/{start}")]
```

Atenție: În rutele atributate, potrivirea expresiilor regulate este parțială – expresia 48|49 de mai devreme ar cuprinde și valori ca 1487 – de aceea, pentru a menține comportamentul, trebuie să includem caracterele ^ și $, corespunzătoare începutului și, respectiv, sfârșitului de șir de caractere.

Rutele atributate admit și alte feluri de constrângeri, pe lângă expresiile regulate – o listă completă se poate găsi aici.

ViewData și ViewBag

Se pot transmite obiecte din controller spre view prin dicționarul de date dinamic `ViewData`, de exemplu:

```
ViewData["fig"] = f;
```

Pentru a-l accesa în view, folosim o sintaxă ce trebuie să conțină un cast:

```
@(((DebutRapid.Models.Figura)ViewData["fig"]).Nume)
```

Obiectul `ViewBag` este un *wrapper* pentru `ViewData`, în sensul că `ViewBag.fig` denotă același obiect ca `ViewData["fig"]` (dar nu îl putem folosi decât pentru chei ce sunt denumiri valide de proprietăți).

Din nou, folosirea acestor facilități duce la un cod greu de întreținut, așa că se recomandă doar folosirea parametrului model al metodei `View` pentru a transmite informații către view.

View Models

Dacă dorim să transmitem viewului informații ce nu corespund neapărat unei clase model, vom crea o nouă clasă special cu acest scop – un așa-numit *view model*. Spre exemplu, următoarea clasă conține o figură și o listă de stringuri:

```
public class FigViewModel
{
    public Figura Fig { get; set; }
    public List<String> Li { get; set; }
}
```

View Model în Controller

Vom inițializa în controller acest nou tip de obiect în același mod cum l-am inițializat pe cel precedent:

```
public ActionResult Prima()
{
    Figura f = new Figura();
    f.Nume = "cerc";
    List<String> li = new List<String> { "a1", "p2" };
    FigViewModel vm = new FigViewModel();
    vm.Fig = f;
    vm.Li = li;
    return View(vm);
}
```

View Model în View

În view, după ce ne asigurăm că directiva de model este:

```
@model DebutRapid.Models.FigViewModel
```

ne putem folosi de următorul cod Razor (despre care am spus că este cod C# integrat în HTML) pentru a-i accesa componentele:

```
<h2>@Model.Fig.Nume</h2>
@if (Model.Li.Count == 0)
{<p>Nimic.</p> }
else
{
    <ul>
        @foreach (String s in Model.Li)
        {
            <li>@s</li>
        }
    </ul>
}
```

View-uri parțiale

Un alt instrument MVC este reprezentat de view-urile parțiale. Pentru a crea un asemenea view, dăm drept-click pe un folder oarecare (de ex. tot cel al controllerului `Figuri`), selectăm Add, View, iar în dialogul ce apare bifăm Create as a partial view. Prin convenție, numele view-urilor parțiale începe cu underscore. În exemplul următor, view-ul se numește `_Part`:

```
@model DebutRapid.Models.Figura

<b>@Model.Nume</b>
```

Folosirea view-urilor parțiale

Pentru a accesa view-ul parțial din view-ul obișnuit, ne folosim de funcția `Html.Partial`. De exemplu, în codul următor, vom pasa către view-ul parțial doar componenta de tip `Figura` (pe care o acceptă ca model) a view model-ului:

```
@Html.Partial("_Part", Model.Fig)
```

Accesul la baza de date

Entity Framework, Code First

Entity Framework

Entity Framework este un *object-relational mapping tool*: el ne permite să lucrăm cu tipuri de date structurate ca obiecte atunci când accesăm o bază de date, în ciuda faptului că o bază de date nu lucrează decât cu tipuri simple de date.

Vom folosi abordarea **Code First**, în sensul că nu vom proiecta direct baza de date, ci ea va fi automat generată de către EF în funcție de codul programului.

Pentru a instala Entity Framework într-o soluție Visual Studio, în mediul Tools, selectați NuGet Package Manager, Manage NuGet Packages for Solution, căutați EntityFramework în Browse, bifați proiectul curent și apăsați Install.

Crearea unei baze de date

Pentru a crea o bază de date nouă, dați drept-click pe folderul `App_Data`, selectați Add, New Item iar apoi SQL Server Database. Se va crea un fișier cu extensia `mdf`.

Pentru a accesa baza de date nou-creată, în fișierul `Web.config`, înainte de `</configuration>` se adaugă următorul fragment de text ce definește un *connection string*:

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

unde `db.mdf` este numele fișierului bazei de date.

Clase

În continuare, vom descrie lucrul cu baza de date. După crearea ei, vom introduce clase-model ce reprezintă tipurile de date corespunzătoare tabelelor. În cele ce urmează vom lucra cu următorul exemplu:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
}
```

DbContext și DbSet

Pentru a descrie structura bazei de date, vom crea o nouă clasă, numită clasă context, ce va moșteni `DbContext` (clasă aflată în `System.Data.Entity`, așadar vom adăuga o directivă corespunzătoare `using`).

```
public class DbCtx : DbContext
{
    public DbCtx() : base("BookCS"){}
    public DbSet<Book> Books { get; set; }
}
```

Constructorul clasei context este mai sus gol, dar apelează constructorul clasei de bază cu un parametru, anume un string ce indică numele *connection string*-ului. Înăuntrul clasei context avem proprietăți de tip `DbSet<T>` ce vor reprezenta tabele, iar fiecare `T` va reprezenta tipul de obiect stocat în tabelul respectiv.

Atenție: Nu putem avea mai mult de un tabel de un anumit tip de date.

Strategii de inițializare

Putem adăuga în acel constructor instrucțiuni precum:

```
Database.SetInitializer<DbCtx>(new
        CreateDatabaseIfNotExists<DbCtx>());
```

Aceasta de mai sus va determina baza de date să fie creată la rularea aplicației în caz că ea nu există. Acesta este, însă, comportamentul implicit, astfel încât linia de mai sus este superfluă. Pentru a opri acest comportament, se introduce în schimb linia:

```
Database.SetInitializer<DbCtx>(null);
```

Există, însă, și alte strategii de inițializare.

Strategii existente

- `DropCreateDatabaseIfModelChanges`
  În momentul actual, dacă o anume clasă-model este modificată, programul ne va da eroare. Folosirea strategiei de mai sus, însă, va recrea în acel caz baza de date de la zero.
- `DropCreateDatabaseAlways`
  Această strategie va recrea baza de date la fiecare rulare a aplicației.

Putem avea și strategii personalizate – de pildă, putem extinde clasa ultimei strategii de mai sus pentru a și popula baza de date după recrearea ei.

Strategii personalizate

Creăm, așadar, o nouă clasă ce derivă din `DropCreateDatabaseAlways` (și aici trebuie, deci, să includem `System.Data.Entity` în `using`):

```
public class Initp : DropCreateDatabaseAlways<DbCtx>
{
    protected override void Seed(DbCtx ctx)
    {
        ctx.Books.Add(new Book {
                Title = "The Atomic Times", Author = "Michael Harris" });
        ctx.Books.Add(new Book {
                Title = "In Defense of Elitism", Author = "Joel Stein" });
        ctx.SaveChanges();
        base.Seed(ctx);
    }
}
```

Acest inițializator se va seta prin linia:

```
Database.SetInitializer<DbCtx>(new Initp());
```

Accesul efectiv

Pentru a accesa în mod concret baza de date dintr-un controller, vom introduce în blocul controllerului un obiect privat din clasa context:

```
private DbCtx ctx = new DbCtx();
```

iar în acțiune ne vom folosi de obiect într-un mod precum următorul:

```
List<Book> books = ctx.Books.ToList();
```

Exemplul de mai sus transformă tabelul într-o listă, iar cu liste știm să lucrăm.

Plan

Vom arăta în continuare cum putem diversifica componentele deja prezentate, mai precis vom înțelege:

- modul cum clasele determină structura bazei de date
- modul cum putem interoga prin cod baza de date

Convenții Entity Framework

Vom explora acum convențiile prin care Entity Framework structurează baza de date, precum și modurile în care se pot ele suprascrie.

După cum am spus, pentru fiecare clasă poate exista cel mult un tabel, iar el va lua implicit numele clasei urmat de 's', exceptând cazul când avem pe rândul dinaintea antetului clasei un atribut precum:

```
[Table("NumeTabel")]
```

care va determina numele tabelului.

Proprietăți scalare

Pentru fiecare dintre proprietățile de tipuri de date simple precum int (numite **proprietăți scalare**), se va crea o coloană denumită după numele ei. Comportamentul poate fi suprascris, ca mai înainte, printr-un atribut ca:

```
[Column("Name")]
```

În caz că numele unei proprietăți este `id` sau este format din numele clasei urmat de `id`, ea va fi cheie primară. Pentru ca o proprietate ce nu se supune convenției să fie cheie primară, se folosește atributul `[Key]`.

Putem folosi atributul `[NotMapped]` dacă vrem să nu avem o coloană corespunzătoare.

Există și atribute precum `Required`, `MinLength`, `MaxLength`, al căror comportament este evident.

Proprietăți navigaționale

Presupunem că există două clase pentru care avem tabele, și le numim **entitate principală** și **entitate dependentă**.

În cazul în care fie în entitatea dependentă există o proprietate al cărei tip este entitatea principală (denumită **proprietate navigațională de referință**), fie în entitatea principală există o proprietate de tip `ICollection<T>`, unde `T` este entitatea dependentă (denumită **proprietate navigațională de colecție**), atunci (în al doilea caz, presupunând în plus că în entitatea dependentă nu există o proprietate de colecție către entitatea principală) va exista în entitatea dependentă o coloană care va fi cheie externă pentru entitatea principală. Distingem două cazuri:

- dacă există în entitatea dependentă o proprietate cu numele cheii primare din entitatea principală, ea va fi cheia externă;
- dacă nu, se creează o coloană nouă care va fi cheie externă, iar numele ei va fi numele entității principale urmat de un underscore și de numele cheii primare.

Atributul ForeignKey

În cazul în care dorim ca o proprietate ce nu se supune convenției să fie cheie externă, folosim atributul `ForeignKey`. El poate fi scris în trei locuri:

- deasupra respectivei proprietăți, pentru a desemna proprietatea de referință
- deasupra proprietății de referință, pentru a desemna respectiva proprietate
- deasupra proprietății de colecție, pentru a desemna respectiva proprietate

Implementarea tipurilor de relație între entități

Având aceste cunoștințe, tipurile de relație dintr-o bază de date se pot implementa după cum urmează:

- one-to-one: cu câte o proprietate de referință în fiecare entitate
  În acest caz, pentru ca mediul să distingă între entități (pentru a ști, de pildă, în care dintre ele putem adăuga elemente fără a avea referiri la cealaltă care eventual este vidă), deasupra proprietății de referință din entitatea dependentă se adaugă atributul `[Required]`.
- one-to-many: cu o proprietate de colecție în entitatea 'one' și/sau una de referință în entitatea 'many'
- many-to-many: cu câte o proprietate de colecție în fiecare entitate

Se observă că în acest ultim caz (care a fost exclus mai devreme), Entity Framework va crea un tabel asociativ unde se vor regăsi cheile externe.

Selectarea informațiilor din baza de date

După cum s-a văzut în cursul de Baze de date, aproape întreaga complexitate a interogării unei baze de date se regăsește în clauzele de tip `SELECT`.

Clasa `DbSet`, ale cărei obiecte reprezintă tabelele, implementează interfața `IQueryable`, și ca urmare putem folosi LINQ, o componentă .NET Framework proiectată pentru a interoga surse de date din cele mai diverse.

Sintaxa LINQ

LINQ poate fi exprimat sintactic ca o succesiune de metode:

```
var query = ctx.Books.Where(s => s.Title.Contains("Atomic"));
List<Book> books = query.ToList();
```

sau ca query asemănător SQL-ului:

```
var query = from s in ctx.Books
            where s.Title.Contains("Atomic")
            select s;
List<Book> books = query.ToList();
```

În loc de (sau pe lângă) “where”, putem scrie cca. 50 de operații posibile, denumite *standard query operations*.

Metode de extensie LINQ

Pe lângă metoda `ToList` folosită mai sus și ce returnează o listă a tuturor rezultatelor interogării, mai avem la dispoziție și metode ce returnează un singur element, anume:

- `Find` $\rightsquigarrow$ primește ca argument un număr și returnează obiectul corespunzător liniei de tabel pe care se găsește acel număr drept cheie primară
- `First` $\rightsquigarrow$ returnează primul rezultat (util când s-a aplicat o operație de sortare)
- `Single` $\rightsquigarrow$ returnează unicul rezultat care există (în caz că există mai mult de un rezultat, spre deosebire de `First`, va arunca o excepție)

Forme și CRUD

și HTML Helpers

Metodele GET și POST din HTTP

Atunci când un client contactează un server web via HTTP, are la dispoziție mai multe **metode** ale protocolului. Cea implicită, prin care am transmis până acum parametri prin *query string*-uri sau rute și prin care nu am modificat conținutul serverului, a fost metoda GET.

Metoda pe care o vom folosi pentru a modifica datele de pe server este POST, ce trebuie menționată explicit. Aici parametrii nu sunt vizibili, de pildă, în bara de adrese, din motive de securitate (de asemenea, browserul de obicei ne atenționează dacă vrem să repetăm trimiterea unei cereri POST). Modul prin care vom transmite acum parametrii necesari va fi, așa cum am anticipat, prin forme.

Vom detalia implementarea unei mecanism complet CRUD: **C**reate, **R**ead, **U**pdate, **D**elete. A doua componentă a fost deja discutată anterior – așadar, le vom descrie în continuare, în ordine, pe celelalte trei.

Create

Vom crea o acțiune numită `New` al cărei view va conține formularul de introducere a unui nou element în baza de date și una `Create` care va realiza efectiv introducerea. Pe pagina ce enumeră obiectele bazei de date vom pune o legătură către `New`. Acțiunea `New` nu va face decât să afișeze view-ul corespunzător, unde se va regăsi următoarea formă:

```
<form method="post" action="/NumeController/Create">
    <label>Title</label>
    <br />
    <input type="text" name="Title" />
    <br />
    <label>Author</label>
    <br />
    <input type="text" name="Author" />
    <br /><br />
    <button type="submit">Add</button>
</form>
```

Observații

Se observă următoarele:

- atributul `method` al formei arată că metoda HTTP folosită este POST
- atributul `action` arată că apăsarea butonului va declanșa apelarea acțiunii `Create`
- numele câmpurilor input au aceleași nume ca elementele clasei tabelului: astfel conținutul lor va putea fi transmis acțiunii `Create`

Model binding

Acțiunea `Create` (ce nu are un view asociat) va avea antetul:

```
public ActionResult Create(Book b)
```

Proprietățile parametrul `b` vor fi instanțiate cu conținutul câmpurilor corespunzătoare din forma ce a apelat acțiunea. Această facilitate se numește **model binding**.

Restul acțiunii Create

Acțiunii `Create` i se adaugă înaintea antetului său atributul `[HttpPost]`, ca ea să poată fi accesată doar via metoda POST și nu, de pildă, direct prin browser.

Acțiunea va adăuga obiectul respectiv în tabelul corespunzător:

```
ctx.Books.Add(b);
ctx.SaveChanges();
```

iar apoi va redirecționa browserul, de pildă, la acțiunea ce enumeră toate obiectele din tabel.

Update

Pentru a implementa facilitatea Update, vom crea două acțiuni, `Edit` și `Update`, ce au un rol analog lui `New` și `Create`.

Pe pagina corespunzătoare unui obiect din baza de date, introducem o legătură către acțiunea `Edit`, dându-i ca parametru id-ul obiectului. Apoi, ea va găsi obiectul în baza de date și va transmite rezultatul view-ului:

```
public ActionResult Edit(int id)
{
    Book book = ctx.Books.Find(id);
    return View(book);
}
```

Diferențele între view-urile New și Edit

View-ul acțiunii `Edit` va arăta ca cel al acțiunii `New`, cu următoarele deosebiri:

- va avea o directivă de model (deoarece i s-a transmis un obiect)
- va accesa acțiunea `Update`
- câmpurile vor conține deja conținutul proprietății corespunzătoare a obiectului ce va fi modificat, de pildă:

```
<input type="text" name="Title"
       value="@Model.Title" />
```

- forma va conține și un câmp ascuns pentru a reține id-ul obiectului:

```
<input type="hidden" name="BookId"
       value="@Model.BookId" />
```

Acțiunea Update

Acțiunea `Update` va opera pe obiectul corespunzător din baza de date:

```
Book book = ctx.Books.Find(b.BookId);
book.Title = b.Title;
book.Author = b.Author;
ctx.SaveChanges();
```

Delete

Vom crea o singură acțiune, `Delete`, pentru a implementa această facilitate. Dat fiind că ea va trebui accesată via metoda POST, se va crea o formă înzestrată cu buton special pentru ea, pe pagina corespunzătoare unui obiect.

Acțiunea va primi ca parametru doar id-ul respectiv (prin Razor; fără model binding), iar apoi va elimina acel element din tabel:

```
Book book = ctx.Books.Find(id);
ctx.Books.Remove(book);
ctx.SaveChanges();
```

HTML Helpers

Razor ne pune la dispoziție anumite metode, numite HTML Helpers, ce pot crea automat anumite taguri HTML.

Pentru a crea legături, folosim metoda `ActionLink`:

```
@Html.ActionLink("New book", "New")

@Html.ActionLink("More About " + p.Title, "Details",
                 new { id = p.BookId })
```

unde în a doua variantă transmitem și un parametru.

HTML Helpers pentru forme

În forme, putem vorbi de variantele *strongly typed* ale lor, care generează automat și atributele corespunzătoare – observăm că, pentru a le folosi, trebuie să avem o directivă model și în view-ul `New`.

```
@Html.LabelFor(m => m.Title)
@Html.TextBoxFor(m => m.Title)
@Html.HiddenFor(m => m.BookId)
```

Observăm și că ele primesc ca argument o funcție! Pentru a include și valori ale atributelor, ne folosim de următoarea variantă:

```
@Html.TextBoxFor(m => m.Title, new { value = Model.Title })
@Html.HiddenFor(m => m.BookId, new { value = Model.BookId })
```

Adnotări și validări

Despre adnotări

Adnotările de date reprezintă atribute ce sunt puse înaintea antetelor proprietăților claselor-model, de pildă cele pe care le-am văzut că indică restricții posibile asupra modului cum proprietățile devin coloane în tabelele unei baze de date.

Unele din ele se regăsesc în namespace-ul `System.ComponentModel.DataAnnotations`, pe care îl putem include eventual ca o directivă `using`.

Adnotarea Display

Putem, de exemplu, să modificăm numele ce va fi afișat (dacă folosim HTML Helpers, firește) ca denumire a unei proprietăți, folosind adnotarea `Display`, anume în felul următor:

```
[Display(Name="Titlu")]
```

Adnotările, însă, pot fi folosite și pentru validări.

Adnotări de validare

Următoarele adnotări (și nu numai) pot fi folosite și pentru validare:

```
[Required]
[StringLength(4)]
[Range(5,25)]
[RegularExpression(...)]
```

Codul ce realizează validarea mai are, însă, două componente.

Codul din acțiune

În acțiunea ce realizează efectiv modificarea bazei de date (cea care este accesată prin metoda POST), ne folosim de proprietatea `ModelState` pentru a verifica dacă datele transmise sunt valide; în caz că nu, ne întoarcem la view-ul ce conține formularul de editare, căruia nu uităm să îi transmitem obiectul ale cărui proprietăți vor fi afișate în câmpurile formularului:

```
if (!ModelState.IsValid)
    return View("Edit", b);
```

Codul din view

Înăuntrul view-ului vom introduce noi HTML Helpers ce vor servi la afișarea mesajelor ce raportează erori de validare:

```
@Html.ValidationMessageFor(m => m.Title)
```

Mesaje personalizate

Dacă vrem ca mesajele respective să nu fie cele implicite, mai adăugăm un parametru adnotării respective:

```
[Required(ErrorMessage = "Not good")]
[Range(5, 25, ErrorMessage = "Not good")]
```

Adnotări și validări personalizate

Uneori dorim ca și regula de validare să fie una personalizată. Pentru aceasta, creăm o clasă nouă (să-i zicem `ValidB`) ce va moșteni clasa `ValidationAttribute` (din namespace-ul `System.ComponentModel.DataAnnotations`, pe care iarăși îl includem ca o directivă `using`). Să spunem că vrem să testăm dacă proprietatea respectivă conține un număr par:

```
protected override ValidationResult
        IsValid(object value,
                ValidationContext validationContext)
{
    var book = (Book)validationContext.ObjectInstance;
    bool cond = book.Number % 2 == 0;
    return cond ? ValidationResult.Success
                : new ValidationResult("Not good");
}
```

În final, înainte de proprietatea `Number` din clasa `Book` includem atributul `[ValidB]`.

Sumar de validare

Dacă vrem să afișăm un sumar al tuturor erorilor de validare, folosim următorul HTML Helper:

```
@Html.ValidationSummary()
```

Dacă vrem doar să sesizăm existența erorilor, folosim următoarea formă a sa:

```
@Html.ValidationSummary(true, "Please fix the following")
```

Autentificare

Activarea autentificării

Pentru a activa funcționalitatea inclusă pentru autentificare (ASP.NET Identity), la crearea unui nou proiect selectăm, în variantele pentru setarea Authentication, opțiunea Individual User Accounts.

Consecințe

Se observă că:

- Entity Framework este deja inclus, odată cu multe alte referințe
- este creată o bază de date în folderul `App_Data` dar care nu apare în Solution Explorer
- apare un connection string pentru acea bază de date nou-creată
- există o funcționalitate de autentificare și înregistrare susținută de cod de tip model, view, controller.

Autorizarea acțiunilor

Se pot restricționa la utilizatorii autentificați:

- o acțiune, prin atributul `[Authorize]` pus înaintea antetului ei;
- toate acțiunile dintr-un controller, prin atributul `[Authorize]` pus înaintea antetului lui;
- toate acțiunile aplicației, prin codul

  ```
  filters.Add(new AuthorizeAttribute());
  ```

  pus în metoda `RegisterGlobalFilters` din clasa `FilterConfig`, din fișierul `App_Start\FilterConfig.cs`.

Atributul `[AllowAnonymous]` suprascrie, pentru o acțiune sau un controller, o restricție ca mai sus aplicată la un nivel superior.

Roluri

Pentru a aloca utilizatorilor la înregistrare un anumit rol, adăugăm un cod precum următorul în metoda `Register` (cea cu `HttpPost`) din controllerul `Account`, anume în ramura de cod `result.Succeeded`:

```
var roleStore = new RoleStore<IdentityRole>(new
                ApplicationDbContext());
var roleManager = new
                  RoleManager<IdentityRole>(roleStore);
if (!roleManager.RoleExists("role_name"))
   roleManager.Create(new IdentityRole("role_name"));
UserManager.AddToRole(user.Id, "role_name");
```

La laborator se va implementa și funcționalitatea schimbării rolurilor de către administrator.

Accesarea rolurilor

Putem adăuga parametrul `Roles` la atributul `Authorize` pentru a restrânge mai mult aria de restricție, anume la utilizatorii ce au anumite roluri (separate prin virgulă):

```
[Authorize(Roles = "Super")]
[Authorize(Roles = "User,Admin")]
```

Într-o acțiune oarecare, putem testa dacă utilizatorul curent este într-un anumit rol astfel:

```
if (User.IsInRole("Super")) s = " Super";
```

Baza de date

Clasa-context pentru baza de date a utilizatorilor este `ApplicationDbContext` (aflată în fișierul `IdentityModels.cs`). Pentru a putea crea legături între tabelele noastre și tabelele de autentificare, le vom pune și pe cele dintâi în aceeași clasă-context.

O astfel de legătură poate fi de pildă una many-to-one care să ne indice utilizatorul ce a creat elementul respectiv:

```
public string UserId { get; set; }
public virtual ApplicationUser User { get; set; }
```

În continuare, vom vedea cum se poate extinde CRUD-ul cu funcționalitatea de mai sus.

CRUD și utilizatori: citirea

De exemplu, pentru a include numele utilizatorului pe pagina de detalii, în acțiune e necesar să facem un join cu ajutorul LINQ:

```
Book book = ctx.Books.Include("User").Single(b =>
                b.BookId == id);
```

iar în view vom accesa proprietatea via `@Model.User.UserName`.

CRUD și utilizatori: modificarea

În acțiunea `New`, în acel model vid pe care îl transmitem vom include și id-ul utilizatorului ce adaugă obiectul:

```
book.UserId = User.Identity.GetUserId();
```

iar în view îl vom include sub forma unui câmp ascuns:

```
@Html.HiddenFor(m => m.UserId)
```

CRUD și utilizatori: restricționarea

Putem, firește, să restrângem drepturile de modificare pentru diversele roluri de utilizatori.

De pildă, putem face ca administratorii să poată modifica toate obiectele, anumiți utilizatori să nu poată modifica niciun obiect, iar alți utilizatori să poată modifica doar obiectele create de ei.

Toate acestea se pot implementa folosind doar metodele și proprietățile descrise anterior.

ASP.NET Web API

Proiecte Web API

Pentru a crea un proiect care permite controllere de tip Web API, se selectează, ca tip de aplicație, Web API în loc de MVC.

Rutarea

Informațiile despre rutare se pot găsi în metoda `Register` din clasa `WebApiConfig`, aflată în fișierul `App_Start\WebApiConfig.cs`.

Se observă că rutarea implicită are forma:

```
api/{controller}/{id}
```

Numele acțiunii (metodei) accesate nu se găsește în URL. Acțiunea va fi selectată în funcție de metoda HTTP a cererii.

Controllerul Values

Se observă că în proiect există un controller API implicit numit `ValuesController`, ce nu va mai moșteni clasa `Controller`, ci clasa `ApiController`.

El conține metode ale căror nume încep cu numele metodelor HTTP corespunzătoare. Această convenție se poate suprascrie prin atributele deja introduse: `HttpGet`, `HttpPost` etc.

Accesarea prin browser a URL-ului `api/values` va produce o accesare a metodei `Get`, care, după cum se vede, returnează un obiect CLR în format XML.

Testare

Pentru a putea testa toate metodele aplicațiilor Web API, putem folosi programul Insomnia, ce se poate descărca de la adresa:

https://insomnia.rest/download/core/

O cerere finalizată cu succes va returna **aici** datele în format JSON și va transmite în mod implicit codul HTTP 200.

Parametri

Parametrii unei acțiuni Web API se pot transmite în două feluri:

- prin URI (prin rută sau prin *query string*, ca până acum)
- prin corpul (*body*) al cererii (îl puteți vedea în Insomnia)

Implicit, tipurile simple se transmit prin URI, iar cele complexe prin corpul cererii. Acest comportament se poate suprascrie prin atributele `[FromBody]`, `[FromUri]` (vedeți chiar și în controllerul `Values`).

CRUD cu Web API

În continuare, descriem cum putem implementa CRUD prin acțiuni de tip Web API.

Adăugăm așadar un nou controller, denumit aici `BooksController`, dar de tip "Web API 2 Controller – Empty".

Vom crea baza de date ca mai devreme și vom include în controller un obiect ce instanțiază clasa context.

GET (Read)

Vom avea două acțiuni GET, una fără parametri ce returnează lista tuturor obiectelor și una cu parametru `id` ce returnează un anume obiect:

```
public List<Book> Get()
{
    return ctx.Books.ToList();
}
public IHttpActionResult Get(int id)
{
    Book book = ctx.Books.Find(id);
    if (book == null) return NotFound();
    return Ok(book);
}
```

După cum se vede, metodele pot returna un rezultat de tip `IHttpActionResult`, pentru care avem la dispoziție helperul `Ok`, ce returnează obiectul dat ca parametru cu codul de succes 200. Helperul `NotFound` va returna, firește, codul 404.

POST (Create)

Acțiunea corespunzătoare POST va returna o valoare dată de helperul `Created` ce conține, pe lângă obiectul nou-creat, și adresa la care el va fi găsit. Ele vor fi returnate sub codul de succes 201.

```
public IHttpActionResult Post([FromBody] Book b)
{
    ctx.Books.Add(b);
    ctx.SaveChanges();
    var uri = new Uri(Url.Link("DefaultApi",
                new { id = b.BookId }));
    return Created(uri, b);
}
```

Parametrul ei este preluat din corpul cererii și de aceea trebuie introdus în format JSON:

```
{
        "Title" : "test1",
        "Author" : "test2"
}
```

PUT (Update)

Acțiunea corespunzătoare PUT funcționează similar. Ea va prelua din URL id-ul obiectului ce trebuie modificat iar din corp datele noi:

```
public IHttpActionResult Put(int id, [FromBody] Book b)
{
    Book book = ctx.Books.Find(id);
    if (book == null) return NotFound();
    book.Title = b.Title;
    book.Author = b.Author;
    ctx.SaveChanges();
    return Ok(book);
}
```

DELETE (Delete)

Acțiunea corespunzătoare DELETE va prelua doar id-ul obiectului ce trebuie șters.

```
public IHttpActionResult Delete(int id)
{
    Book book = ctx.Books.Find(id);
    if (book == null) return NotFound();
    ctx.Books.Remove(book);
    ctx.SaveChanges();
    return Ok(book);
}
```

AJAX, jQuery și DataTables

Recapitulare – Tehnici web

- **AJAX** reprezintă folosirea XMLHttpRequest în codul JavaScript pentru a accesa date de pe server cu scopul modificării conținutului paginii web fără a o reîncărca

- **jQuery** este o bibliotecă JavaScript folosită în genere la a simplifica lucrul cu DOM, evenimente, AJAX
  - referință pe scurt: https://www.impressivewebs.com/jquery-tutorial-for-beginners/
  - este preinstalat în orice proiect ASP.NET – vezi folderul Scripts

CRUD cu AJAX și jQuery

În continuare, vom arăta cum putem accesa din view-uri CRUD-ul implementat în API, folosind jQuery.

Vom porni de la un proiect ce conține atât CRUD-ul implementat inițial, cât și cel cu API. Pentru a putea avea două controllere cu același nume, putem pune unul dintre ele într-un subfolder.

Vom considera că datele sunt afișate sub formă de tabel, nu de listă; pe slide-urile următoare detaliem cum anume.

Tabele

```
<table id="books"
       class="table table-bordered table-hover">
    <thead>
        <tr>
            <th>Title</th>
            <th>Details</th>
            <th>Delete</th>
        </tr>
    </thead>
```

Punem un id tabelului pentru a-l putea accesa mai ușor în DOM; clasele CSS servesc la randarea lui corectă de către Bootstrap.

Tabele

```
    <tbody>
        @foreach (Curs8.Models.Book p in Model)
        {
        <tr>
            <td> @p.Title</td>
            <td> @Html.ActionLink("More about " + p.Title,
                "Details", new { id = p.BookId })</td>
            <td> </td>
        </tr>
        }
    </tbody>
</table>
```

A treia coloană o păstrăm liberă pentru a putea pune acolo butonul de ștergere.

Butonul de ștergere

Vom pune butonului de ștergere o clasă (din nou, pentru a-l putea accesa mai ușor):

```
<button class="bt-delete">Delete item</button>
```

Codul JavaScript prin care îl vom accesa îl vom pune la sfârșitul fișierului de view:

```
@section scripts
{
    <script>
        $(document).ready(function () {
            $("#books").on("click", ".bt-delete",
                function () {
                confirm("Are you sure?"); }); });
    </script>
}
```

El trebuie plasat în secțiunea `scripts` pentru ca jQuery să poată fi folosit.

Accesarea API-ului

Pentru a putea accesa API-ul, avem nevoie să știm id-ul obiectului din baza de date și de aceea îl vom stoca în atributul `data-id` al butonului:

```
<button class="bt-delete" data-id="@p.BookId">
    Delete item</button>
```

Vom modifica codul din funcția anonimă de click astfel:

```
if (confirm("Are you sure?")) {
    $.ajax({
        url: "/api/books/" + $(this).attr("data-id"),
        method: "DELETE",
        success: function () {
            console.log("Success");
        }
    });
}
```

Ștergerea liniei din tabel

În final, vrem să ștergem și linia din tabelul afișat. Pentru aceasta, avem nevoie de o referință la buton, pe care o punem înainte de `if`:

```
var but = $(this);
```

Apoi, în funția anonimă de succes (unde nu putem folosi `this` pentru a ne referi la buton, fiind în alt domeniu de vizibilitate), scriem:

```
but.parents("tr").remove();
```

Eventual, putem modifica și prima referință la buton în:

```
but.attr("data-id")
```

DataTables

Pentru a încărca date prin AJAX în tabel, vom folosi biblioteca DataTables. O vom instala prin NuGet Package Manager, ca pe Entity Framework (pachetul se numește jquery.datatables).

Se observă că avem fișierele noi în Scripts. Pentru a le putea accesa, în fișierul `App_Start\BundleConfig.cs`, vom adăuga la bundle-ul `~/bundles/jquery` fișierele:

```
"~/scripts/datatables/jquery.datatables.js"
"~/scripts/datatables/datatables.bootstrap.js"
```

iar la bundle-ul `~/Content/css` fișierul:

```
"~/content/datatables/css/datatables.bootstrap.css"
```

Pentru a face ca tabelul nostru să fie de forma `DataTable`, la începutul funcției anonime accesate prin `ready` adăugăm:

```
$("#books").DataTable();
```

Date în tabel prin API (Read)

În primul rând, vom șterge codul existent în tagul `tbody` (acel `foreach`). Apoi, vom înlocui apelul funcției `DataTable` cu următorul:

```
$("#books").DataTable({
    ajax: { url: "/api/books", dataSrc: "" },
    columns: [
        { data: "Title" },
        { data: "Title",
          render: function (data, type, row) {
           return "<a href='/books/details/" +
                  row.BookId + "'>More about " +
                  row.Title + "</a>"; } },
        { data: "BookId", render: function (data) {
            return "<button class='bt-delete' data-id=" +
                   data + ">Delete item</button>"; } }
    ]
});
```

Ștergerea liniei din DataTable

În final, trebuie să vedem cum putem șterge linia într-un mod corespunzător DataTable. Apelul existent

```
but.parents("tr").remove();
```

nu funcționează corect (putem testa, ștergând o linie, ea apărând apoi la căutare).

Vom prelua o referință la tabel în momentul apelului funcției `DataTable`:

```
var table = $("#books").DataTable({...
```

pe care o vom folosi în locul apelului citat mai sus, astfel:

```
table.row(but.parents("tr")).remove().draw();
```

Caching, Filtre

Atribute

Am văzut deja cum putem pune atribute în fața acțiunilor care să le modifice comportamentul, de exemplu:

- atribute ce denotă metode HTTP;
- atribute de autorizare.

În continuare vom prezenta alte asemenea atribute (filtre).

Output Caching

De exemplu, pentru a stoca rezultatul unei acțiuni în *cache*, scriem înaintea antetului ei atributul `OutputCache`:

```
[OutputCache(Duration = 5)]
```

Numărul reprezintă durata, în secunde, pentru care rezultatul este stocat.

Pentru a testa comportamentul, introducem în view o afișare a timpului curent:

```
@DateTime.Now.ToString()
```

Data Caching

Pentru a stoca date în *cache*, adăugăm referința `System.Runtime.Caching.dll` (dând drept-click pe folderul References din Solution Explorer și selectând Add Reference). Ne este convenabil apoi să adăugăm `System.Runtime.Caching` ca directivă `using`.

Putem folosi câmpurile dicționarului `MemoryCache.Default` ca pe niște variabile globale, statice:

```
if (MemoryCache.Default["i"] == null)
    MemoryCache.Default["i"] = 0;
int current = (int)MemoryCache.Default["i"];
current++;
MemoryCache.Default["i"] = current;
return View(current);
```

Debug

În namespace-ul `System.Diagnostics` există obiectul `Debug`, cu care putem să tipărim mesaje în consola de debug, în modul următor (doar când modul Debug este pornit, adică atunci când rulăm cu F5):

```
Debug.WriteLine("mesaj");
```

Vom crea în continuare filtre personalizate și vom folosi această facilitate pentru a le testa.

Filtre personalizate

Un filtru personalizat va fi o clasă (pentru simplificare, o considerăm tot ca model) ce va moșteni clasa `ActionFilterAttribute` din namespace-ul `System.Web.Mvc`.

Vom considera în continuare că această clasă se numește `TheFilter` și că în fișierul ei avem directive `using` pentru `System.Web.Mvc` și `System.Diagnostics`.

Acest filtru va fi aplicat unei acțiuni scriind înaintea antetului ei atributul `[TheFilter]`.

Metodele filtrelor personalizate

Putem suprascrie metodele clasei de bază prin metode ce vor juca rol de evenimente și vor avea antete ce încep cu `public override void` la care se adaugă:

```
OnActionExecuting(ActionExecutingContext filterContext)
OnActionExecuted(ActionExecutedContext filterContext)
OnResultExecuting(ResultExecutingContext filterContext)
OnResultExecuted(ResultExecutedContext filterContext)
```

Ele vor fi rulate, respectiv, în următoarele momente:

- înainte de acțiunea căreia îi este aplicat filtrul;
- imediat după acțiune;
- înainte de rularea `ActionResult`-ului corespunzător;
- imediat după el.

Informații din parametri

Obiectele ce sunt transmise ca parametri acestor acțiuni conțin informații utile. De exemplu, în `OnActionExecuting` avem următoarele:

```
filterContext.HttpContext.Request.Url
filterContext.HttpContext.Request.UserHostAddress
filterContext.HttpContext.Request.UserAgent
filterContext.RouteData.Values["controller"]
filterContext.RouteData.Values["action"]
```

iar în `OnActionExecuted` putem vedea modelul transmis view-ului:

```
((ViewResultBase)filterContext.Result).Model
```

Facebook for Developers

Autentificare cu Facebook

Vom descrie acum facilitatea de autentificare cu Facebook. Documentația sa se poate găsi la adresa:

https://developers.facebook.com/docs/facebook-login

Vom crea un proiect nou care are activat Individual User Accounts și vom crea un cont pe Facebook for Developers, la adresa https://developers.facebook.com (site pe care vom dezactiva orice eventual ad blocker).

În secțiunea My Apps, vom crea o nouă aplicație (Create an App) și vom bifa Build Connected Experiences – apoi, vom putea da un nume aplicației.

Autentificare cu Facebook

În panoul de control (Dashboard) al aplicației, există secțiunea Facebook Login. Apăsăm Set Up și apoi www. Introducem adresa site-ului – precum `https://localhost:44335/` – în câmpul de acolo.

În meniul din stânga, selectăm Settings, Basic. De acolo vom prelua informațiile de la App ID și App Secret pentru a le putea folosi în aplicația web.

Autentificare cu Facebook

În fișierul `App_Start\Startup.Auth.cs` vom introduce aceste date, mai precis vom decomenta liniile ce conțin apelul la `UseFacebookAuthentication` și vom introduce în acel apel parametrii obținuți de pe site, astfel:

```
app.UseFacebookAuthentication(
   appId: "428741541836078",
   appSecret: "b5a73a1ca4240d47cb786d28b12f7dee");
```

Instagram Basic Display

Arătăm în continuare cum se poate folosi facilitatea Instagram Basic Display. Documentația API-ului se poate găsi la adresa:

https://developers.facebook.com/docs/instagram-basic-display-api/

Ca la Facebook Login, apăsăm în Dashboard pe butonul Set Up de la Instagram Basic Display. Pe pagina care apare, apăsăm pe Create New App (deoarece trebuie să creăm o aplicație Instagram).

Vom introduce, ca nume, numele aplicației Facebook, iar ca URL-uri introducem un URL standard ce returnează codul de succes 200, cum ar fi:

https://httpstat.us/200

Salvăm modificările. Notăm ID-ul și secretul asociate aplicației Instagram.

Instagram Basic Display

În continuare vom adăuga un utilizator Instagram ca tester. În meniul din stânga, selectăm Roles, Roles. La secțiunea Instagram Testers, adăugăm un handle de Instagram.

Ne autentificăm apoi pe Instagram cu acel handle, iar în setările Instagram, la Apps and Websites, Tester Invites, aprobăm aplicația creată mai devreme.

Instagram Basic Display

Vom accesa apoi din browser un URL precum următorul:

```
https://api.instagram.com/oauth/authorize
  ?client_id=829873255464505
  &redirect_uri=https://httpstat.us/200
  &scope=user_profile,user_media&response_type=code
```

unde vom modifica valoarea lui `client_id` cu ID-ul aplicației noastre Instagram. În pagina ce apare, vom autoriza cererea și vom fi redirecționați la o pagină al cărei URL arată precum:

```
https://httpstat.us/200?code=AQB08cVLYKNqkjx7Rkc-GZ
5b8FnDO_K9J7b3dMYPBduvw9gdDnS1I40mGq8mk_dLtD1e7
s5_-UKe_9vErP_F5g69Q_PyQZ5AvOYbdqSbblqj-2g96Wtdcx21X
bjz4KWdwHb8h5i7wC86SKzZRMeQLR4mcIgaQ9JgMATOmIFoj8_C1a
ifr9JgzLS4s49R1QlNJocPGonzJkdKfZrE9azLcWX6S5i4rLnd
D15nnh1WEgwbSA#_
```

Valoarea parametrului `code`, exceptând cele două caractere de final `#_`, va reprezenta un cod de autorizare.

Instagram Basic Display

În continuare vom cere un token de acces. Vom apela o cerere de tip POST către adresa

`https://api.instagram.com/oauth/access_token`

și ca urmare vom folosi Insomnia. Vom selecta ca formă a corpului cererii Form URL Encoded, iar acolo vom crea următoarele câmpuri:

- `client_id` – ID-ul aplicației Instagram
- `client_secret` – secretul aplicației Instagram
- `grant_type` – `authorization_code`
- `redirect_uri` – `https://httpstat.us/200`
- `code` – codul de autorizare de pe slide-ul anterior

Vom primi în răspuns, sub formă JSON, un `access_token` și un `user_id`, pe care le vom nota.

Instagram Basic Display

Acum vom cere un token de acces de lungă durată. Vom apela o cerere de tip GET, ceea ce putem face și în browser. Dacă totuși dorim să o facem în Insomnia, vom avea grijă să punem URL-ul ca mai jos și să lăsăm corpul cererii gol.

URL-ul va fi de forma:

```
https://graph.instagram.com/access_token
   ?grant_type=ig_exchange_token
   &client_secret=7e211cf63693d9ba1b41dece32215e53
   &access_token=IGQVJWeGY...
```

unde vom modifica valoarea lui `client_secret` cu secretul aplicației noastre Instagram, iar pe cea a lui `access_token` cu tokenul recepționat anterior. Vom primi un nou token, precum și perioada lui de valabilitate (5184000 de secunde, adică 60 de zile).

Instagram Basic Display

Acum arătăm cum putem accesa efectiv API-ul. Vom apela o nouă cerere de tip GET, de forma:

```
https://graph.instagram.com/me
    ?fields=id,username,media_count
    &access_token=IGQVJWeGY...
```

unde token-ul de acces va fi cel obținut anterior (de scurtă sau de lungă durată). Vom primi ca răspuns `id`-ul și `username`-ul utilizatorului Instagram ce a autorizat mai devreme accesul (identificat după token), precum și numărul de postări ale sale.

Instagram Basic Display

Dacă accesăm un URL de forma:

```
https://graph.instagram.com/me/media
    ?fields=id,timestamp
    &access_token=IGQVJWeGY...
```

vom primi o listă cu `id`-urile postărilor utilizatorului respectiv, precum și cu momentele lor corespunzătoare.

Instagram Basic Display

Ca să primim înapoi URL-ul corespunzător conținutului unei postări (de exemplu, o poză), vom accesa:

```
https://graph.instagram.com/18010305792862624
?fields=media_url
&access_token=IGQVJWeGY...
```

unde acel număr va fi `id`-ul recepționat anterior.

Accesarea API-ului Instagram

Acum vom exemplifica accesarea acestui API dintr-o aplicație web. Vom avea un controller cu două acțiuni, `Index` și `Show`.

Acțiunea `Index` (ce nu are view asociat) va folosi metoda `Redirect` pentru a accesa URL-ul prin care vrem să obținem codul inițial de autorizare, **însă** vom pune în parametrul `redirect_uri` adresa acțiunii `Show` (**de asemenea**, trebuie să avem grijă ca acea adresă să fie trecută în setările aplicației la secțiunea Valid OAuth Redirect URIs).

Accesarea API-ului Instagram

Așadar, acțiunea `Show` va primi ca parametru acel `code`. Ea va transmite către view-ul său o listă de string-uri ce vor reprezenta URL-urile imaginilor ce vor fi afișate de acesta. View-ul va avea, deci, următoarea formă:

```
@model List<string>
@foreach (var p in Model)
{
    <img src="@p" height="300" />
}
```

Accesarea API-ului Instagram

Rămâne de explicat codul acțiunii `Show`. Vom folosi clasa `WebClient`, aflată în namespace-ul `System.Net`:

```
WebClient cl = new WebClient();
```

Pentru a trimite parametrii sub forma Form URL Encoded, vom specifica acest lucru în header-ul clientului:

```
cl.Headers[HttpRequestHeader.ContentType] =
      "application/x-www-form-urlencoded";
```

Vom stoca parametrii ca *query string* (adăugând parametrul `code`) și vom folosi metoda `UploadString` a clientului pentru a-i transmite via POST. Rezultatul cererii va fi stocat ca string.

```
string para = "redirect_uri=https://
 localhost:44388/Home/Show&app_id=1&app_secret=2
 &grant_type=authorization_code&code=" + code;
string HtmlResult = cl.UploadString("https://
 api.instagram.com/oauth/access_token", para);
```

Accesarea API-ului Instagram

Acum va trebui să transformăm stringul JSON într-un dicționar cheie-valoare. Pentru aceasta, folosim metoda de deserializare a clasei `JsonConvert`, situată în namespace-ul `Newtonsoft.Json`. Din acest dicționar, vom prelua token-ul de acces.

```
Dictionary<string, string> dic =
 JsonConvert.DeserializeObject
 <Dictionary<string, string>>
 (HtmlResult);
string access_token = dic["access_token"];
```

Acum putem, ca mai devreme, să accesăm efectiv API-ul. Vom stoca într-o listă URL-urile imaginilor ce vor fi afișate:

```
List<string> list_mediaurls = new List<string>();
```

Accesarea API-ului Instagram

Ca mai devreme, accesăm URL-ul corespunzător pentru a obține lista `id`-urilor postărilor (însă acum vom folosi metoda HTTP GET, ca urmare folosim metoda `DownloadString` a clientului):

```
HtmlResult = cl.DownloadString(
 "https://graph.instagram.com/
 me/media?access_token="
 + access_token);
```

După cum am văzut în Insomnia, acest dicționar JSON nu este uniform, așadar tipul valoare va fi aici unul generic `object`:

```
Dictionary<string, object> dic2 =
 JsonConvert.DeserializeObject
 <Dictionary<string, object>>
 (HtmlResult);
```

Accesarea API-ului Instagram

Pe noi ne interesează câmpul `data`, așadar pe acela îl vom converti la `string` și îl vom deserializa în continuare la o listă de dicționare obișnuite:

```
List<Dictionary<string, string>> list =
 JsonConvert.DeserializeObject
 <List<Dictionary<string, string>>>
 (dic2["data"].ToString());
```

Accesarea API-ului Instagram

Mai apoi, ce avem de făcut este să accesăm fiecare postare și să îi preluăm URL-ul:

```
foreach (Dictionary<string, string> p in list)
{
 HtmlResult = cl.DownloadString(
  "https://graph.instagram.com/" + p["id"] +
  "?fields=media_url&access_token=" + access_token);
 Dictionary<string, string> dic3 =
  JsonConvert.DeserializeObject
  <Dictionary<string, string>>(HtmlResult);
 list_mediaurls.Add(dic3["media_url"]);
}
```

La final, vom transmite lista de URL-uri view-ului:

```
return View(list_mediaurls);
```