VISOKA ŠKOLA
ELEKTROTEHNIKE I RAČUNARSTVA
STRUKOVNIH STUDIJA

Zoran Ćirović

PROGRAMIRANJE APLIKACIJA BAZA PODATAKA

Visoka škola elektrotehnike i računarstva
strukovnih studija u Beogradu, 2019

Naslov:	Programiranje aplikacija baza podataka
	1. izdanje
Autor:	dr Zoran Ćirović
Recenzenti:	dr Svetlana Štrbac-Savić
	mr Miloš Pejanović
Izdavač:	Visoka škola elektrotehnike i računarstva strukovnih studija u Beogradu
Tehnička obrada:	Zoran Ćirović
Korice:	Ana Miletić
Štampa:	Razvojno-istraživački centar grafičkog inženjerstva TMF, Beograd
Tiraž:	20
ISBN:	978-86-7982-307-6

Nastavno-stručno veće Visoke škole elektrotehnike i računarstva, na svojoj sednici 18.4.2019. godune odobrilo je izdavanje i korišćenje ovog udžbenika u nastavi.

CIP- Каталогизација у публикацији
Народна библиотека Србије

004.9(075.8)
004.65(075.8)

ЋИРОВИЋ, Зоран, 1970-
Programiranje aplikacija baza podataka / Zoran Ćirović. - 1. izd. - Beograd : Visoka škola elektrotehnike i računarstva strukovnih studija, 2019 (Beograd : Razvojno-istraživački centar grafičkog inženjerstva TMF). - 281 str. : ilustr. ; 25 cm

Tiraž 20. - Bibliografija: str. 268-270. - Registri.

ISBN 978-86-7982-307-6
а) Апликативни софтвер б) Базе података
COBISS.SR-ID 276297740

Predgovor

Knjiga „Programiranje aplikacija baza podataka” najvećim delom pokriva gradivo istoimenog predmeta koji se sluša na drugoj godini master studija Visoke škole za elektrotehniku i računarstvo strukovnih studija u Beogradu.

Knjiga je podeljena u četrnaest poglavlja koja postepeno uvode nove pojmove počev od osnovnih pojmova i instalacije razvojnog alata, preko ADO.NET biblioteke pa sve do praktičnih projektnih šablona veb aplikacija.

Prvo poglavlje je posvećeno uvođenju osnovnih pojmova u radu sa aplikacijama baza podataka. U okviru ovog poglavlja pokazuje se i instalacija neophodnih alata za pisanje aplikacija kao i preuzimanje tipičnih baza za rad. Prikazuju se i osnove okruženja za upravljanje bazom.

Drugo poglavlje je posvećeno ADO.NET biblioteci. Ovde se korisnik vodi kroz osnovne pojmove ove biblioteke. Uvodimo pojam objekata konekcije i konekcijskog stringa, zatim radimo objekat komande, vrste komande i načini njihove upotrebe, na kraju se uvodi DataSet i DataAdapter.

Treće poglavlje detaljno se bavi jezikom LINQ. U okviru ovog poglavlja detaljno se prikazuje ne samo sintaksa i upotreba jezika već i njegova veza sa standardnom sintaksom uvođenjem metoda proširenja. Rade se operatori različitih vrsta i daju primeri upotrebe, uvode se lambda izrazi kao i anonimni tipovi.

U četvrtom poglavlju je prezentovan radni okvir Entity Framework. Prikazani su osnovni koncepti i struktura modela u ovom okruženju. Na praktičnim primerima je urađeno kreiranje modela, opis zapisa jednog modela, prikazani su opisi entiteta, asocijacija i mapiranja kao i njihova upotreba u kodu. Zatim je prikazan rad sa klasama modela, kako se vrši dobavljanje podataka, izmena na podacima, dodavanje i brisanje. Posebno je opisana konekcija do baze kao i postojanje podrazumevanih transakcija.

U glavi pet prikazan je rad sa EDM modelima kroz praktične primene. Objektno relaciono mapiranje je demonstrirano kor nekoliko praktičnih primera upotrebe koje uključuju i izmene konceptualnog modela korišćenjem diskriminativnih polja ili podele jednog entiteta na više. Prikazani su i primeri rada sa složenim upitima, kreiranje kompleksnih tipova.

Šesta glava je posvećena MVC arhitekturi i izradi osnovne aplikacije koristeći odgovarajući šablon. Opisana je struktura foldera kao i upotreba HTML stranica, zatim podešavanja i upotrebu razvojnog servera za testiranje.

U sedmoj glavi obrađuju se posebno komponente kontroleri. Pokazano je kreiranje kontrolera, njihovo testiranja, upotreba parametara u metodama. Takođe se obrađuju tehnike rutiranja, način adresiranja i primena višestrukih ruta kao i primena pomoćnih metoda za zaštitu od prosleđivanja zlonamernih podataka.

Nakon kontrolera u osmoj glavi detaljno se obrađuje komponenta pogledi. Definiše se podrazumevani pogled, načini prenosa parametara

od metode kontrolera do pogleda, kreiranje standardnih i parcijalnih pogleda. Uvodi se pojam anotacija i neke od njih se objašnjavaju.

Deveto poglavlje je u potpunosti posvećeno sintaksi Razor koja se primenjuje za povezivanje pogleda sa pozadinskim kodom. Najpre se kreće od osnovog umetanja vrednosti, preko blokovskog koda i izraza. Zatim se prikazuje rad sa pomoćnim metodama i kreiranje sopstvenih. Na kraju se uvode tehnike rada sa nizovima, čitanje iz serverskih fajlova, rad sa formama kao i metode za organizaciju pogleda.

Deseto poglavlje je posvećeno radnom okviru Bootstrap (BS). Pošto su pogledi zasnovani na primeni ovog radnog okvira, neophodno je bilo prikazati u posebnom poglavlju rad sa ovim okruženjem. Ovom glavom obuhvaćeno je: prikaz strukture radnog okvira, šablon primene odnosno povezivanja pogleda i BSa. Zatim su prikazane klase za rad sa mrežastom strukturom, medija upitima, ugnježdavanja, prikaz odnosno sakrivanje sadržaja i još puno drugih klasa.

Jedanaesto poglavlje je posvećeno aplikacijama koje su povezane sa već postojećim bazama podataka, tzv. konceptu *Database First*. Za ovaj koncept aplikacije je prikazan način kreiranja kontrolera i pogleda, ali i izmena modela i ažuriranje podataka preko modela. Definisani su različiti tipovi filtera, a na kraju je prikazan i rad sa povezanim podacima.

Dvanaesto poglavlje uvodi novu tehniku izrade aplikacija povezanih sa podacima zasnovanu na početnom formiranju modela u kodu, a ne na postojanju neke baze. U okviru ovog poglavlja prikazuje se povezivanje modela i nekog servera baze, izmene na modelima, primena Scaffold mehanizma za generisanje šablonskog koda.

Trinaesto poglavlje je posvećeno u celosti radu sa izmenama na podacima. Najpre su opisane tehnike migracije i one su primenjene pri kreiranju početnih podataka, zatim su napravljene izmene na modelu i kreiranje nove migracije. Na kraju je urađena promena dodavanjem nove funkcionalnosti, poput pretrage po više polja.

Četrnaesto poglavlje je posvećeno kreiranju samostalnih projekata tipa API biblioteka. Pokazano je kreiranje aplikacije od samog početka. Napravljeno je nekoliko metoda i objašnjeno adresiranje kao i format podataka koji se koristi za razmenu sa klijentskim kodom pisanim u JavaScript jeziku ili pomoću jQuery biblioteke.

Nakon svakog poglavlja data su pitanja i zadaci namenjeni proveri znanja i boljem razumevanju izložene materije.

Cilj knjige je da posluži kao udžbenik za sticanje osnovnih znanja iz programiranja aplikacija baza podataka, prvenstveno veb orijentisanih. Zato knjiga obiluje primerima, slikama, odgovarajućim kodom i dodatnim objašnjenjem.

Beograd, 2019. Autor

Sadržaj

1. Pojmovi i instalacija

U ovom poglavlju biće prezentovani osnovi pojmovi vezani za platformu .NET na kojoj će biti zasnovan rad i primeri u nastavku knjige. Nakon uvođena osnovnih pojmova, poglavlje prikazuje postupak instalacije razvojnog okruženja. Nakon toga se prezentuju pojmovi agilnog razvoja kao i konkretni šabloni projekata.

.NET

.NET (dot net) je proizvod softverske kompanije Microsoft. Veži za trenutnu i buduću orijentaciju u pogledu softvera iste kompanije, ali i drugih multinacionalnih kompanija. Platforma se danas razvija i u pravcu otvorenog koda, veb aplikacija, komponenti i servisno orijentisanih aplikacija.

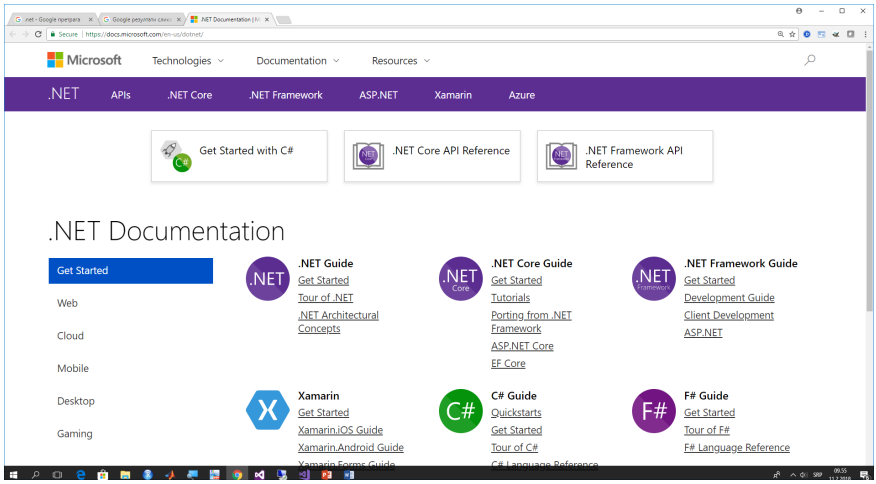
.NET je osnova za nove operativne sisteme. Koristi se u visoko pouzdanim sistemima za kontrolu leta. Omogućava razvoj univerzalnih Windows aplikacija. Omogućava razvoj i univerzalnih mobilnih aplikacija (iOS, Android, WP). Otvorena specifikacija omogućila je realizaciju i na operativnim sistemima koji nisu Windows.

Novi stil stvaranja kolekcija specifičnog koda na jednom mestu umesto tradicionalnog gde se stvaraju redundantne kopije na mnogo mesta. Pruža programske celine raspoložive različitim uređajima. Pomera razvoj

aplikacija od objektno orijentisanih ka servisnim (eng. *services privided).* Jezička nezavisnost postignuta je postojanjem međujezika (IL ili MSIL).

Podrška za VEB (XML) servise. Jaka podrška za XML standard. Poboljšani pristup bazama podataka preko ADO.NET tehnike. Znatno unapređen rad sa dinamičkim Veb stranicama baziranim na ASP.NET tehnologiji. Umesto dll-ova uvodi se koncept sklopova. Rad pod različitim platformama zasniva se na postojanju **.NET Framework-a**.

.NET Framework sadrži **CLR** (eng. Common Language Runtime) kao i kolekcije klasa ovog okruženja. Svi programski jezici obuhvaćeni Microsoft VisualStudio.NET imaju zajedničku platformu i to ih čini ravnopravnim.



Slika 1.1. .NET dokumentacija https://docs.microsoft.com/en-us/dotnet/

Veb razvoj

Razvoj veb aplikacija napreduje brzo prateći potrebe programera odnosno krajnjih korisnika veba. Veb razvoj ide u nekoliko različitih

pravaca od trenutka kada su Veb Forme prvi put objavljene. Pogledajmo aktuelne standarde.

Standardi i REST

Usaglašavanje sa standardima Veba danas predstavlja gotovo obavezu u projektovanju aplikacija. Veb aplikacije se primenjuju na sve većoj raznolikosti uređaja i pretraživača nego ikada ranije, a veb standardi (HTML, CSS, JavaScript i tako dalje) ostaju važeći. Moderne veb platforme ne mogu sebi priuštiti da ignorišu potrebu programera da postoji usaglašenost sa veb standardima.

HTML5 je postao *mainstream* koji obezbeđuje veb programeru da se osloni na nove funkcije koje omogućavaju klijentskoj aplikaciji da obavlja posao koji je ranije bio isključivo odgovornost servera. Nove opcije kao i sve moćnije JavaScript biblioteka (AngularJS, jQuery, jQuery UI i Mobile) znači da su standardi postali sve važniji i predstavljaju kritičnu osnovu za sve bogatije Veb aplikacije.

Napomena: Mada se u ovom kursu koristi HTML5, jQuery, Bootstrap, nećemo se baviti detaljima u vezi ovih tehnologija. Objasnićemo osnovne elemente koje koristimo stavljajući naglasak na ASP MVC dok bi prateće tehnologije razmatrane detaljno predstavljale obimnije celine odnosno svaka bi mogla da bude zasebna tema.

Takođe treba pomenuti **REST** arhitekturu (eng. REpresentational State Transfer) koja je postala dominantna pri izradi aplikacija koje zahtevaju interoperabilnost zasnovanu na HTTP protokolu. REST arhitektura je zasnovana na definisanju resursa (URI) koji predstavljaju entitete iz stvarnog sveta i standardnim operacijama (HTTP metodama) koje su istovremeno i dostupne operacije na tim resursima. Na primer: metodom PUT može se postaviti nov entitet:

```
ttp://www.mytest.com/Products/kafa1
```

metodom DELETE može se izbrisati entitet:

`http://www.mytest.com/Customers/kupac1.`

Današnje Veb aplikacije ne koriste samo HTML protokol za dobavljanje i prosleđivanje podataka. Često aplikacije koriste i druge formate zajedno ili odvojeno, poput JSON ili XML formata (eng: JavaScript Object Notation, eXtensible Markup Language). Tako se na primer asinhroni rad na vebu tipično ostvaruje primenom AJAX tehnologije a asinhrona komunikacija preko REST servisa, identično za Veb servisa odnosno Veb aplikacija.

Agilni i razvojni testovi

Agilni i razvojni testovi nisu aktuelni samo za razvoj Veb aplikacija, već se odnosi generalno na softverske tehnologije. Razvoj softvera u celini se pomerio prema agilnim metodologijama. Pri tome, ovi pojmovi mogu označavati mnogo različitih stvari, ali u najvećoj meri radi se o softverskim projektima koji su prilagodljivi i koji omogućavaju efikasno planiranje i razvoj. Agilne metodologije idu ruku pod ruku sa nizom razvojnih praksi i alata (obično otvorenog koda) koji promovišu i pomažu ove prakse.

Testni razvoj tj. razvoj baziran na testovima (eng. Test-driven development - TDD) odnosno razvoj baziran na ponašanjima (eng. Behavior-driven development - BDD), jesu primeri dva agilna razvoja. Ideja je da se dizajnira softver tako što se prvo opišu primeri željenih ponašanja (poznatih kao testovi ili specifikacije). Na taj način se tokom razvoja, u bilo kom trenutku, može proveriti tačnost i stabilnost koda izvršavanjem paketa testova u odnosu na implementaciju.

- *Unit testing* alati omogućavaju da se navede ponašanje određenih klasa, metoda ili drugih komponenata koda. Mogu se efikasno primeniti na softver koji je dizajniran kao skup nezavisnih modula, tako da se svaki test može izvoditi zasebno.

- *UI automation* alatke omogućavaju simulaciju serije interaktivnih akcija i potpune simulacije rada aplikacije u određenim uslovima i za određene zadatke.

Radno okruženje

Razvojni alat za ASP MVC aplikacije je deo integrisanog razvojnog okruženja Visual Studio, počev od verzije 2012 pa novije. U ovom kursu koristićemo Visual Studio 2017.

Microsoft pruža besplatnu verziju Visual Studio-a, koja takođe sadrži i SQL Server. Ovu verziju možete preuzeti sa veb adrese:

https://www.visualstudio.com/downloads/.

Instalacija

Korak 1. Pre početka instaliranja potrebno je da proverite zahteve za instalaciju i resurse koji su na raspolaganju, pogledati detalje na linku: https://docs.microsoft.com/en-us/visualstudio/productinfo/vs2017-system-requirements-vs

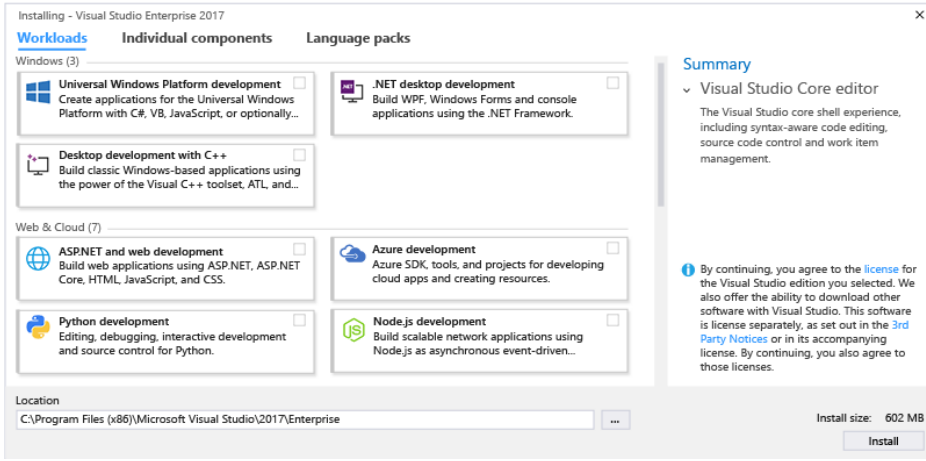
Korak 2. Skinuti sa mreže verziju radnog okruženja VS koja vam stoji na raspolaganju. Za edukativne svrhe brojne školske ustanove i fakulteti poseduju različite licence za učenje pa se informišite u vezi alata koji vam stoje na raspolaganju. U svakom slučaju dve besplatne verzije možete skinuti sa zvaničnog sajta koristeći link koji smo već naveli.

Korak 3. U folderu "Downloads" dvaput kliknite na odeljak koji odgovara ili je sličan jednoj od sledećih datoteka:

- vs_enterprise.exe za Visual Studio Enterprise
- vs_professional.exe za Visual Studio Professional

- vs_communiti.exe za Visual Studio Community

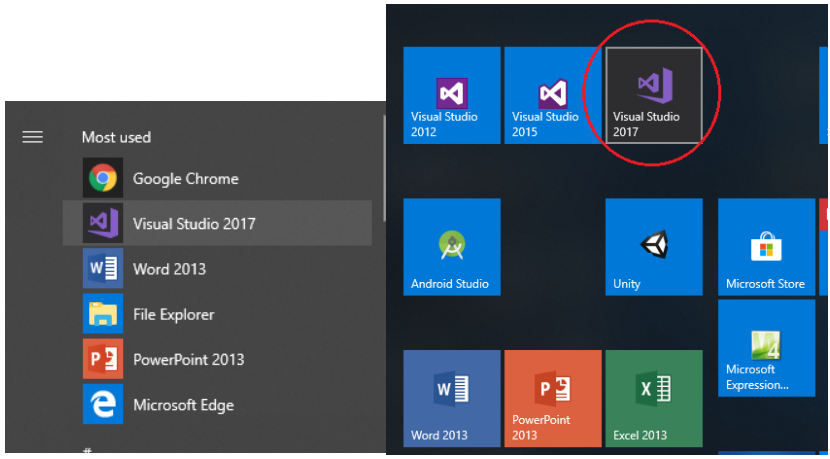
Korak 4 – Bira se vrsta inicijalne instalacije. VS je veoma veliko i moćno okruženje. Najverovatnije nemate potrebe da baš sve instalirate, tako da u ovom koraku pažljivim i pravilnim izborom opcija čuvate resurse vašeg računara.



Slika 1.2. Instalacija razvojnog okruženja Visual Studio

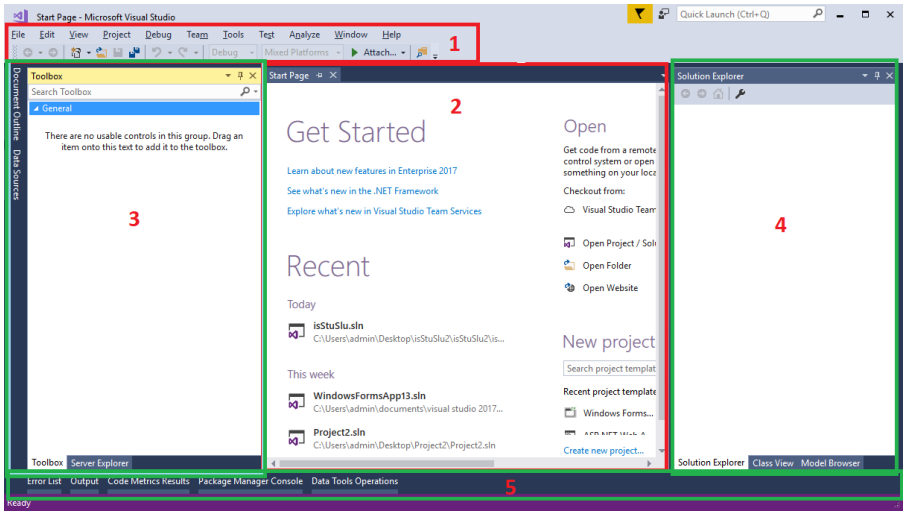
Razvojno okruženje

Nakon urađene instalacije pokrenite aplikaciju preko menija ili skraćenice.



Slika 1.3. Pokretanje Visual Studio IDE okruženja za razvoj aplikacija

Nakon pokretanja, otvara se aplikacija i dobijate radno okruženje pripremljeno za prve projekte, kao na slici:



Slika 1.4. Izgled razvojnog okruženja

Na slici se uočavaju osnovni regioni:

1. Navigacioni meni preko koga možete odabrati bilo koju opciju.
2. Centralni deo. Obično je namenjen uređivanju koda ili dizajniranju.

3. Levi pomoćni deo. Obično se koristi za prozore poput Toolbox odnosno Server Explorer prozora. Tu su pomoćne komponente za uređivanje aplikacija.
4. Desni pomoćni deo. Obično su tu prozori: *Solution Explorer, Class View, Model Browser.*
    a. *Solution Explorer* – prikazuje hijerarhijsku organizaciju fajlova u projektu koji se obrađuje.
    b. *Class View* – prikazuje organizaciju klasa.
5. Pomoćni prozori na dnu. Obično se koriste pri otkrivanju greška i evidentiranju poruka sistema.

Šabloni projekata

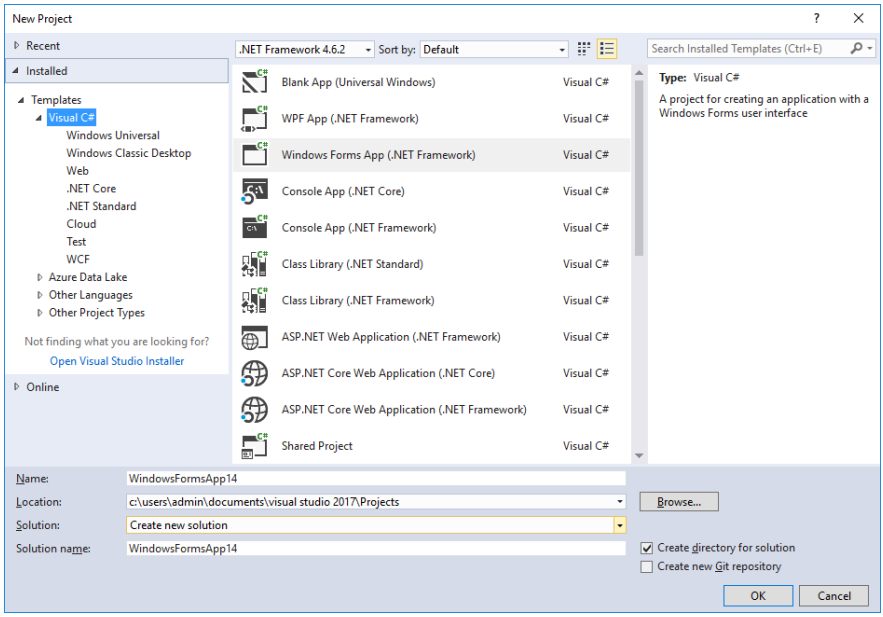
Projekat sadrži veći skup datoteka koji zajedno čine celinu odnosno njihovim prevođenjem i povezivanjem kreira se jedna aplikacija. Projekat je objedinjen zahvaljujući razvojnom okruženju i izmene na njemu treba da se obavljaju preko razvojnog okruženja VS.

Datoteke mogu sadržati kod, na primer C# kod, JavaScript kod, HTML stranice, XML dokumenta, slike, video fajlove i slično.

Pogledajmo postupak kreiranja najjednostavnijeg projekta.

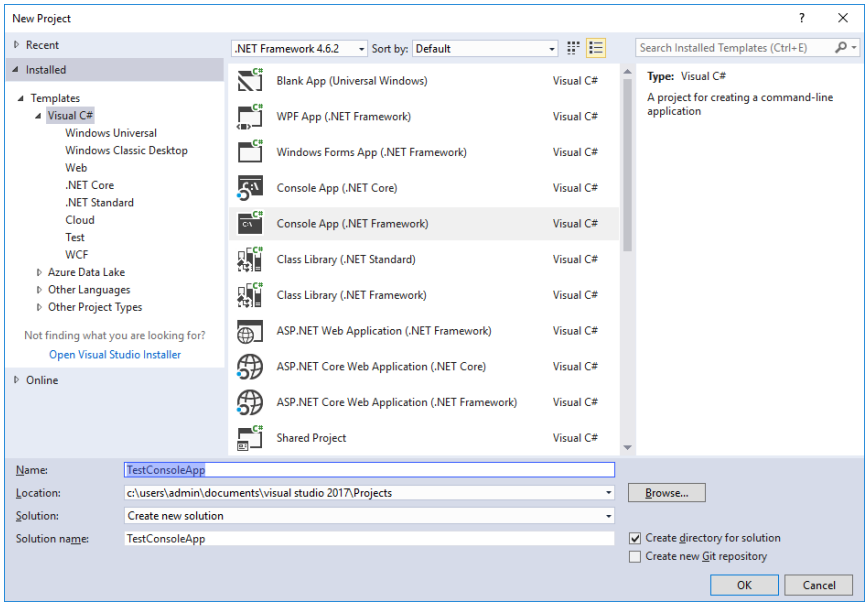
Korak 1. Bira se opcija File > New > Project

Korak 2. Zatim se dobija prozor „New Project“ kojim se inicijalizuje projekat za razvoj. Inicijalizacija znači početno uključivanje potrebnih biblioteka i osnovni dizajn sa kojim se kreće u razvoj. Sve što se u ovoj fazi definiše moguće je naknadno promeniti, ali je svakako jednostavnije napraviti pravilan izbor, pogotovo za početak.



Slika 1.5. Izbor novog projekta

U ovom kursu bavimo se veb projektima tako da ćemo se ovom tipu projekta potpuno posvetiti u nastavku kursa. Međutim, kao prva aplikacija u VS okruženju obavezno se koristi najjednostavniji tip projekta tj. kreiranje konzolne aplikacije. Konzolne aplikacije imaju praktičnu vrednost pri testiranju, administriranju i naravno za učenje osnova programskog jezika. Pošto se odabere tip projekta Console App (.NET Framework) popunjavaju se preostala polja.



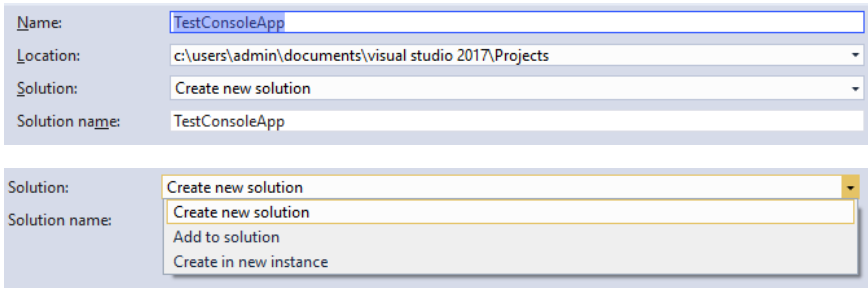
Slika 1.6. Izbor konzolne aplikacije

Location – predstavlja poziciju u File sistemu za smeštanje projekta.

Zatim se popunjava naziv projekta tj. polje Name.

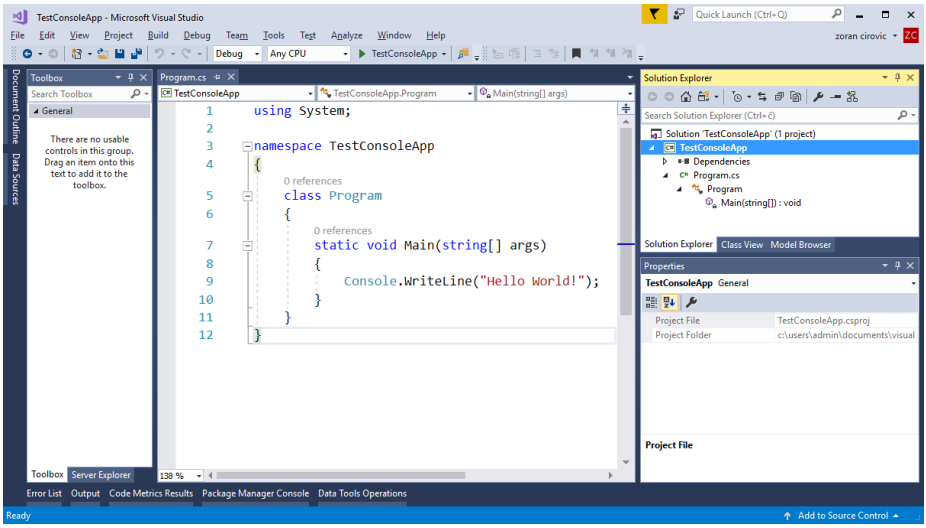
Automatski se popunjava i *Solution name*, tj to postaje podrazumevano ime za celo rešenje – eng. Solution. Rešenje sadrži bar jedan, a može i više projekata.

U polju Solution se bira opcija za kreiranje novog projekta ili se projekat dodaje nekom od drugih Solution-a.

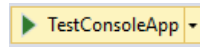


Slika 1.7. Definisanje određenog rešenja

Korak 2. Zatim se dobija kreirana aplikacija pripremljena za dalje promene.



Slika 1.8. VS nakon kreiranja aplikacije

Pritiskom na **F5** ili **Ctrl+F5** ili preko menija na ikonu 

se prevođenje fajlova, njihovo povezivanje u aplikaciju i izvršavanje.

F5 odnosno Ctrl+F5 predstavljaju dva načina pokretanja aplikacije: sa debagovanjem ili bez debagovanja. Debagovanje je termin koji je već duže vreme odomaćen u srpskom jeziku a koristi se da opiše za rad na otkrivanju grešaka.

Napomena: U toku ovog kursa koristi se C# programski jezik i IDE Visual Studio. Imajući u vidu sličnosti sa već pomenutim C-olikim jezicima nećemo se baviti osnovnom sintaksom jezika C#. Ukoliko želite da ispitate dodatno neke karakteristike jezika, referentne i vrednosne tipove i slično onda je rad na aplikacijama u konzoli dobar izbor.

Pitanja i zadaci za proveru znanja

1. Ispitati verziju .NET koju imate na računaru?
2. Instalirati Visual Studio. Koje ste komponente odabrali?
3. Šta je to šablon projekta i koje poznajete?
4. Šta je konzolna aplikacija?
5. Kakva je Windows Forms aplikacija?
6. Koje sve tipove Veb aplikacija možete kreirati preko VS IDE okruženja?
7. Kakva je REST arhitektura?
8. Koji su prozori u VS IDE okruženju i čemu služe?
9. Kako možete pogledati kreirane klase a kako fajlove?

2. Uvod u ADO.NET

ADO.NET je skup klasa za pristup i manipulaciju podacima koji su smešteni u izvorima podataka kao što su baze podataka. Obično se pod bazama podataka podrazumevaju relacione baze. Pristup podacima u bazama treba da bude: siguran, pouzdan, višekorisnički, pa se taj pristup ostvaruje preko servera baza. Primeri nekih servera baza su: SQL Server, MySql, Postgre, Microsoft Access, itd. ADO.NET se može koristiti kod svih pomenutih baza, tačnije kod svih za koje postoji odgovarajući drajver, ali se može koristiti i kod nerelacionih izvora podataka.

U ovom poglavlju biće objašnjeno:

Rad sa konekcijama odnosno upotreba objekta `Connection` sa svakim od tipova .NET dobavljača podataka. Kreiranje stringova veze. Definisanje korisnika i lozinki. Upoznavanje sa ostalim metodama i svojstvima.

Rad sa komandnim objektom `Command`. Kreiranje i izvršavanje komande kao i prihvatanje rezultata. Rad sa `DataReader` objektom.

Korišćenje `DataSet` objekta. Imenovani i neimenovani objekat DataSet.

Upotreba `DataAdapter` objekta i njegova primene zajedno sa `DataSet` objektom.

Instaliranje *Northwind* baze

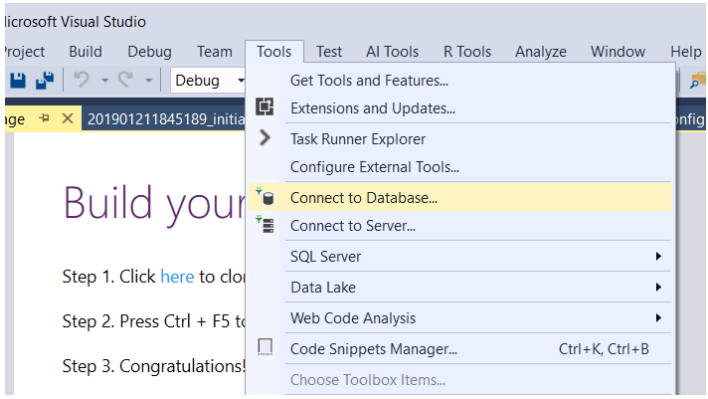
Ukoliko nemate već instaliranu neku bazu za rad potrebno je da instalirate neku koja vam stoji na raspolaganju. Bitno je da imate na umu da nije bitno koju bazu koristite, ali da svaki server ima svoje drajvere za pristup i da u ovom slučaju moramo koristiti .net drajvere. Za potrebe učenja u ovom kursu koristiće se Northwind baza. Za više informacija i preuzimanje baze pogledajte link: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/downloading-sample-databases

Pri radu nije neophodno da imate neku od verzija MS Sql Server-a. Naravno, to je poželjno i često predstavlja realan slučaj u praktičnoj primeni, ali je takođe čest slučaj i rad sa podacima pomoću lokalnog servera. Ukoliko se opredelite za instalaciju sopstvenog servera i programa za upravljanje podacima vodite računa da je `Express` verzija na raspolaganju bez plaćanja.

Konekcija

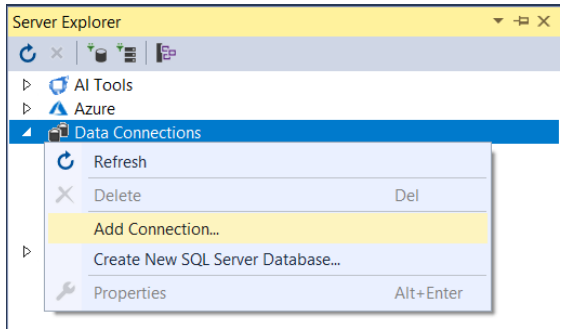
Veza do baze podataka u ADO.NET-u uspostavlja se preko komponente *Connection.* Konkretni objekat koji se koristi za konekciju zavisi od primenjenog posrednika koji se koristi za dobavljanje podataka. Međutim, važno je da zapamtite da sve klase za konekciju u ADO.NET-u implementiraju isti interfejs, a to znači da imaju zajednički skup metoda.

Dodavanje nove konekcije započinjete iz menija *Tools,* izborom opcije *Connect to Database...*



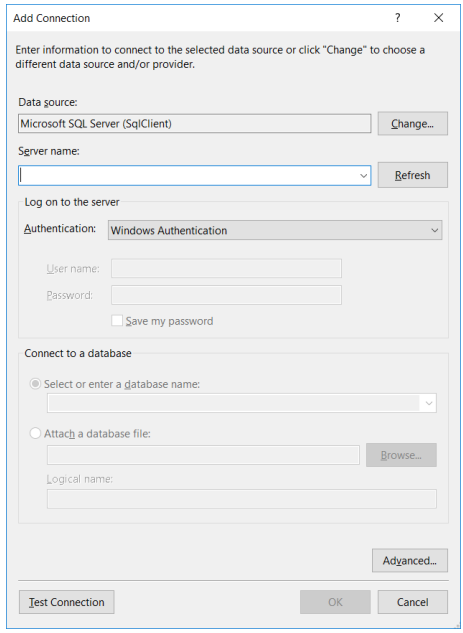
Slika 2.1. Otvaranje alatke za povezivanje sa bazom

Drugi način pristupa je preko pomoćne kartice - prozora, obično sa leve strane, koji se naziva *Server Explorer*. Koristeći ovaj pogled, možete, takođe, dodati novu konekciju.



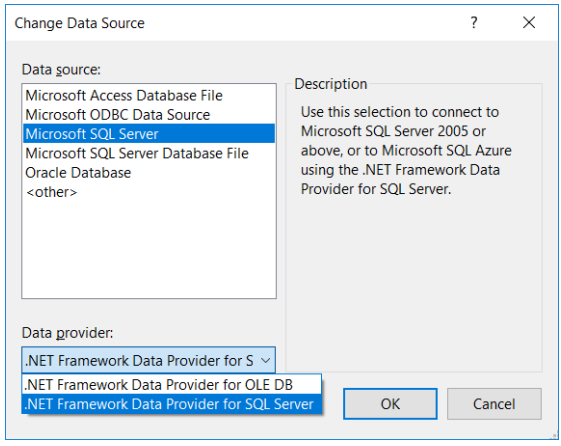
Slika 2.2. Dodavanje nove konekcije preko kartice Server Explorer

Otvara se dijalog *Data Link Properties*, kao na slici



Slika 2.3. Forma za dodavanje nove konekcije

Pogledajmo šta se podešava promenom *Data source* opcije, klikom na dugme *Change*.

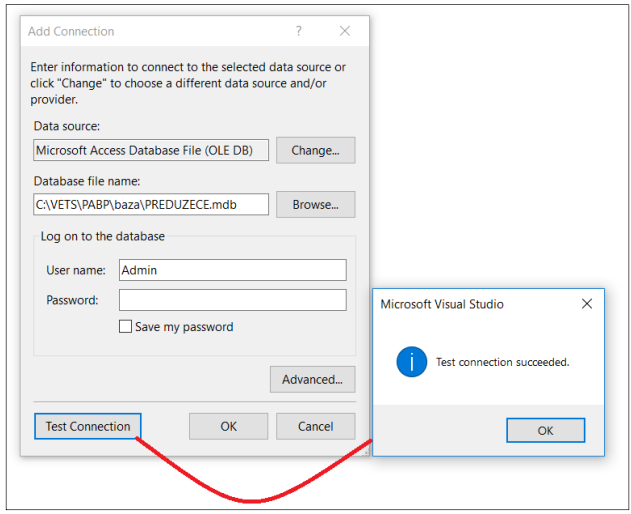


Slika 2.4. Izmena izvora podataka

Ovde je moguće podesiti izvor podataka, kao i vrstu dobavljača za taj izvor. Konkretno, u gornjem primeru se vidi da za MS SQL Server mogu

postojati dva dobavljača: *.NET Framework Data Provider for OLE DB* i *NET Framework Data Provider for SQL Server*.

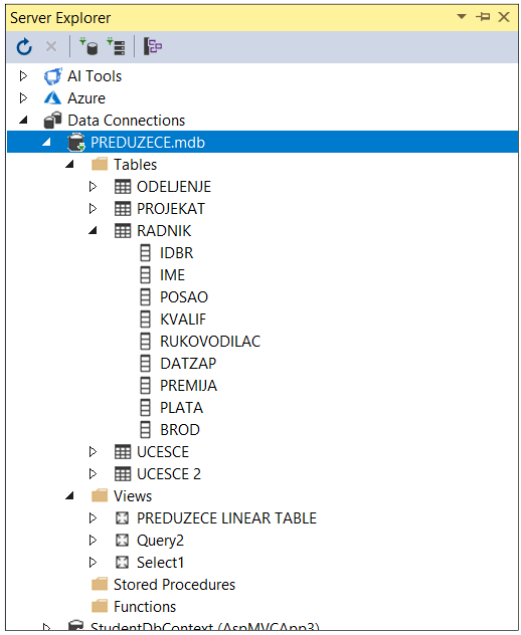
U sledećem primeru uspostavićemo konekciju sa bazom PREDUZECE.mdb. Baza je u ovom slučaju smeštena u jednom folderu na disku, a pristup bazi nije zaštićen:



Slika 2.5. Podešavanje konekcije do baze PREDUZECE.mdb

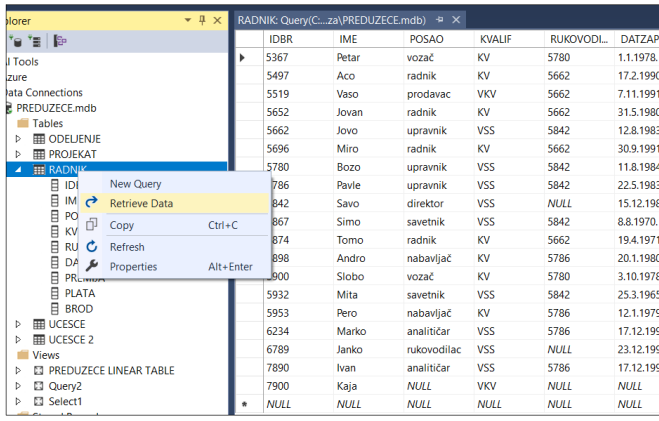
Konekciju obavezno testirajte, pritiskom na dugme Test Connection. Ukoliko test ne prođe, parametre konekcije niste postavili dobro i dalji rad sa podacima neće biti moguć. Odgovor mora da bude kao na slici gore.

Istovremeno, Server Explorer identifikuje ovu konekciju do baze. Pogledajte sliku ispod. Možete lako prići do podataka o tabelama i drugim objektima koji su kreirani u bazi.



Slika 2.6. Pogled na elemente baze

Takođe, koristeći *Server Explorer*, moguće je videti sadržaj u tabelama. Ako otvorite konteksni meni na tabeli RADNIK, a zatim izaberete stavku *Retrieve Data from Table*, dobićete sadržaj te tabele.



IDBR	IME	POSAO	KVALIF	RUKOVODI...	DATZAP
5367	Petar	vozač	KV	5780	1.1.1978. 0
5497	Aco	radnik	KV	5662	17.2.1990.
5519	Vaso	prodavac	VKV	5662	7.11.1991.
5652	Jovan	radnik	KV	5662	31.5.1980.
5662	Jovo	upravnik	VSS	5842	12.8.1983.
5696	Miro	radnik	KV	5662	30.9.1991.
5780	Bozo	upravnik	VSS	5842	11.8.1984.
786	Pavle	upravnik	VSS	5842	22.5.1983.
842	Savo	direktor	VSS	NULL	15.12.1981.
867	Simo	savetnik	VSS	5842	8.8.1970. 0
874	Tomo	radnik	KV	5662	19.4.1971.
898	Andro	nabavljač	KV	5786	20.1.1980.
900	Slobo	vozač	KV	5780	3.10.1978.
5932	Mita	savetnik	VSS	5842	25.3.1965.
5953	Pero	nabavljač	KV	5786	12.1.1979.
6234	Marko	analitičar	VSS	5786	17.12.1990.
6789	Janko	rukovodilac	VSS	NULL	23.12.1999.
7890	Ivan	analitičar	VSS	5786	17.12.1990.
7900	Kaja	NULL	VKV	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL

Slika 2.7. Dobijanje i pogled na sadržaj tabele

Dobavljači

Svaki dobavljač ima svoj prostor imena odnosno klase za rad sa bazama. Na sreću svi oni imaju zajedničke metode i svojstva koja nam omogućavaju da lako kreiramo aplikacije za različite baze i sa različitim klasam.

Tabela 2.1. Prikaz prostora imena i objekata različitih dobavljača

Objekat	Prostor imena
OleDbConnection	System.Data.OleDb
OdbcConnection	System.Data.Odbc
SqlConnection	System.Data.SqlClient

Kreiranje

Rad sa konekcijom se svodi na rad a odgovarajućim klasama. Sve klase koje rade sa konekcijom, bez obzira na konkretnog dobavljača, odnosno bez obzira na konkretno korišćenu klasu, konekcije imaju zajedničke metode. Pogledajmo kako to izgleda u dva slučaja.

MS Sql Server

Uvođenje prostora imena sa klasama od interesa:

```
using System.Data.SqlClient;
```

Kreiranje konekcijskog objekta:

```
SqlConnection conn = new SqlConnection();
conn.ConnectionString =
//ime računara ili IP adresa
"Server = imeRac;" +
// koristimo Windows prijavljivanje
```

```
"Integrated Security = SSPI;" +
// čekamo samo 5 sekundi
"Connection Timeout = 5;"
```

Access

Uvođenje prostora imena sa klasama od interesa:

```
using System.Data;
using System.Data.OleDb;
```

Kreiranje konekcijskog objekta:

```
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString =
//ime dobavljača za SQL je bio SQLOLEDB
"Provider = Microsoft.Jet.OLEDB.4.0;" +
//izvor tj. mdb datoteka
"Data Source = c:\\...\\northwind.mdb;"
```

Elementi konekcijskog stringa

Konekcijski string se sastoji od više elemenata tj. delova koji su razdvojeni tačka-zarezom. Objasnimo preciznije ove delove.

Integrated Security

- SSPI – eng. *Security Support Provider Interface.* Povezuje sigurnosni sistem SQL servera sa sigurnosnim sistemom Windows-a. Možemo koristiti *true* umesto SSPI.
- Ako koristimo korisničko ime i lozinku Izbaciti član "*Integrated Security ".* Dodati:

  User id=sa;

  Password=1x2yz3;

  "sa" je podrazumevani korisnik

Connection Timeout

Predstavlja maksimalno vreme čekanja do uspostavljanja veze tj. do ostvarivanja konekcije. Podrazumevana vrednost je 15 sekundi, ukoliko ne uradimo podešavanje ovog elementa.

Database

Ukazuje na bazu na koju se vezujemo kod SQL Servera.

Otvaranje/zatvaranje konekcije

Konekcije su SKUPE. Zauzimaju memorijski prostor na klijentskoj mašini ali i serveru. Previše otvorenih veza usporava rad a može da spreči otvaranje novih.

Nekada se broj veza posebno doplaćuje. Zato se otvaraju i drže otvorene samo kada je neophodno tj. u neposrednom radu sa serverom. Rad sa konekcija je uobičajen u blokovima za obradu izuzetaka. Obično se dodaje i blok `finally` u kom se vrši zatvaranje konekcije. Pogledajte opšti primer:

```
try
{
    . . .
    conn.Open();
    . . .
    conn.Close();
    . . .

}
catch
{
   // obrada izuzetka ex
}
```

Objekti *Command* i *DataReader*

Ovi objekti pripadaju skupu objekata specifičnih za dobavljača - *.Net Data Provider*, kao što su i objekti: `DataAdapter` i `Connection`. Objekat `Command` čuva i/ili izvršava SQL upit kojim se podaci preuzimaju iz baze.

Komanda omogućava izvršavanje SQL naredba ili uskladištenih procedura (eng. *Store procedure*) koristeći objekat `Connection`. Komanda može kao rezultat imati vraćeni niz zapisa ili pak samo izazvati neke promene u bazi.

Kreiranje

Kreiranje se obavlja na dva načina:

1. Samostalno:

```
SqlCommand cmd = new SqlCommand();
```

2. Koristeći `Connection` objekat

```
SqlCommand cmd = conn.CreateCommand();
```

U prvom slučaju se mora obezbediti naknadno povezivanje sa nekim konekcijskim objektom. Zašto? Jer komanda mora da se izvršava preko neke (otvorene) konekcije.

```
cmd.Connection =conn;
```

Postoji više konstruktora za ovaj objekat. Evo primera nekoliko njih:

```
new xxCommand(),
new xxCommand(string komanda),
new xxCommand(string komanda, xxConnection kon),
new xxCommand(string komanda, xxConnection kon, Trans).
```

Dodeljivanje komandnog teksta tj. sql naredbe:

```
cmd.CommandText = sqlNaredba;
```

treba da bude u `try-catch` bloku i omogućeno je samo valjanim postavljanjem konekcijskog objekta.

Svojstva

Svojstva komandnog objekta detaljnije definišu njegove karakteristike. Sledi opis svojstava objekta `Command`.

- `CommandText`, je znakovni niz (string) koji sadrži ili stvarni tekst komande koja treba da se izvrši na konekciji ili naziv uskladištene procedure iz izvora podataka.
- `CommandTimeout` određuje vreme koje će komanda čekati na odgovor od servera pre nego sto generiše grešku. Uzmite u obzir da je ovo vreme koje protekne pre nego što objekat `Command` počne da prima rezultate, a ne vreme koje je potrebno komandi da se izvrši. Izvoru podataka može biti potrebno 10 ili 15 minuta da vrati sve redove neke ogromne tabele, ali pod uslovom da je prvi red vraćen u toku zadatog perioda `CommandTimeout` neće generisati nikakvu grešku.
- `CommandType` definiše način na koji komanda treba da protumači sadržaj svojstva `CommandText`.
    - Vrednost `TableDirect` podržana je za objekat `OleDbCommand`, ali ne i za objekat `SqlCommand`, a ekvivalentna je naredbi SELECT * FROM <ime_tabele>, pri čemu je <ime_tabele> ime tabele specifikovano u svojstvu `CommandText`
- Svojstvo `Parameters` objekta `Command` sadrži kolekciju parametara za SQL naredbu ili uskladištenu proceduru navedenu u svojstvu `CommandText`. Ovu kolekciju ćemo detaljno ispitati u nastavku.
- Svojstvo `Transaction` sadrži referencu na objekat `Transaction` i služi pri obradi transakcija.

Izvršavanje

Postoji nekoliko metoda u okviru `Command` objekata preko kojih me moguće izvršavanje nekoliko različitih tipova komandi. To su:

1. Iskaz nije upit – `ExecuteNonQuery`

2. Jedna vrednost – `ExecuteScalar`
3. Jedan ili više redova – `ExecuteReader`
4. XML – `ExecuteXMLReader`

Slede primeri:

ExecuteNonQuery

```
cmd.CommandText = "insert into Employees
(FirstName,LastName) values ('Perica','P')";
cmd.Connection.Open();
int cnt = (int)cmd.ExecuteNonQuery();
```

ExecuteScalar

```
cmd.CommandText = "select count(*) from Employees";
cmd.Connection.Open();
int cnt = (int)cmd.ExecuteScalar();
```

ExecuteReader

Izvršavanje SQL upita koji treba da vrati više redova je nešto složenije od prethodnih primera. Upit može da znači vraćanje neprimereno velikog skupa podataka, pa se preuzimanje rezultata upita tj. pojedinih slogova vrši preko posebnog objekta.

`DataReader` je objekat koji se koristi kada komanda vraća skup zapisa – **select** tip SQL komande. Važna karakteristika čitanja redova ovog objekta je da tok podataka koji vraća `DataReader` je takav da se može **samo čitati** i kretati **samo unapred.** Naravno, razlog je brzina i efikasnost ovakvog pristupa. `DataReader` se ne kreira osim što se prihvata kao rezultat `ExecuteReader` metode.

Zapamtite da se u memoriji istovremeno nalazi samo jedan red podataka. Na primer:

```
cmd.CommandText = "select EmployeeID, FirstName, LastName
from Employees where LastName like '" + textBox1.Text +
"%'";
cmd.Connection.Open();
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
    string prezime = (string)reader["LastName"];
    int id = (int)reader["EmployeeID"];
    string ime = (string)reader["FirstName"];
    listBox1.Items.Add("" + id + "; " + ime + " " +
prezime);
}
```

Parametri

Parametri u komandi su jedna vrsta liste koja se definiše u .NET. Veličina liste nije ograničena, samo memorijom na računaru. Podrška različitih metoda za rad sa kolekcijom važi za sve kolekcije pa i za listu parametara.

Parametri komande se koriste primenjujući sledeći postupak:

1. Specificirati parametre kroz SQL komandu u upitima ili uskladištenim procedurama
2. Definisati odgovarajuće parametre u kolekciji Parameters
3. Dodeliti vrednosti

Na primer

```
cmd.CommandText = "select EmployeeID, FirstName, LastName
from Employees where LastName like @prm1";
SqlParameter p1 = new SqlParameter();
// vr. Se preuzima iz textBox-a
p1.Value = textBox1.Text + "%";
p1.ParameterName = "@prm1";
cmd.Parameters.Add(p1);
```

Upotreba parametara je višestruko korisna. Prvo, parametri obezbeđuju sigurnosni mehanizam formiranja upita, pa se njihovom primenom izbegava bezbednosni rizik poznat kao "*sql injection*". Druga prednost je jasno definisanje fiksnog dela komande kao i delova komande koji su promenljivi, odnosno preglednost je važna prednost primene parametara.

Objekat *DataSet*

Ovo je objekat koji je zadužen za čuvanje odnosno rad sa podacima (obično iz neke baze) za potrebe same aplikacije. `DataSet` ne uspostavlja vezu sa izvorom podataka, već samo čuva memorijsku sliku podataka u obliku kolona i redova tj. Ima mogućnost čuvanja tabelarne organizacije podataka. Ovo je jedan od najbolje osmišljenih i najviše korišćenih objekata, ali i je i jedna od važnijih karakteristika Microsoft-ove .NET platforme. Neke važne karakteristike su:

- Može da čuva rezultate više različitih SQL upita.
- Možete koristiti ovaj objekat nezavisno od konekcije.
- Može se kreirati XML dokument (kao i odgovarajuću XSD šemu) iz `DataSet` objekta .
- Možete kreirati objekat na osnovu odgovarajuće XML šeme dokumenta.

Klasa `DataSet` predviđena je da obuhvati više kolekcija .NET tipova:

Kolekcija **tabela**. A svaka tabela od:

kolekcija **redova**,

kolekcija **kolona**,

ograničenja.

Kolekcije **veze**

Metode

Osnovne metode *DataSet* objekta obezbeđuju manipulaciju sa podacima i njegovu primenu u aplikacijama. Evo nekoliko često korišćenih:

- `Clear` – briše sve tabele.
- `Clone` – kopira strukturu `DataSet`-a.
- `Copy` – kopira i strukturu i podatke.
- `HasChanges` – da li u objektu `DataSet` postoje izmene koje čekaju.

Vrste *DataSet*-a

U osnovi razlikujemo dve vrste:

- Tipizirani i
- Netipizirani.

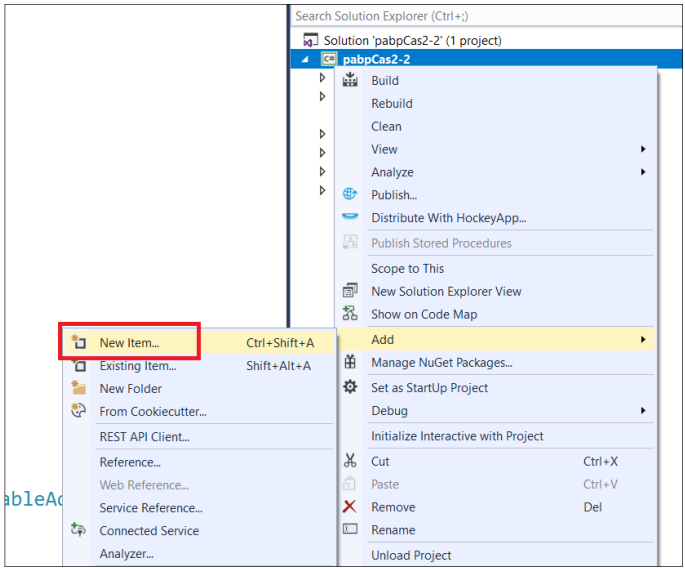
Netipizirani objekti su instance `DataSet` klase. U okviru ovog objekta se koriste kolekcije `DataTable`, `DataColumn` i `DataRow`. Ovi objekti opet sadrže kolekcije koje vraćaju standardne objekte. Na primer, kolekcija `Tables` objekta `DataSet` uvek prikazuje objekte tipa `DataTable` nezavisno od toga koja se tabela koristi. Pogledajmo primer primene ovih objekata.

```
DataSet ds = new DataSet();
DataTable dt = new DataTable("voce");
dt.Columns.Add("id", typeof(int));
dt.Columns.Add("naziv", typeof(string));
ds.Tables.Add(dt);
ds.Tables["voce"].Rows.Add(1, "jabuka");
ds.Tables["voce"].Rows.Add(2, "kruska");
```

```
ds.Tables["voce"].Rows.Add(3, "sljiva");
```

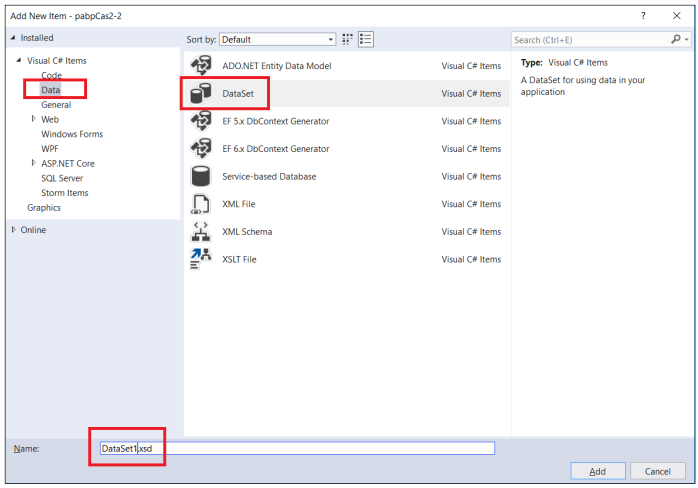
Visual Studio omogućava kreiranje. tipiziranih **DataSet-ova**. Tipizirani su kolekcije klasa koje odgovaraju klasama izvedenim od **DataSet**, **DataTable**, **DataRow**,… , ali sadrže specifična svojstva i metode dobijene na osnovu unapred definisane strukture podataka kroz XSD šemu **DataSet** objekta. Pogledajmo postupak dodavanja imenovanog **DataSet** objekta koji će odgovarati strukturi Northwind baze.

Korak 1. Dodavanje nove stavke projektu.



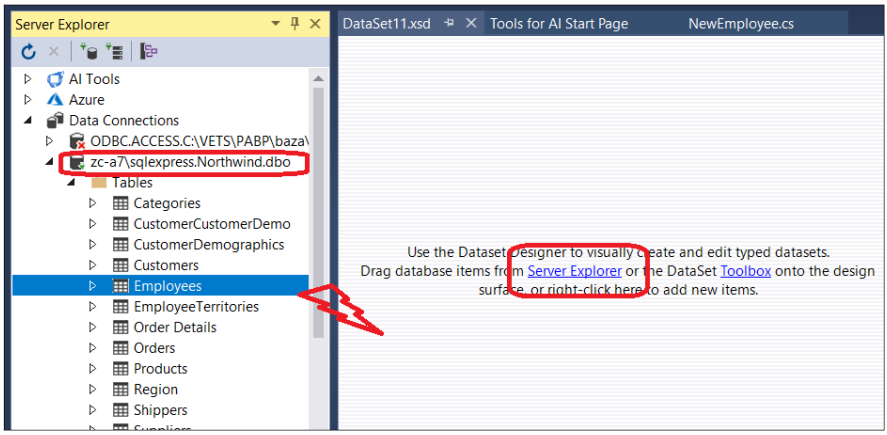
Slika 2.8. Dodavanje nove stavke projektu

Korak 2. Izbor vrste stavke.



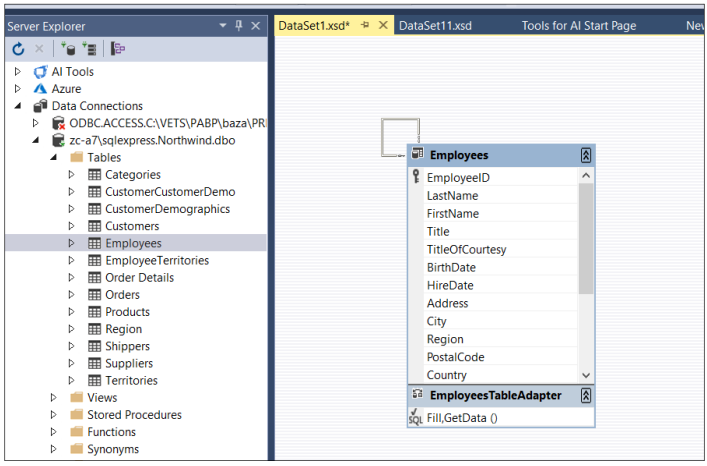
Slika 2.9.Izbor nove stavke

Korak 3. Formiranje šeme za **`DataSet`**. Prevlačenjem tabela iz prozora *Server Explorer* formirate tipizirane tabele u **`DataSet`** objektu



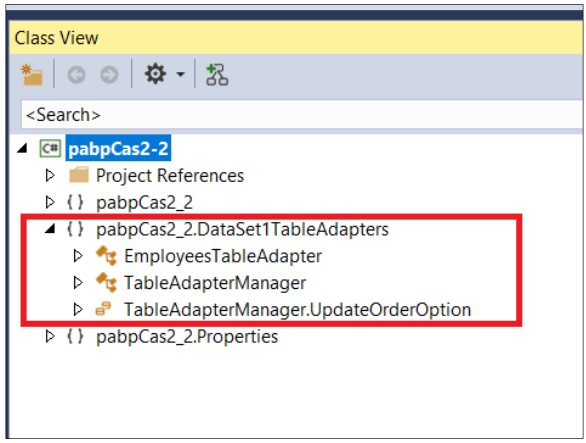
Slika 2.10.Formiranje šeme DataSet objekta

Konačan izgled kreiranog DataSet1 objekta:



Slika 2.11.Konačna izgled formirane šeme

Pogledajmo šta se dogodilo. U prozoru *Class View* pronađite kreirani prostor imena i klasa koji su zaduženi za rad sa podacima. Pogled na kreirane klase tipiziranog **`DataSet1`** objekta.



Slika 2.12.Pogled na kreirane klase

Prikažimo sada rad sa kolekcijama ovog objekta, a zatim i drugi objekat **`DataAdapter`**. Razlog je što ćemo upotrebu **`DataSet`** objekta povezati sa upotrebom **`DataAdapter`**-a.

Pristup podacima

Kao što smo rekli, svaka tabela se sastoji od kolekcije redova, `Rows`. Pojedinačnoj vrsti pristupamo preko indeksa. Na primer `Rows[11]` daje objekat `Row` iz kolekcije redova u odgovarajućoj tabeli, na poziciji 11, to je 12-ti po redu pošto indeksiranje startuje od 0.

Pristup podatku u određenoj koloni vrši se preko imena ili pozicije kolone.

Na primer, ako želimo da prikažemo sadržaj u dataSet1 objektu, tabele ODELJENJE, u vrsti 2, a koloni IME, kod je sledeći:

```
MessageBox.Show(
dataSet1.Tables["ODELJENJE"].Rows[2]["IME"].ToString() );
```

Objekat *DataAdapter*

Ovaj objekat se lako primenjuje za dobijanje podataka iz baze, ili za ažuriranje podataka u bazi. `DataAdapter` koristi konekcije za pristup podacima. Kreiranje `DataAdapter`-a implicitno prouzrokuje i kreiranje `Connection` objekta.

Obezbeđuje lak način za rad sa podacima. Ovaj objekat objedinjuje četiri komande: **Insert**, **Delete**, **Update** i **Select**. Preko ovih metoda `DataAdapter` vrši ažuriranje podataka u bazi na intuitivan način. Sadrži: objekat konekcije i četiri vrste komande.

Kreiranje

```
string sql = "select* from Customers";
SqlConnection sqlconn = new SqlConnection(...);
SqlDataAdapter da = new SqlDataAdapter(...,..);
//Kuda sa podacima?
//Kreiranje DataSet objekta za prihvat podataka
DataSet ds = new DataSet();
da.Fill(ds);
```

Svojstva

Osim četiri komande, ovaj objekat poseduje još neka svojstva.

- `TableMappings` – kolekcija koja obezbeđuje relaciju između kolona iz objekta `DataSet` i izvora podataka. (Kako `Fill` ume da prebaci podatke iz baze u DS).
- `AcceptChangesDuringFill` – određuje da li se na objektu `DataRow` poziva `AcceptChanges` kada se doda u objekat Data Table.
- `MissingMappingAction` – definiše akciju koja će se dogoditi kada se ne mogu upariti podaci sa nekom postojećom kolonom ili tabelom.
- `MissingSchemaAction` - slično ali za šemu.
- U slučaju greške moguće akcije za `MissingMappingAction` su:
  - `Error`
  - `Ignore`
  - `Passthrough` – kolona ili tabela koja se ne pronađe dodaje se u `DataSet` korišćenjem njenog imena u izvoru podataka
- `TableMappings` kolekcija - Relacije između izvora podataka sa jedne strane i skladišta podataka (`DataSet`-a) sa druge strane. Vezuje se ime kolone u tabeli u bazi sa imenom kolone u tabeli u `DataSet`-u

Metode

`Fill` - Popunjava `DataSet` podacima. Moguće je uraditi prethodno brisanje postojećih ili ne. To se podešava u određenim svojstvima – pronađite kako.

`Update` - Promene koje su učinjene na podacima se prebacuju u bazu. Pogledajte kako glasi sql upit komande Update. Zahvaljujući tome što

poseduje 4 `Command` objekta `DataAdapter` može da uradi sve 4 operacije i tako omogući ažuriranje tabele u bazi podataka nalik ažuriranju lokalne tabele.

Događaji

Osim događaja greške, koji su veoma bitni u ispravnom radu sa podacima, često se koriste sledeći događaji:

- `OnRowUpdating` – Izvršava se kada metod Update postavi vrednosti parametara komande koja treba da se izvrši, ali pre samog izvršavanja.
- `OnRowUpdated` – Nakon ažuriranja nekog sloga.

  Kada metod `Update` - izvrši odgovarajuću komandu na izvoru podataka. Svojstva argumenta koji ide uz ovoj događaj pogledaćemo na primeru:

  `Command` - komanda za podatke koja treba da se izvrši.

  `Errors` – greške koje .NET generiše.

  `Row` – objekat DataReader koji treba da se ažurira.

  `StatementType` – *Select,Insert,Update,Delete.*

  `TableMapping` – Objekat `DataTableMapping` koji se koristi za ažuriranje.

Ažuriranje podataka u bazi

Metod `Update`) objekta `DataAdapter` proverava sve redove objekta `DataTable` u okviru `DataSet`-a, i ako je potrebno, ažurira te podatke u tabeli. Svaki objekat `DataRow` ima svoje stanje u kome se prati da li je red izbrisan, dodat ili promenjen.

Tako na primer, metod `Delete` kojim se briše neki red, u stvari samo označava taj red u `DataSet`-u obrisanim. Stvarno brisanje u bazi se

obavlja pozivom metode `Update()`. Isto važi i za izmenu ili dodavanje podataka u nekom redu.

Na primer, ako hoćemo da dodamo novi red, potrebno je prvo dodati novi red u odgovarajuću tabelu objekta `DataSet`, a zatim izvesti `Update()` ovog skupa podataka, što rezultuje promenom u bazi. Ako izostavite ovu komandu, promena će biti vidljiva, ali bez stvarnih izmena u bazi.

```
DataRow newrow = dataSet1.Tables["ODELJENJE"].NewRow();
newrow["BROD"] = 22;
newrow["IMEOD"] = "NOVO";
newrow["MESTO"] = "NNN";
this.dataSet1.Tables["ODELJENJE"].Rows.Add(newrow);

this.oleDbDataAdapter2.Update(this.dataSet1, "ODELJENJE");
```

Brisanje jednog reda izvodite selekcijom tog reda i pozivom metode `Delete`. Ne zaboravite `Update`, inače promene neće biti upisane u bazu. Na primer:

```
DataRow dr = dataSet1.Tables["ODELJENJE"].Rows.Find("22");
dr.Delete();
this.oleDbDataAdapter2.Update(this.dataSet1, "ODELJENJE");
```

Na kraju, da bi samo promenili podatke u bazi, dovoljno je da objekte za unos podataka ostavimo, tako da je unos moguć (`ReadOnly = false`). Promena u kontroli menja `DataSet` objekat koji je vezan za kontrolu. Dakle, kada promenite vrednost nekog reda, dovoljno je još samo pozvati metod `Update`.

Pitanja i zadaci za proveru znanja

1. Koja je uloga objekta Connection?
2. Šta su to dobavljači podataka i koje znate dobavljače?

3. Da li jedna aplikacija može da koristi više dobavljača za rad sa jednom bazom?
4. Da li jedna aplikacija može da ažurira više baza podataka?
5. Napravite konekciju do Northwind baze i izdvojite konekcijski string.
6. Čemu služi objekat *Command*?
7. Koje vrste komandi odnosno metoda za izvršenje komandi poseduje Command objekat?
8. Napišite kod za preuzimanje imena tabele Employees koristeći Command i DataReader.
9. Proširite prethodno pitanje sa uslovom da ime počinje nekim zadatim stringom.
10. Ako već niste uradili primenom parametara prethodni zadatak, onda primenite parametre.
11. Objasnite objekat DataSet.
12. Objasnite objekat DataAdapter i pokažite njegovu primenu uz upotrebu DataSet objekta.

3. LINQ

Cilj ovog poglavlja je da čitaocu pruži potrebna znanja u primene novog jezika za upite zasnovanog na kolekcijama podataka – LINQ. Na osnovu ovih znanja kasnije ćemo moći da kreiramo efikasne upite u kodu vezane neposredno za bazu podataka ili kolekcije u okviru objekta `DataSet`, liste različitih vrsta i slično. Poglavlje obrađuje lambda izraze, anonimne tipove, ali i standardne operatore: filtriranje, sortiranje, grupisanje i slično.

Uvod

Linq (eng. Language-Integrated Query) je naziv za skup tehnologija zasnovanih na integraciji upitnih svojstava direktno u jeziku C#. Tradicionalni upiti prema podacima predstavljaju stringove koji se ne mogu proveriti u toku kompajliranja ili preko ugrađene inteligentne podrške u IDE (eng. IntelliSense). U slučaju različitih upitnih jezika programer treba da zna različite tipove upita: SQL baze podataka, NoSql baze, XML dokumenta, različite veb servise itd. Primenom Linq upita koristi se jedinstven upit zasnovan na klasama, metodama i događajima.

Linq je sastavni deo rada sa podacima preko *Entity Framework* okruženja i sada ćemo ga detaljnije obraditi kako bi rad sa podacima na nivou upita bio jednostavniji, brži i dosledan konceptu ASP.NET MVC okruženju.

Osnovna sintaksa upita

Pogledajmo jedan primer jednostavnog Linq izraza:

```
var query = from e in employees
            where e.id == 1
            select e.name;
```

Na prvi pogled sintaksa značajno odstupa od standardne C sintakse koju koristi C#. Ipak, ovo **nije pseudokod**; ovo je LINQ sintaksa i slična je SQL sintaksi.

LINQ poseduje bogatu kolekciju naredbi za implementaciju kompleksnih upita koje sadrže agregatne funkcije, združivanje, grupisanje, sortiranje i još puno toga.

Funkcionalnost

Linq poseduje jedinstven model koji se može koristiti u više različitih namena.

- ***LINQ to Objects*** je aplikacioni interfejs koji prikazuje standardne operatore upita) za dobijanje podataka iz nekog objekta čija klasa implementira interfejs **IEnumerable<T>.** Ovi upiti se izvršavaju na podacima u memoriji.
- ***LINQ to ADO.NET*** proširuje standardne operatore na rad sa relacionim podacima. Sastoji se iz 3 dela:
  - **LINQ to SQL** se koristi za upite ka relacionim bazama kao MSSQL.
  - **LINQ to DataSet** se koristi za upite koji koriste ADO.NET skupove podataka.

- **LINQ to Entities** je Microsoft ORM (eng. Object-Relational Mapping) rešenje za upotrebu Entities objekata.
- *LINQ to XML*

LINQ to Objects

Najopštiji prikaz rada Linq jezika prikazaćemo primenom na objektima, tačnije na kolekcijama objekata.

Za ove primere možete koristiti postojeću ASP aplikaciju ili kreirati potpuno novu, na primer konzolnog tipa.

Najpre dodajmo novu klasu `Person`:

```
class Person
{
    public int ID;
    public int IDRole;
    public string LastName;
    public string FirstName;
}
```

Zatim napravimo listu ovih objekata i ujedno uradimo inicijalizaciju objekata u ovoj listi:

```
List<Person> people = new List<Person>
{
    new Person()
    {
        ID = 1, IDRole = 1,
        LastName = "Petar", FirstName = "Ilić"
    },
    new Person() {
        ID = 2, IDRole = 2,
        LastName = "Jovan", FirstName = "Jović"
    },
    new Person() {
        ID = 3, IDRole = 1,
```

```
        LastName = "Ana", FirstName = "Milić"
    }
};
```

Zatim u nekoj metodi napišimo prvi konkretan upit. Neka to bude:

```
var query = from p in people
            where p.ID == 1
            select p;
```

Uočavaju se ključne reči iz C#: `var`, `in`, kao i operatori.

Takođe se prepoznaju ključne reči za upite: `from`, `where`, `select`.

Rezultat upita je kolekcija tipa (tačnije interfejsa) `IEnumerable`. Ako su objekti koji se selektuju unapred definisane klase, kao u našem primeru onda je rezultat generički tip `IEnumerable<Tip>`.

Bez obzira što u izrazu koristimo tip var za rezultat upita, ovaj tip ne ograničava prihvatanje stvarnog tipa od upita. Postavlja se pitanje kako koristiti ove podatke? Postoji više načina, a ovde ću pokazati samo dva.

- IEnumerable interfejs omogućava prolaz kroz sve elemente kolekcije, pa bi mogući prihvat rezultata mogao da bude:
  ```
  List<Person> lst = new List<Person>();
  if (query != null)
  {
      foreach (object item in query)
      {
          lst.Add((Person)item);
      }
  }
  ```
- Direktnom konverzijom u željenu kolekciju, na primer u listu: `lst = query.ToList<Person>();`

Metode proširivanja

Linq sintaksa koristi ključne reči, na primer: `where` i `select`, koji se transformišu u C# metode kao što se ceo Linq izraz transformiše u sekvencu metoda koje se pozivaju na nekoj kolekciji objekata. Ove ključne reči se transformišu u dve metode: `Where<T>()` i `Select<T>().`

Metode se nadovezuju, tako da se rezultati `Where` metode mogu dalje izdvajati pozivom `Where` metode ili pomoću `Select` metode. Tip podataka je `IEnumerable<T>`.

Ove metode su vezane za tip podataka na koji se odnose, odnosno vrše proširenje funkcionalnosti postojećih klasa, primenom metoda proširivanja (eng. *extension methods*). Takve metode su još: `Join`, `OrderBy`, `GroupBy`, …

Prema tome, standardni Linq izraz:

```
var query = from p in people
            where p.ID == 1
            select p;
```

može da se napiše drugačije, primenom metoda proširenja.

```
var queryCopy = people.Where(x => x.ID == 1).Select(x=>x);
```

Argumenti ovih metoda proširenja, na primer `x => x.ID == 1`, predstavljaju specifične izraze tzv. **Lambda izraze**. U standardnom Linq izrazu koristi se skraćena verzija Lambda izraza, `p.ID == 1`.

Lambda izrazi

Metoda `Where` se primenjuje na kolekciji objekata. Metoda vrši odabir određenih objekata na osnovu neke proizvoljne funkcije koja treba da se prosledi ovoj metodi. Ta funkcija obavlja filtriranje tako što vraća `true` za objekte koji zadovoljava uslov filtriranja, odnosno `false` za one koji to ne zadovoljavaju.

Funkcija, tj druga metoda, koja se prosleđuje kao argument nekoj metodi mora biti tipa `delegate`. Dakle, ono što `Where` metoda prima je zapravo objekat tipa `delegate`.

Druga osobina primene funkcija za filtriranje koje se prosleđuju Where metodi jeste da su to obično kratke metode i koje se koriste samo na tom mestu. Zbog toga ta funkcija se piše neposredno kada se i koristi na način kako se pišu funkcije bez imena koje se odmah i koriste. Ovo su tzv anonimne funkcije.

Dakle, `x => x.ID == 1,` predstavlja lambda izraz koji opisuje generisanje delegata za funkciju filtriranja. Izraz vraća tip `bool` u zavisnosti da li je argument `x.ID == 1.` Ova funkcija se izvršava na celoj kolekciji na koju se metoda `Where` odnosi.

Primer drugačije upotrebe ovih izraza dat je u sledećem kodu:

```
delegate int del(int i);
static void Main(string[] args)
{
       del myDelegate = x => x * x;
       int j = myDelegate(5);  //j = 25
}
```

Anonimni tipovi

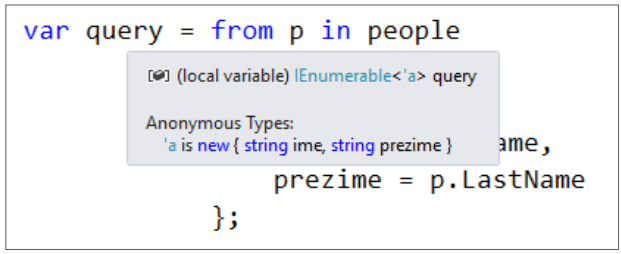
Anonimni tipovi, slično anonimnim funkcijama, znače tipove objekata koji nisu prethodno definisani. Na primer:

```
var v = new { ime = "Jovan", prezime = "Jović" };
```

U Linq izrazima anonimni tipovi dozvoljavaju da se radi sa rezultatima upita bez eksplicitne definicije klase koja ih predstavlja.

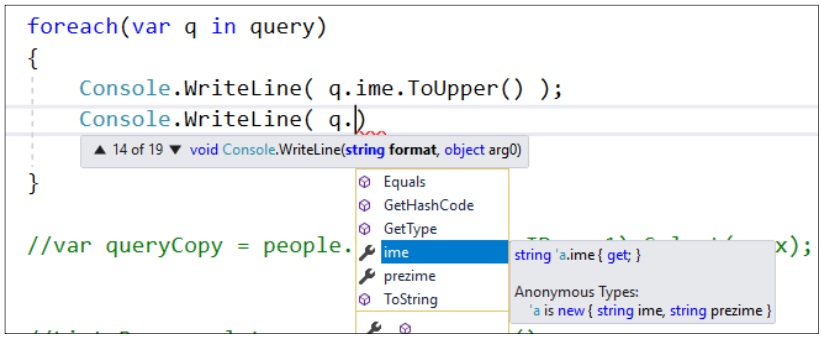
```
var query = from p in people
            where p.ID == 1
            select new {
                ime = p.FirstName,
                prezime = p.LastName
            };
```

Pogledajmo koji tip pokazuje IDE tokom formiranja Linq izraza:



Slika 3.1. Rad sa anonimnim tipovima u Linq izrazima

Nakon kreiranja Linq izraza, rad pomoću *IntellySense* alatke omogućava laku upotrebu sa prepoznavanjem osobina rezultujućeg tipa u izrazu. Pogledajmo primer kada je rezultat Linq poznati tip iako je promenljiva deklarisana kao var.



Slika 3.2. Rad sa formiranim objektom

Standardni operatori

Filtriranje

Operatori za filtriranje su:

- **`Where`**
- **`OfType`**

Primer:

```
System.Collections.ArrayList list = new
System.Collections.ArrayList();
list.Add("Petar");
list.Add("Jova");
list.Add(new object());
list.Add(32.22);
list.Add(new object());
var query = from x in list.OfType<string>() where x ==
"Jova" select x;
```

`OfType` operator se može koristiti u slučaju ne-generičkih kolekcija, kao na primer `ArrayList`.

`Napomena.` Pošto `ArrayList` ne implementira `IEnumerable<T>`, `OfType` operator je jedini Linq operator koji možemo primeniti na

listu. `OfType` je takođe koristan ako radite više nasleđenih klasa i ako želite da selektujete samo objekte određenog tipa.

Sortiranje

Operatori za sortiranje su:

- **`OrderBy, OrderByDescending,`**
- **`ThenBy, ThenByDescending,`**
- **`Reverse.`**

Primer:

```
var q = from p in people orderby p.LastName,
p.FirstName select p;
```

Povratna vrednost `OrderBy` operatora je `IOrderedEnumerable<T>`. Ovaj specijalni interfejs je nasleđen od `IEnumreable<T>` i dozvoljava primenu operatora `ThenBy` , `ThenByDescending`.

Skupovni operatori

- **`Distinct`** - za izbacivanje dupliranih vrednosti,
- **`Except`** - vraća razliku dve sekvence,
- **`Intersect`** - vraća presek dve sekvence,
- **`Union`** - vraća uniju elemenata dve sekvence.

Primer:

```
int[] niz2 = { 2, 4, 6, 8, 10 };
int[] niz3 = { 3, 6, 9, 12, 15 };
//6
var intersection = niz2.Intersect(niz3);
// 2, 4, 8, 10
var except = niz2.Except(niz3);
```

```
// 2, 4, 6, 8, 10, 3, 9, 12, 15
var union = niz2.Union(niz3);
```

Kvantifikatori

- **All,**
- **Any,**
- **Contains**

Primeri:

```
int[] niz2 = { 2, 4, 6, 8, 10 };
// true
bool areAllevenNumbers = niz2.All(i => i % 2 == 0);
// true
bool containsMultipleOfThree = niz2.Any(i => i % 3 == 0);
// false
bool hasSeven = niz2.Contains(7);
```

Projekcije

Projekcije su operatori koji vraćaju rezultat upita. Postoje dva takva operatora:

- Select

  Vraća 1 izlaz za 1 ulaz. Može da formira i novi tip podataka.

- SelectMany

  Koristi se kada radimo sa sekvencom od sekvenci. SelectMany se koristi kada postoji višestruki from.

Primeri

```
string[] tekst = new string[]{
"Ovo je samo primer",
"Ovo je druga recenica",
"Treci primer" };
```

```
var query = (from recenice in tekst from reci in
recenice.Split(' ')
            select reci).Distinct();
```

Iz početnog objekta **tekst** koji je niz nekoliko stringova, najpre se izdvajaju rečenice:

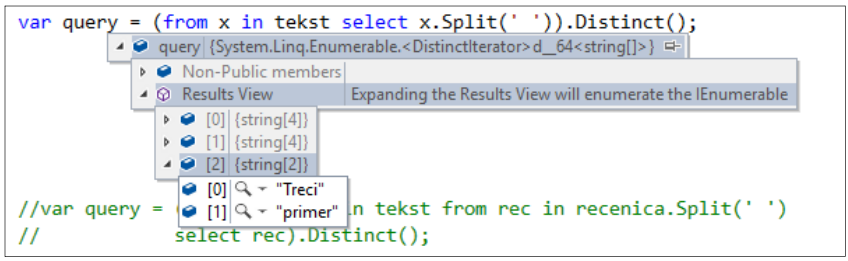
```
"Ovo je samo primer"
"Ovo je druga recenica"
"Treci primer"
```

To je prvi deo upita: `from recenice in tekst`. Zatim se vrši sledeće operacije u upitu. Na kolekciji izdvojenih stringova, **recenice**, izvodi se string operator **Split** koji deli string na osnovu ulaznog argumenta koji je karakter ili niza karaktera. Tako se dobijaju reči, tj objekti **reci**. Na kraju ova kolekcija se koristi za primenu funkcije koja izdvaja samo različite elemente. Rezultat je kolekcija različitih reči u tekstu.

Razlog je postojanje kolekcija unutar kolekcije **text**. Pogledajmo rezultat koji se dobija primenom upita bez ugnježdavanja upita tj. bez višestrukog from operatora, na primer:

```
var query = tekst.Select(x=> x.Split(' ')).Distinct();
// identicno sa
//  var  query  =  (from  x  in  tekst  select  x.Split('
')).Distinct();
```

dobija se rezultat koji je kolekcija od tri kolekcije. Svaka od tri kolekcije sadrži stringove tj. reči unutar rečenice na koju se odnosi, pogledajte sliku.

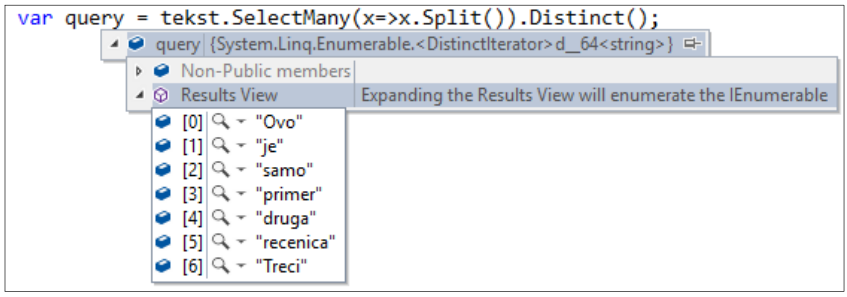


Slika 3.3. Prikaz rezultata Linq izraza u IDE okruženju

Drugi operator projekcije, `SelectMany`, sve rezultate vraća u jednu kolekciju. Na ovaj način upit:

```
var query = tekst.SelectMany(x=>x.Split()).Distinct();
```

vraća identičan rezultat kao više ugnježdenih upita.



Slika 3.4. Dodavanje klase Student projektu

Izdvajanje rezultata

Ovi operatori omogućavaju izdvajanje dela rezultata. Ovo je naročito smisleno kada je potrebno koristiti deo po deo dobijenih rezultata za prikaz tzv. straničenje. Osnovni operatori su:

- Skip ,
- Take.

Na primer, da bi dobili treću stranicu rezultata, pri čemu se uzima 10 zapisa po stranici, možete da koristite `Skip(20)` a zatim `Take(10).`

Takođe, postoje i operatori `SkipUntil`, `TakeUntil`.

Primer:

```
int[] numbers = { 1, 3, 5, 7, 9 };
var query = numbers.SkipWhile(n => n < 5).TakeWhile(n => n
< 10);
// daje 5, 7, 9
```

Združivanja

Kao i u standardnim SQL upitima, nekada je važno uključiti više tabela u upit, naravno po osnovu određenih kriterijuma. Slično važi i za Linq upite. Osnovni operatori združivanja su:

- **`Join`** sličan **`SQL INNER JOIN.`**
- **`GroupJoin`** sličan **`LEFT OUTER JOIN`**

Za potrebe primera rada sa ovim podacima potrebno je da uvedemo, osim klase `Person`, novu klasu `Roles`, kao u kodu:

```
class Role
{
    public int ID;
    public string name;
}
```

A zatim da uradimo inicijalizaciju podataka kao i ranije:

```
List<Role> roles = new List<Role> {
    new Role
    {
        ID=1,
        name = "Student"
    },
    new Role
    {
        ID=2,
        name = "Teacher"
    }
};
```

Povezivanje

Sada možemo uraditi prvo povezivanje i testiranje rezultata primenom Join operatora.

```
var query = from p in people
```

```
            join r in roles
            on p.IDRole
            equals r.ID
            select new
            {
                osoba = p.LastName + " " + p.FirstName,
                uloga = r.name
            };
```

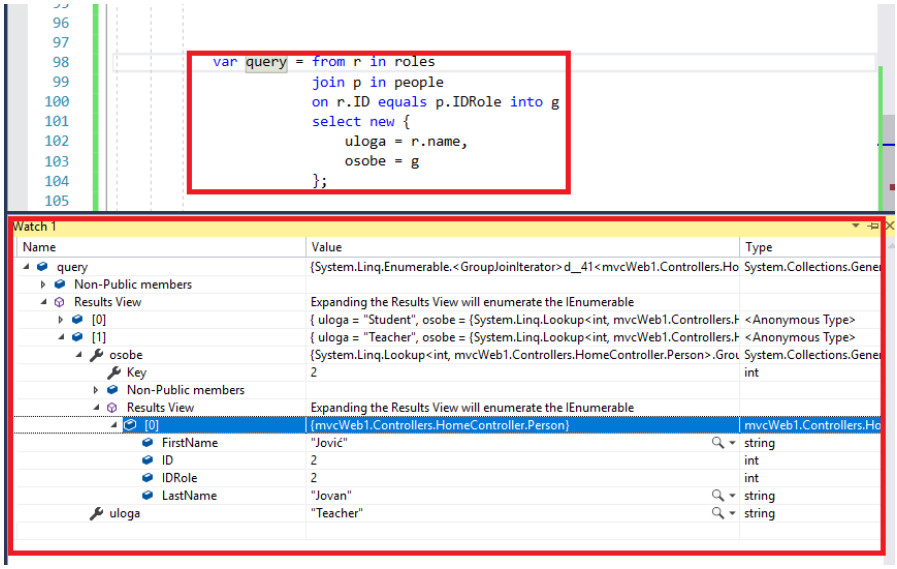
Kod je potpuno analogan unutrašnjem SQL združivanju. U našem slučaju dobija se lista osoba sa ulogom koja je preuzeta iz druge tabele.

Višestruko povezivanje

Jedna `join` klauzula u upitu koja se koristi zajedno sa `into` izrazom, naziva se `group join`. Na primer:

```
var query = from r in roles
            join p in people
            on r.ID equals p.IDRole into g
            select new {
                uloga = r.name,
                osobe = g
            };
```

Za razliku od prethodnog join upita koji daje parove elemenata za svako podudaranje, `group join` daje samo jedan rezultujući objekat za svaki element prve kolekcije, `roles`. Odgovarajući elementi druge kolekcije, koji su u ovom primeru `people`, grupisani su u jednu kolekciju. Kao rezultat upita dobija se anonimni tip za svako podudaranje, a sastoji se od jedne uloge i kolekcije osoba.



Slika 3.5. Prikaz rezultata Linq izraza preko *Watch* prozora u toku izvršavanja

Grupisanje

- GroupBy ,
- ToLookup

Ovi operatori vraćaju sekvencu tipa `IGrouping<K,V>`. Ovaj interfejs specificira da objekat izlaže svojstvo `Key` koje omogućava grupisanje.

Primer:

```
int[] niz = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = niz.ToLookup(i => i % 2);
foreach (IGrouping<int, int> group in query)
{
    Console.WriteLine("Key: {0}", group.Key);
    foreach (int element in group) {
Console.WriteLine(element); }
}
```

Vraćaju se dve grupe sa `Key` vrednostima 0,1. U svakoj grupi je lista objekata originalnog niza koji joj pripadaju.

Osnovna razlika između `GroupBy` i `ToLookup` operatora je što operator `GroupBy` obavlja izvršavanje sa kašnjenjem (eng. *lazy*). `ToLookup` obavlja izvršavanje odmah.

Opšti operatori

- `Empty` kreira praznu sekvencu `IEnumerable<T>.`

```
var empty = Enumerable.Empty<Employee>();
```

- `Range` operator generiše sekvencu brojeva
- `Repeat` generiše sekvencu bilo kojih vrednosti.

```
var empty = Enumerable.Empty<Employee>();
int start = 1;
int count = 10;
IEnumerable<int> numbers = Enumerable.Range(start,
count);
var tenTerminators = Enumerable.Repeat(new Employee {
Name = "Arnold" }, 10);
```

- `DefaultIfEmpty` generiše praznu kolekciju sa podrazumevanom vrednošću koja pripada tipu kada se primeni.

Operatori jednakosti

- `SequenceEquals`
  - Prolazi kroz dve sekvence i poredi objekte unutar obe da li su jednaki.

Primer:

```
Person e1 = new Person() { ID = 1 };
Person e2 = new Person() { ID = 2 };
Person e3 = new Person() { ID = 3 };
var employees1 = new List<Person>() { e1, e2, e3 };
var employees2 = new List<Person>() { e3, e2, e1 };
//false
bool result = employees1.SequenceEqual(employees2);
```

Element operatori

- `ElementAt`,
- `First`,
- `Last`,
- `Single`

Za svaki operator postoji odgovarajući `or default` operator koji se može koristiti da se izbegne izuzetak kada element ne postoji: `ElementAtOrDefault`, `FirstOrDefault`, `LastOrDefault`, `SingleOrDefault`.

Primer:

```
string[] empty = { };
string[] notEmpty = { "Zdravo", "Programeri" };
var result = empty.FirstOrDefault(); // null
result = notEmpty.Last(); // Programeri
result = notEmpty.ElementAt(1); // Programeri
result = empty.First(); // InvalidOperationException
result = notEmpty.Single(); // InvalidOperationException
result = notEmpty.First(s => s.StartsWith("Z"));
```

Osnovna razlika između operacija `First` i `Single` je što `Single` operator šalje izuzetak ako sekvenca ne sadrži jedan element, dok `First` vraća rezultat koji je prvi element. `First` šalje izuzetak samo ako ne postoji ni jedan element.

Konverzije

- `OfType`,
- `Cast`

`OfType` operator je i operator filtriranja – vraća samo objekte koje može kastovati tj. pretvoriti u neki tip, dok će operator `Cast` vršiti konverziju i pri tome će slati izuzetak ako ne može da kastuje sve objekte u neki tip.

```
object[] podatak = { "Pera", 3, "Aca" };
// kreira sekvencu od 2 stringa
var query1 = podatak.OfType<string>();
```

```
// izaziva izuzetak
var query2 = podatak.Cast<string>();
```

Spajanje

- Concat - operator koji spaja dve sekvence.

Sličan je *Union* operatoru ali ne izbacuje duplikate.

```
string[] ime = { "Joca", "Joca" };
string[] prezime = { "Ilic", "Mijic"};

//“Ilic”, “Joca”, “Joca”, “Mijic”
var nadovezivanje = ime.Concat(prezime).OrderBy(s => s);

//“Ilic”, “Joca”, “Mijic”
var unija = ime.Union(prezime).OrderBy(s => s);
```

Agregacija

- `Average`, `Count`, `LongCount` (za velike rezultate), `Max`, `Min`, `Sum`.

Agregatnim operacijama dobijaju se statistički podaci za kolekcije.

Primer:

```
int[] niz = { 2, 4, 6, 8, 10 };
var summary = new
{
    ProcessCount = niz.Count(),
    TotalThreads = niz.Sum(),
    MinThreads = niz.Min(),
    MaxThreads = niz.Max(),
    AvgThreads = niz.Average()
};
```

Pitanja i zadaci za proveru znanja

1. Objasnite šta je Linq?
2. Navedite prime jednog Linq izraza?
3. Linq se može koristiti gde postoji implementiran jedan interfejs. Koji?
4. Vrste Linq primene su: Linq to _____?
5. Pokušajte da objasnite metode proširivanja.
6. Šta je to lambda izraz? Navedite primer.
7. Kakvi su anonimni tipovi? Kada se koriste? Navedite primer.
8. Kako izgleda neki operator filtriranja?
9. Napiši Linq izraz za filtriranje svih zaposlenih sortiran po prezimenu, pa po imenu.
10. Ako promenljiva zemlje sadrži niz zemalja, kako bi izgledao Linq upit kojim bi izdvojili zaposlene koji pripadaju tom nizu zadatih zemalja?
11. Napiši upit čiji izlaz su proizvodi grupisani po kategorijama, tako da za svaku grupu imamo pripadajuću listu proizvoda.
12. Napiši upit koji će grupisati narudžbine (Order tabela) po vrednosti CustomerID, a zatim po CustomerID i EmployeeID.
13. Napiši upit za izdvajanje prvog i poslednje zaposlenog.
14. Koja je razlika u primeni operator First i FirstOrDefault i to prikaži na primeru?
15. Koja je razlika u primeni operator First i Single?
16. Uporedi operatore spajanja Concat i Union.
17. Uporedi operatore konverzije OfType i Cast.
18. Prikaži sumu svih artikala na jednoj narudžbenici.
19. Prikaži ukupne sume po narudžbenicama.

20. Koje znate agregatne operacije?

4. Entity Framework

U ovom poglavlju uvodimo objektno relaciono mapiranje kroz primenu radnog okvira *Entity Framework*. U okviru njega prikazaćemo formiranje modela odnosno osnovne koncepte razdvajanja aplikativne šeme modela od fizičke šeme kao i njihovo međusobno povezivanje putem mapiranja. Svaki postupak generisanja modela i rad sa modelom biće ilustrovan praktičnim primerom. Rad sa ovim radnim okvirom nije uslovljen bilo kojim tipom aplikacije.

Uvod

ADO.NET *Entity Framework* (EF) je deo Microsoft-ove nove generacije .NET tehnologija. Uvodi se sa namerom da se rad sa podacima olakša i učini još efikasnijim.

EF je komponenta ADO.NET koja koristi *Entity Data Model* (EDM).

EF donosi novine, a to su:

- Modelovanje: Uvodi se
  - konceptualni model (eng. Conceptual Models) koji predstavlja objekte i preko njega bolje odvajanje aplikativnih od stvarnih šema podataka.
  - fizički model ili model za skladištenje (eng. Storage Model) koji

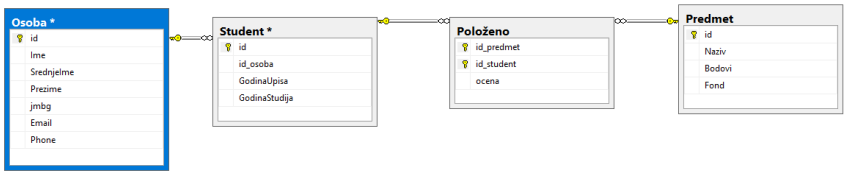
predstavlja objekte koji su neposredno povezani sa izvorom podataka i preko kojih se obavlja perzistencija podataka.

- Bolju integraciju: Integracija podataka kao objekata sa aplikativnim objektima.
- EF poseduje važnu LINQ podršku.

Koncepti

Svaka aplikacija koja radi sa podacima poseduje eksplicitno ili implicitno definisan konceptualni model podataka tj. model koji se koristi u aplikaciji.

Pretpostavimo sledeću Db šemu podataka.



Slika 4.1. Primer šeme podataka

Pogledajmo kako bi izgledao jedan tradicionalni upit za gornji primer. Neka se upitom generiše pogled na studente treće koji su upisani posle 2015 godine, a upit treba da prikaže imena i godinu studija studenta.

```
SELECT o.Ime, s.GodinaStudija
FROM Student s
INNER JOIN Osoba o ON o.id = s.id
WHERE s.GodinaUpisa >= '2015-01-01’
AND s.GodinaStudija = 3
```

Zapazite da se za kreiranje odgovarajućeg prikaza podataka pogleda koristi jedan JOIN.

Ako sada proširimo zahtev uslovom da se prikažu položeni ispiti sa ocenom 10, upit bi se proširio sa još dva JOIN-a:

```
SELECT o.ime, o.prezime, s.indeks, pr.naziv
FROM  Polozeno po
INNER JOIN Predmet pr
      ON po.id_predmet = pr.id
INNER JOIN Student s
      ON s.id = po.id_student
INNER JOIN Osoba o
      ON o.id = s.id
WHERE po.ocena = 10
AND   s.GodinaStudija = 3
```

Očigledno da upit postaje komplikovan za primenu. Možemo reći da je Db šema podataka u bazi izdeljena i da nije prilagođena aplikaciji. Treba razumeti model baze podataka, a teško je i pratiti relacije između tabela u bazi. Dakle, bilo bi od koristi:

- Iskoristiti aplikativne zahteve za drugačije modelovanje, iako postoje već kreirane Db šeme.
- Izbeći da se Db šema provlače kroz aplikativni kod.

Tradicionalan model zasnovan je na modelu podataka koji je u bazi podataka. Njega čine: tabele, pogledi, uskladištene procedure kao i relacije.

Umesto tradicionalnog, primenom EF-a koristi se objektno-orijentisan tj. konceptualan model, pogodniji za korišćenje u razvoju aplikacija. Ovaj model čine: objekti, ponašanja objekata, svojstva, nasleđivanja, kompleksni tipovi.

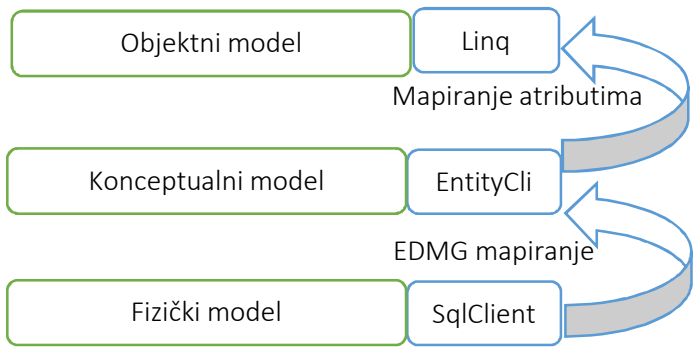
Struktura modela

Model se zapisuje kao fajl metapodataka u XML formatu. Tačnije, postoje 3 celine:

- .CSDL (eng. Conceptual Schema Definition Language)
  Mapira tipove entiteta koji se koriste u konceptualnom modelu
- .SSDL (eng. Store Schema Definition Language)
  Šema podataka za bazu.
- .MSL (eng. Mapping Schema Language)
  Mapira konceptualnu šemu u šemu podataka baze

Inače, model se kreira koristeći alatke pri dizajnu ili EMDGen. Mora biti na raspolaganju u toku izvršavanja programa, a kopira se u *bin* folder aplikacije, ili se koristi ugrađivanje resursa tokom *build* opcije.

Dakle, tri modela učestvuju u formiranju celokupnog modela za rad sa bazom podataka i međusobno su povezani.



Slika 4.2. Povezanost modela

Koncepti modela

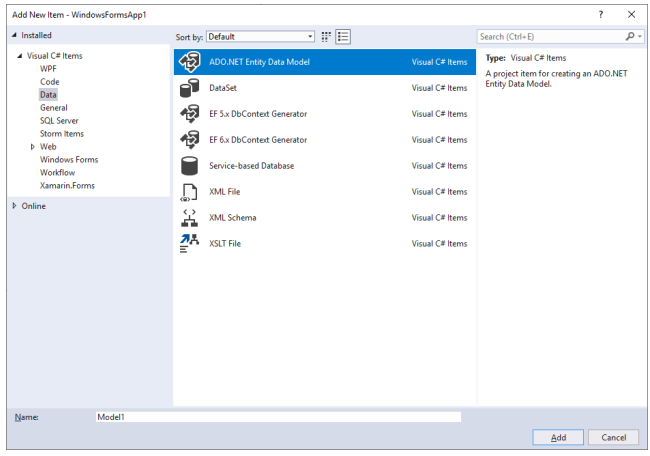
Model podataka koji koristimo naziva se EDM model (eng. Entity Data Model). Ovaj model se sastoji od elemenata *entitet-relacije*. Osnovni pojmovi modela su:

- ***Tip*** *Entiteta* je struktuirani zapis sa ključem.
- *Entitet* je jedna **instanca** tipa entiteta.
- Entiteti su grupisani u **skupove** entiteta - eng. *Entity-Sets.*
- Tip jednog entiteta može **naslediti** drugi tip.

Kreiranje modela

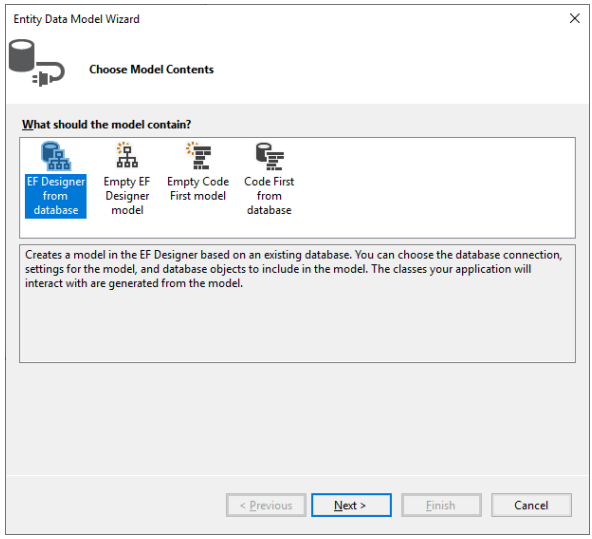
Postupak kreiranje modela na osnovu šeme koja postoji u bazi, sastoji se od nekoliko koraka.

Korak 1. Dodavanje nove stavke



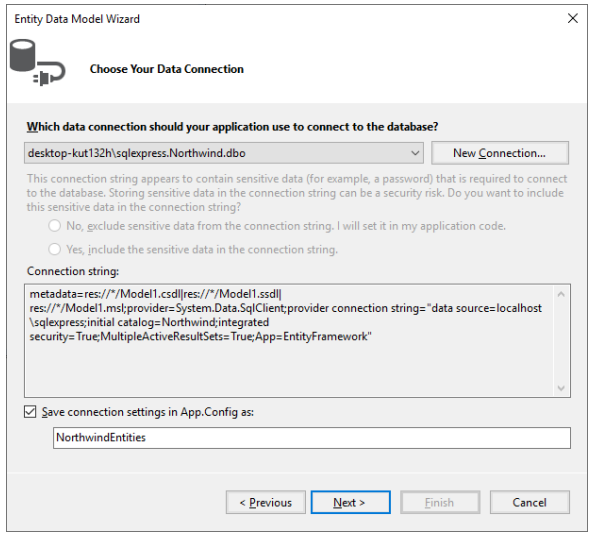
Slika 4.3. Dodavanje nove stavke: prvi korak u kreiranju modela

Korak 2. Izbor opcije za kreiranje EDM modela na osnovu Db šeme baze podataka.



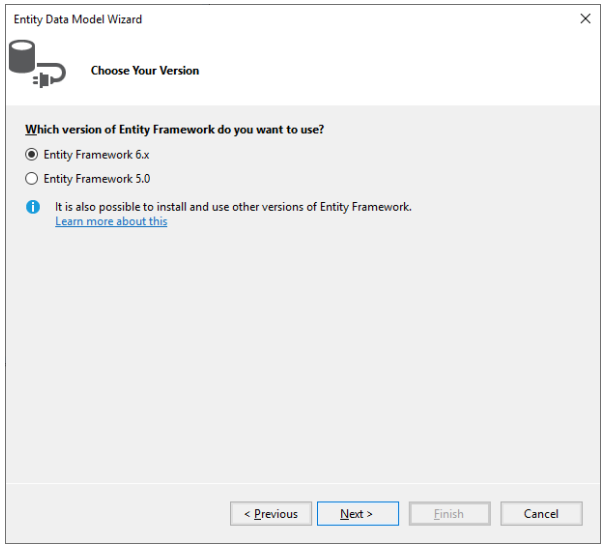
Slika 4.4. Izbor opcije za formiranje EDM modela iz baze

Korak 3. Izbor baze za koju se kreira EDM model. Ovo se vrši zapravo definisanjem konekcije do odgovarajuće baze tj. izvora podataka.



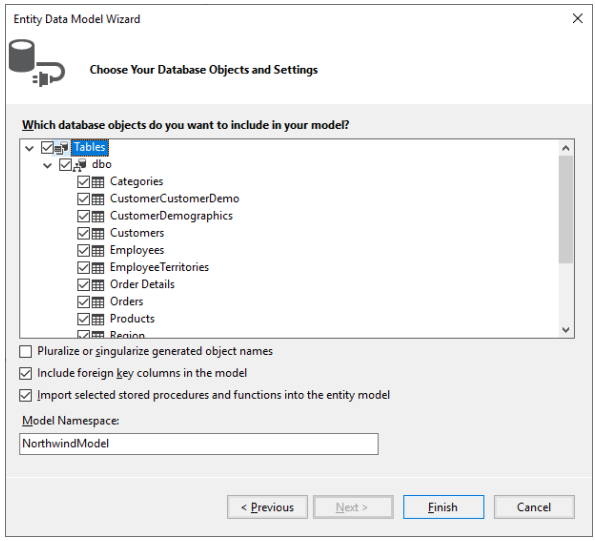
Slika 4.5. Izbor odgovarajuće konekcije

Korak 4. Odabir verzije EF za kreiranje modela.



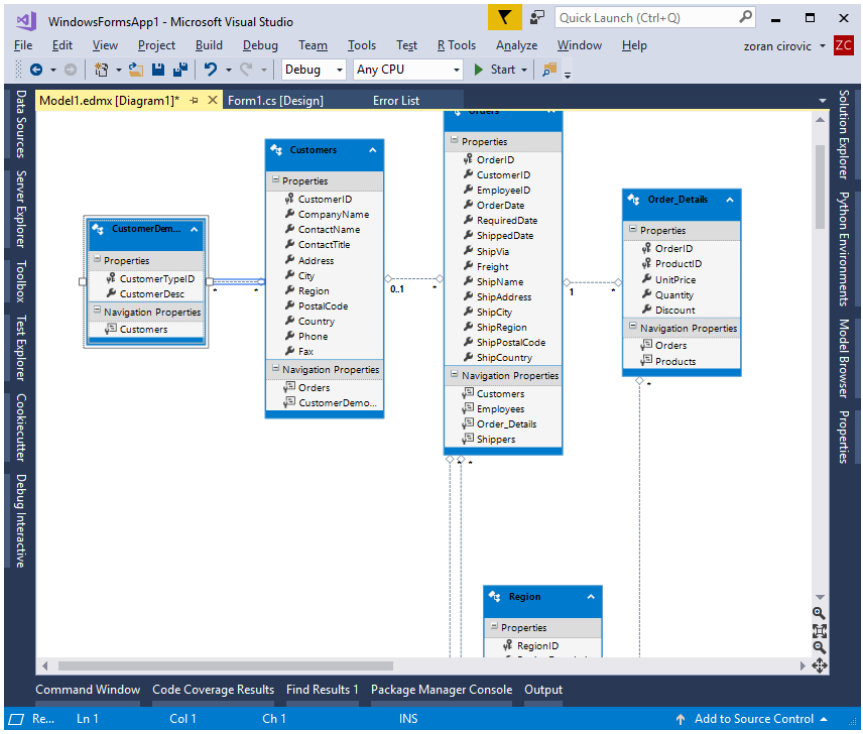
Slika 4.6. Izbor verzije EF-a za formiranje modela

Korak 5. Izbor objekata baze koji će učestvovati u formiranju EDMa. Tu se mogu koristiti ne samo tabele već i drugi objekti.



Slika 4.7. Izbor objekata baze koji učestvuju u EDM modelu

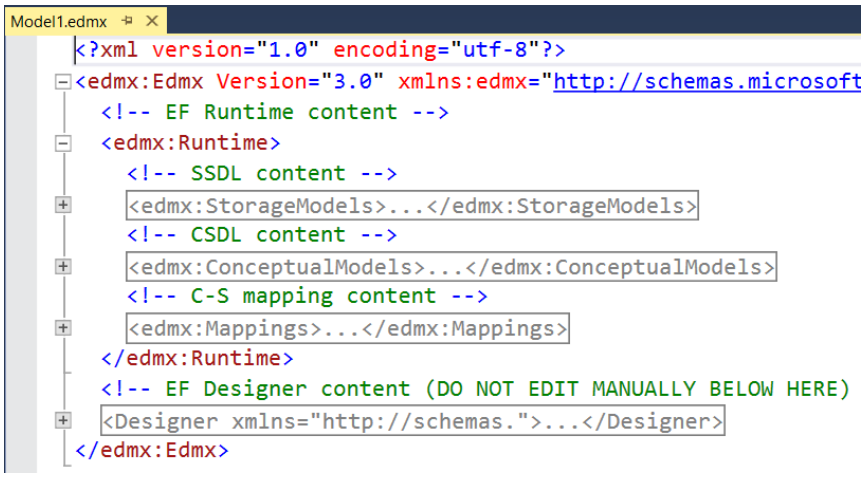
Konačno, formiran je model. Model je prikazan u grafičkom obliku na način na koji se obično prikazuju entiteti i veze između njih. Obratite pažnju da je prikaz zapravo samo grafička interpretacija fajla Model1.edmx odnosno jednog XML fajla koji sadrži opis EDM modela i to na način koji smo već opisali.



Slika 4.8. Grafički prikaz modela

Zapis modela

Dakle, kreirani EDM model je, ništa drugo nego jedan XML dokument koji je sastavljen iz delova koje čine model. Fajl ima ekstenziju edmx i odvojene tri celine. Pogledajmo kako to izgleda u konkretnom slučaju za upravo kreirani model.



Model je u celosti zapisan u XML dokumentu. Dakle tu je konceptualna šema, fizički model kao i mapiranje koje povezuje ta dva dela.

Naravno, programer u radu sa entitetima nekog modela manipuliše objektima. Opis entitetskih objekata, tj. tipova entiteta, može da vidi kao i da utiče na vrednosti svojstava. U fazi projektovanja, pre pokretanja programa, to može da uradi preko IDE okruženja.

EnitiySet

Kontejner za objekte jednog entitet tipa naziva se *EntitySet*. Dva su osnova polja koje sadrži skup entiteta: *Name*, *EntityType*.

EntityType definiše tip objekta sa kojim se radi preko punog naziva. Na primer:

```
<EntitySet Name="Customers" EntityType="Self.Customers"
Schema="dbo" store:Type="Tables" />
<EntitySet Name="Employees" EntityType="Self.Employees"
Schema="dbo" store:Type="Tables" />
```

EntityType

Tip podataka u modelu je *EntityType*. U prethodnom primeru za *EntitySet* definišu se *Customers* i *Employees*. Tipovi entiteta opisani su XSD šemom podataka.

Tip entiteta čine tri različite komponente. To su:

- Ključ,
- Svojstava – Property
- Navigaciona svojstva

Kada se u XML šemi proširi tip *Address* dobija se XML koji je prikazan. Prepoznaju se:

- ključ tj. *Key* i
- svojstva tj. *Property* elemenata.

```
<EntityType Name="Customers">
  <Key>
    <PropertyRef Name="CustomerID" />
  </Key>
  <Property Name="CustomerID" Type="String" MaxLength="5"
FixedLength="true" Unicode="true" Nullable="false" />
  <Property Name="CompanyName" Type="String" MaxLength="40"
FixedLength="false" Unicode="true" Nullable="false" />
. . .
  <NavigationProperty Name="Orders"
Relationship="Self.FK_Orders_Customers" FromRole="Customers"
ToRole="Orders" />
  <NavigationProperty Name="CustomerDemographics"
Relationship="Self.CustomerCustomerDemo" FromRole="Customers"
ToRole="CustomerDemographics" />
</EntityType>
```

Key

Definiše koja svojstva formiraju identitet tj. jedinstvenost entiteta.

Može biti sačinjen od više svojstava.

U Dizajneru i kodu predstavlja se sa *EntityKey*.

Property

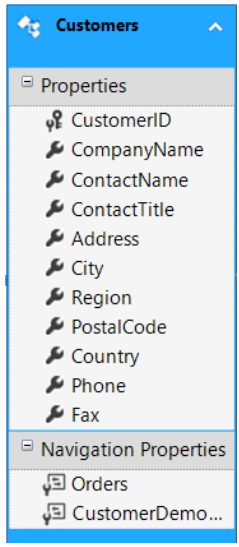
Oni su definisani imenom ali i dodatnim tipom podataka. Tipovi podataka koji definišu ova svojstva nazivaju je prostim tipovima – (eng. *simple types*). Ovo su osnovni tipovi u EF objektnom modelu koji su najbliži tipovima u .NET Framework-u. Međutim, osnovni tipovi u Entity Framework-u se koriste samo za definisanje svojstava entiteta.

Osnovna svojstva ovih elemenata se vide i iz *Properties* prozora kao na narednoj slici.

Navigation properties

Navigaciona svojstva su čvrsto povezana sa asocijacijama koje predstavljaju linije između entiteta. Koristeći ova svojstva postoje veze između entiteta, a navigaciona svojstva su kreirana svojstva u jednom objektu preko kojih se ostvaruje veza sa drugim entitetima tj. objektima.

Pogled na entitetski tip vidljiv je i preko grafičkog dizajna koji uobličava XML šemu u grafičke objekte koje lakše razumemo i sa kojima je jednostavnije raditi.



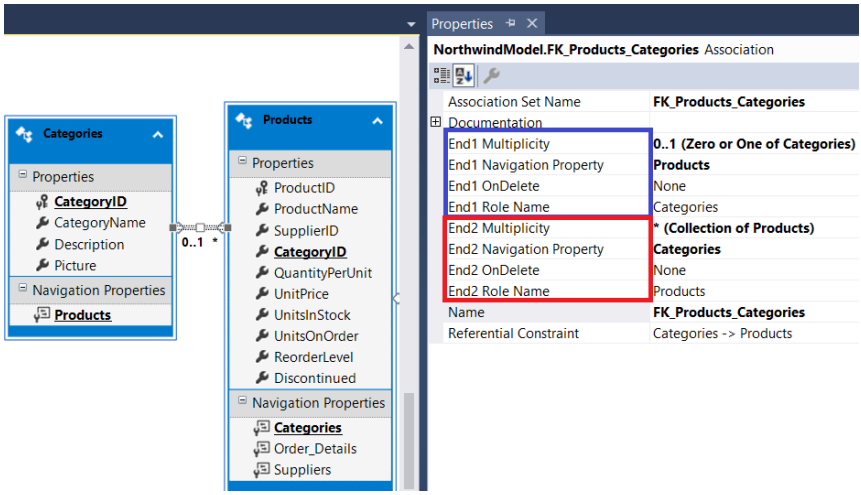
Slika 4.9. Grafički prikaz jednog entiteta

Asocijacije

Definišu veze između entitetskih tipova. Asocijacija nije isto što i relacija između tabela mada ima puno sličnosti. Jednu asocijaciju definišu krajnje tačke, entiteti koji su uključeni u vezu, kao i njihova multiplikacija. Pri automatizovano generisanom modelu asocijacijama se definišu veze između entiteta.

U primeru Northwind modela, postoji više formiranih asocijacija. Recimo asocijacija između *Products* i *Categories*, ukazujući da postoji veza između njih.

Inače, naziv konkretne asocijacije preuzima se iz predefinisanog imena relacije u bazi tokom faze kreiranja modela pomoću IDE "čarobnjaka".

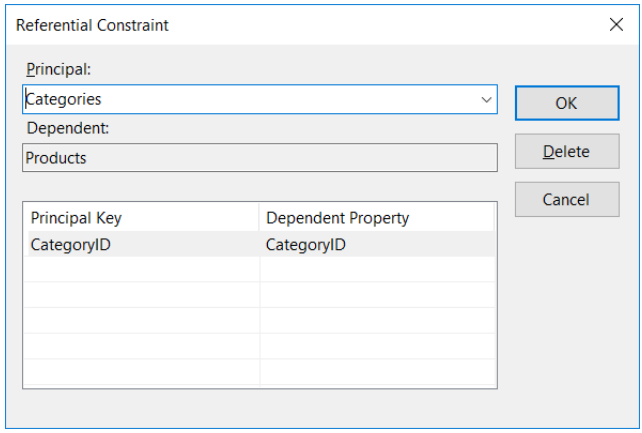


Slika 4. 10. Primer jedne asocijacije

Properties window za asocijaciju iz primera je prikazan na slici. Vidi se da je asocijacija sastavljena od svojstava koja posebno definišu vezu na oba kraju. Moguće vrednost za vrstu veze je: 1, 0..1, * (jedan, nula ili jedan, više).

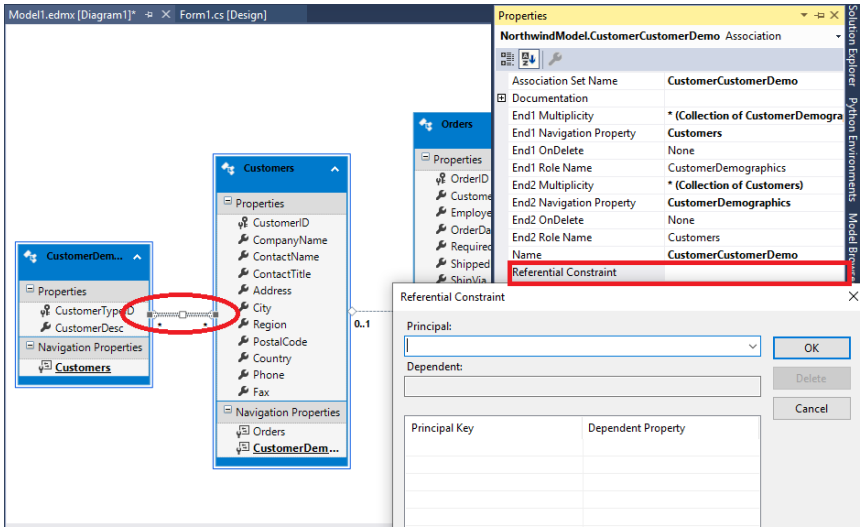
Zapazimo svojstvo ***Referential Constraint***.

U modelu koji sadrži strane ključeve u entitetima, kao što je svojstvo *CategoryID* u *Products*, definišu se zavisnost između entiteta koji su u relaciji. Neki objekat tipa *Categories* može, a ne mora, pokazivati na neke objekte iz *Products*.



Slika 4. 11. Definisano referencijalno ograničenje

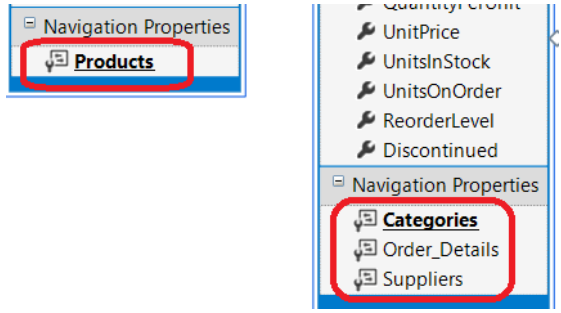
Pogledajmo još jedan neobičan detalj u modelu. Veza između entiteta *Customers* i *CustomerDemographics* je **više na više**. Takođe zapazimo da ne postoji pridruženo referencijalno ograničenje. Detaljnije o ovom slučaju biće opisano u narednom poglavlju.



Slika 4.12. Primer veze „više na više“

Navigation Property

Već smo rekli da su navigaciona svojstva deo entitetskih tipova i čemu služe. Na primer, kada se radi sa objektom *Categories* u kodu, svojstvo *Products* će se pojavljivati baš kao i druga svojstva. Mada su druga svojstva označena kao skalarna, tj. kao svojstva koja su vrednosti, navigaciono svojstvo opisuje kako se dolazi do entiteta sa kojim postoji veza. Slično, objekat tipa *Products* ima navigaciona svojstva: *Categories*, *Order_Details*, *Suppliers*.



Slika 4.13. Prikaz navigacionih svojstava u modelu

Važno svojstvo navigacije je *Association*. Ovo svojstvo ukazuje na asocijaciju u modelu koja sadrži informacije u vezi navigacije do entiteta u relaciji.

Mapiranje

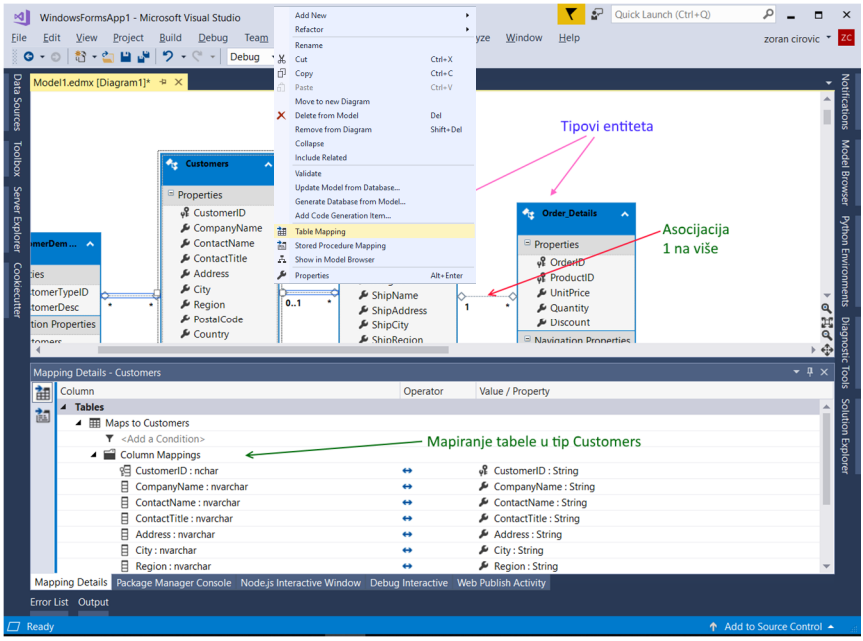
EDM je ORM model tj. objektno-relaciono-mapirani model. Mapiranje se izvodi između konceptualnog sloja odnosno modela koji se koristi na aplikativnom nivou i fizičkog odnosno skladišnog modela.

Mapiranje je poslednja sekcija u **edmx** fajlu, a ujedno i deo EDM modela.

Mapiranje jeste jedna sekcija u edmx fajlu, ali se mapiranje podešava po pravilu koristeći Dizajner. Dvostrukim klikom na edmx fajl modela u

Solution Explorer-u otvara se grafički editor preko koga možemo pristupiti editovanju mapiranja.

Da bi videli mapiranje, treba otvoriti prozor *Mapping Details window*. Na primer, desni klik na entitet *Contact,* a zatim se odabere opcija menija *Table Mapping*.



Slika 4.14. Prozor za uređivanje mapiranja

Klase modela

Pogledajmo jedan primer upotrebe modela.

```
using (var context = new NorthwindEntities())
{
    var employees = context.Employees;
    foreach (var employee in employees)
    {
        string ime = employee.FirstName;
        string prezime = employee.LastName;
    }
}
```

Očigledno da u projektu postoje kreirane klase za rad sa modelom. Klase su kreirane tokom generisanja modela, a mogu se naknadno i menjati. Svakom objektu iz baze, koji je označen tokom generisanja modela pridruženi su odgovarajući entiteti.

Fajl *Model.Designer.cs* je automatski generisan i možemo ga pogledati preko prozora *Solution Explorer*. U tom fajlu je definisan model koji se koristi.

```
public partial class NorthwindEntities : DbContext
{
    public NorthwindEntities()
        : base("name=NorthwindEntities"){}
    protected override void OnModelCreating(DbModelBuilder
modelBuilder){ . . . }
    public virtual DbSet<Categories> Categories { get; set; }
    public virtual DbSet<Products> Products { get; set; }
    public virtual DbSet<Region> Region { get; set; }
    public virtual DbSet<Shippers> Shippers { get; set; }
    public virtual DbSet<Suppliers> Suppliers { get; set; }

    . . .
    public virtual DbSet<Territories> Territories { get; set;
}
}
```

Dakle, model je klasa izvedena iz `DbContext` klase. Elementi ove klase su skupovi pojedinih entiteta. Svaki skup je opisan tipom `DbSet<EntityTip>`.

Generator čita konceptualni sloj modela i na osnovu njega kreira *ObjectContext* klasu zasnovanu na *EntityContainer-u*, a zatim jednu klasu entiteta za svaki entitet u modelu.

Pomenimo na ovom mestu, a u ovoj knjizi će biti naknadno obrađeno u poglavlju *Code first*, da se mogu najpre formiraju klase modela, a zatim se od njih kreirati tabele opet u proizvoljnoj bazi podataka.

Dobavljanje podataka

Integracija funkcionalnosti delova modela i baze omogućava lako dobavljanje podataka odnosno upotrebu modela preko Linq izraza. Pogledajmo primer.

Recimo da je potrebno izvršiti preuzimanje liste objekata tipa `Emplyees` za koje je postavljen uslov da je `Country == "USA"`. Rezultujući kod, koji uključuje odgovarajući Linq izraz je:

```
using (NorthwindEntities ctx = new NorthwindEntities())
{
    var q = from z in ctx.Employees
            where z.Country == "USA"
            select z;
}
```

Dobijeni rezultat je kolekcija objekata koji su sadržani u skupu `Employees`. Rezultat takođe implementira interfejs `IEnumerable.`

Izmene na podacima

ObjectContext sadrži metodu *SaveChanges*, koja omogućava da čuvaju promene na entitetima. Gde se čuvaju promene? Uobičajeno je u bazi, a podaci o konekciji se čuvaju u konfiguracionom fajlu. Pozivom metode *SaveChanges* biće provereni svi objekti, tačnije njihova stanja, *ObjectStateEntry,* kojima se upravlja od strane konteksta čije stanje nije *Unchanged,* a zatim će biti korišćeni detalji za izgradnju posebnih upita *Insert*, *Update*, odnosno *Delete* koji se šalju bazi.

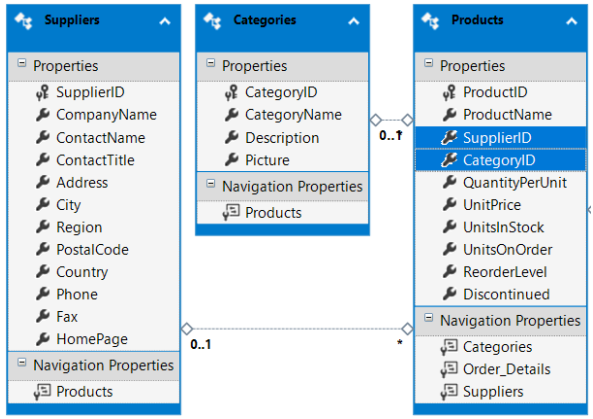
```
using (NorthwindEntities ctx = new NorthwindEntities())
{
    // menjamo kategoriju za koju je CategoryID == 4
    var cat = ctx.Categories.FirstOrDefault(x => x.CategoryID
== 4);
    if (cat != null) cat.Description = "Domaći...";
    ctx.SaveChanges(); // metoda vraća broj izmenjenih redova
}
```

Dodavanje novog objekta

Dodavanje novog objekta nekom skupu entiteta se izvršava na dosledan način tj. kao i u prethodnim primerima. Najpre kreiramo novi objekat odgovarajućeg tipa, koji zatim dodajem skupu *DbSet*, pa se na kraju vrši snimanje promena. Na primer:

```
using (NorthwindEntities ctx = new NorthwindEntities())
{
    Categories novaKat = new Categories();
    novaKat.CategoryName = "novaKat";
    ctx.Categories.Add(novaKat);
    ctx.SaveChanges();
}
```

Na ovom mestu pogledaćemo veoma efikasan način dodavanja više povezanih entiteta. Pogledajmo jedan specifičan primer zasnovan na tri tabele: *Products*, *Categories* i *Suppliers*. Zapazite istovremeno i odgovarajuće entitete odnosno veze:



Slika 4.15. Više povezanih entiteta

Ako se istovremeno dodaje novi objekat *Products* koji je povezan sa roditeljskim entitetima *Suppliers* i *Categories*, kod bi mogao da bude zasnovan samo na objektima i bez neposredne primene ključeva povezanih entiteta:

```
using (NorthwindEntities ctx = new NorthwindEntities())
{
    Categories novaKat = new Categories();
    novaKat.CategoryName = "novaKat";
    ctx.Categories.Add(novaKat);
    Suppliers noviSup = new Suppliers();
    noviSup.CompanyName = "novaKompanija";
    ctx.Suppliers.Add(noviSup);

    Products noviProizod = new Products();
    noviProizod.ProductName = "NaziNovogProizvoda";
    noviProizod.Categories = novaKat;
    noviProizod.Suppliers = noviSup;
    ctx.Products.Add(noviProizod);

    ctx.SaveChanges();
}
```

Brisanje

Na analogan način vrši se brisanje objekata, odnosno redova u odgovarajućoj tabeli. Ako treba da obrišemo zaposlenog čiji je `EmployeeID == 14` kod bi bio sledeći:

```
using (NorthwindEntities ctx = new NorthwindEntities())
{
    Employees emp = ctx.Employees.Where(x => x.EmployeeID ==
14).First();
    ctx.Employees.Remove(emp);
    ctx.SaveChanges();
}
```

Konekcija

Konekcijski string predstavlja jednu vrednost tipa string koja je sastavljena od svih parametara za povezivanje sa izvorom podataka. Parametri u konekcijskom stringu su razdvojeni karakterom ';'. Kreiranje stringa obično se vrši primenom specijalizovane klase *ConnectionStringBuilder*.

Naredni kod koristi dva objekta *ConnectionStringBuilders*: prvi za kreiranje SQL Server konekcijskog stringa, a drugi za kreiranje EF konekcijskog stringa.

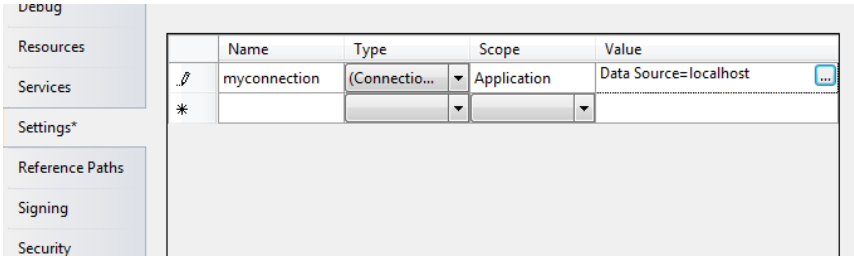
```
SqlConnectionStringBuilder sqlCnStrBuilder = new
SqlConnectionStringBuilder();
sqlCnStrBuilder.DataSource = ".";
sqlCnStrBuilder.InitialCatalog = "Northwind";
sqlCnStrBuilder.IntegratedSecurity = true;
sqlCnStrBuilder.MultipleActiveResultSets = true;

EntityConnectionStringBuilder entityCnStrBuilder = new
EntityConnectionStringBuilder();
entityCnStrBuilder.Provider = "System.Data.SqlClient";
entityCnStrBuilder.ProviderConnectionString =
sqlCnStrBuilder.ConnectionString;
entityCnStrBuilder.Metadata =
@"res://*/Northwind.csdl|res://*/Northwind.ssdl|res://*/North
wind.msl";
```

Konekcijski string se obično zna pre pokretanja aplikacije, a spada u podešavanja koja se mogu menjati i bez promene koda. Zato se konekcijski string čuva i koristi preko *config* fajla.

```
  <connectionStrings>
    <add name="NorthwindEntities"
connectionString="metadata=res://*/Model1.csdl|res://*/Model1
.ssdl|res://*/Model1.msl;provider=System.Data.SqlClient;provi
der connection string=&quot;data
source=localhost\sqlexpress;initial
catalog=Northwind;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramewo
rk&quot;" providerName="System.Data.EntityClient" />
  <connectionStrings>
```

Naravno, konekcijski string se može editovati preko dela IDE-a koji se odnosi na podešavanja projekta aplikacije.

Desnim klikom na projekat odaberite opciju Properties u kontekstnom meniju, a zatim odaberite karticu Settings. Pošto odaberete ime za vrednost konekcijskog string koji će biti dodat u config fajlu odaberemo *Type* kao *(ConnectonString)*. Zatim se u polje *Value* za vrednost aktivira čarobnjak preko dugmeta označenog sa tri tačke. Pogledajte sliku.

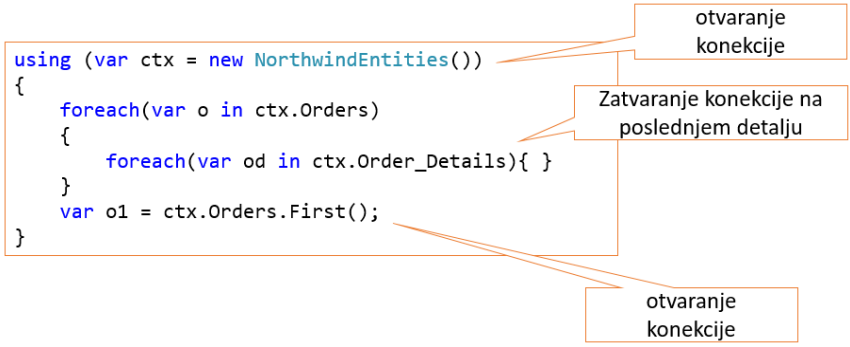


Slika 4.16. Prozor za podešavanje konekcijskog stringa

Automatsko otvaranje konekcija

Rad sa konekcijama je veoma važan za efikasno izvršavanje upita. Otvaranje konekcije je relativno spora operacija, a istovremene konekcije na serveru takođe mogu da budu ne samo skupe u pogledu vremena već i novca.

Konekcija se OTVARA odnosno ZATVARA automatski pri svakom izvršavanju upita. Ovaj automatizam olakšava pisanje koda, ali takođe čini napisani kod lak za razumevanje i jednostavan za održavanje.



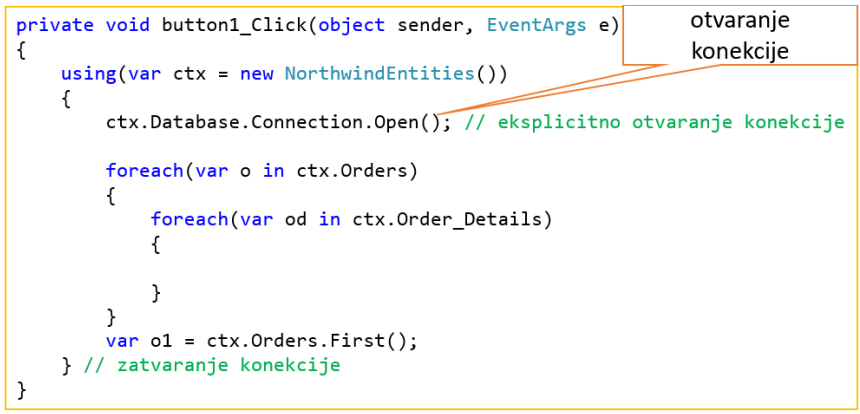
Slika 4.17. Primer sa ilustrovanim automatizmom otvaranja konekcije

Eksplicitno otvaranje konekcije

Kada se konekcija eksplicitno otvori ona ostaje otvorena i može omogućiti u nekim slučajevima efikasnije izvršavanje više komandi.

Ako je konekcija eksplicitno otvorena, zatvaranje se obavlja:

- eksplicitnim zatvaranjem ili
- pozivom *Dispose* metode ili
- na kraju *using* bloka.



Slika 4.18. Primer eksplicitnog otvaranja/zatvaranja konekcije

Podrazumevana transakcija

Pri radu sa konekcijama odnosno komandama koristeći EF implicitno se koriste i transakcije. Dakle, postoji ugrađena transakcija koja se kreira za izvršavanje metode **SaveChanges.** To znači, da će sve obuhvaćene promene biti izvedene kao jedna celina ili neće ni biti urađena ni jedna! Na primer:

```
using (var context = new BAEntities())
{
    var contact = context.Contacts.Where(c => c.ContactID
== 5)
    .FirstOrDefault();
    context.DeleteObject(contact);
    var reservation = context.Reservations.FirstOrDefault;
    var payment = new Payment();
    payment.Amount = "500";
    payment.PaymentDate = System.DateTime.Now;
```

```
    payment.Reservation = reservation;
    context.SaveChanges();
}
```

Pokušaj brisanja jednog *Contact* objekta iz baze neće uspeti zbog referecijalnog ograničenja. Zato će biti automatski pokrenut roolback za sve komande.

Svojstvo. ObjectContext.AcceptAllChanges menja stanje svih entiteta koji su menjani. Postavljaju se *OriginalValues* preklapajući tekuće vrednosti i menja se vrednost svojstva *EntityState* na *Unchanged*.

U toku izvršenja *SaveChanges,* nakon kraja podrazumevane transakcije, metoda *AcceptAllChanges* se automatizovano poziva, na taj način obezbeđuje da *ObjectContext* bude ažuran, a njegovi entiteti odgovaraju podacima u bazi.

Pitanja i zadaci za proveru znanja

1. Objasnite svojim rečima ADO.NET Entity Framework.
2. Navedite vaš primer, koristeći Northwind bazu, jednog tradicionalnog upita koji uključuje 3 tabele.
3. Od kojih elemenata se sastoji struktura EDM modela? Kako su ti elementi povezani?
4. Kreirajte vaš EDM model na osnovu Northwind baze.
5. Nađite u projektu u kome koristite EDM model konekcijski string.
6. Prikažite model u grafičkom i u XML editoru.
7. Koja svojstva su tipična za svaki *EntityType*?
8. Objasnite upotrebu navigacionih svojstava i asocijacija.

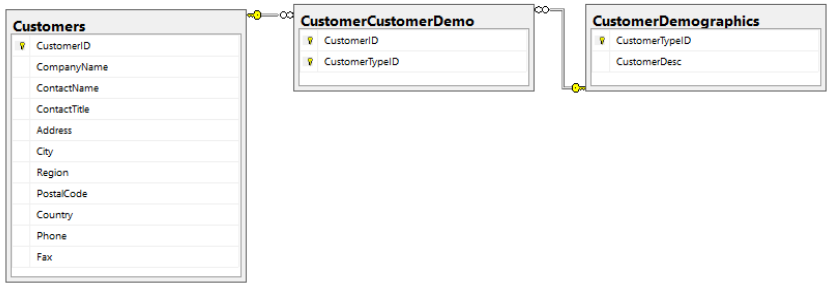
9. Šta je to mapiranje i kako možete u konkretnom slučaju da pogledate realizaciju mapiranja?
10. Iz koje klase je izvedena klasa modela?
11. Od čega se sastoji realizacija klase modela?
12. Šta su klase DbSet odnosno DbContext?
13. Napišite primer za prikaz zaposlenih iz neke zemlje sortiran po prezimenu.
14. Napišite primer za dodavanje novog zaposlenog.
15. Napišite primer za promenu podataka postojećeg zaposlenog.
16. Napišite primer za brisanje jednog zaposlenog.
17. Pokušajte da promenite konekciju do baze preko konfiguracionog fajla, a zatim da pogledate konekciju preko prozora za podešavanje.
18. Kakva je to podrazumevana transakcija?

5. EDM primene

Cilj ovog poglavlja je da istakne specifičnosti objektno relacionog mapiranja i uvede čitaoca u praktičnu primenu modela i manipulaciju sa podacima. Poglavlje obuhvata rad sa postojećim modelom uz karakteristične operacije sa podacima, ali takođe i promene na konceptualnom sloju modela kao primere upotrebe izmenjenih modela.

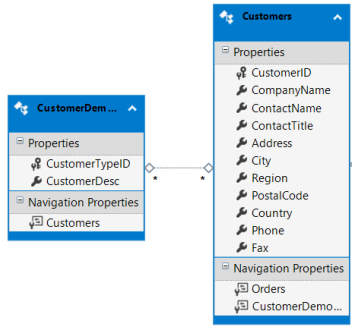
Mapiranje "više na više"

Pogledajmo tri tabele u bazi Northwind.



Slika 5.1. Povezivanje više na više preko pomoćne tabele u bazi

Veza između tabele *Customers* i *CustomerDemographics* je ostvarena preko *CustomerCustomerDemo* kao „više na više". Nakon automatizovano kreiranja modela formirani su odgovarajući entiteti u modelu kao na sledećoj slici. Obratite pažnju da je veza između entiteta „više na više" bez tabele posrednika.

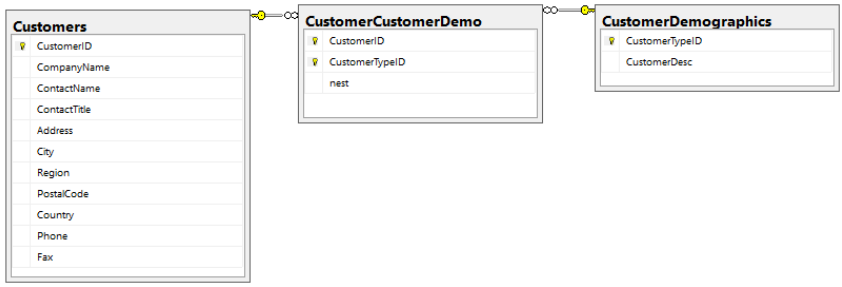


Slika 5.2. Povezivanje više u EDM modelu

Sada pogledajte primer kojim se dodaje više *Customers* entiteta za jedan *CustomerDemographics* i obrnuto:

```
using (NorthwindEntities ctx = new NorthwindEntities())
{
    // dodavanje jednog CustomerDemographics za dva zapisa u
Customers
    var cDem = new CustomerDemographics {
CustomerTypeID="abcdefghi0", CustomerDesc = "cDem" };
    var cA = new Customers { CustomerID = "ccccA",
CompanyName = "A" };
    var cB = new Customers { CustomerID = "ccccB",
CompanyName = "B" };
    cDem.Customers.Add(cA);
    cDem.Customers.Add(cB);
    ctx.CustomerDemographics.Add(cDem);

    // dodavanje jednog Customers za dva CustomerDemographics
    var cDem1 = new CustomerDemographics { CustomerTypeID =
"abcdefghi1", CustomerDesc = "cDem1" };
    var cDem2 = new CustomerDemographics { CustomerTypeID =
"abcdefghi2", CustomerDesc = "cDem2" };
    var c = new Customers { CustomerID = "ccccC", CompanyName
= "C" };
    c.CustomerDemographics.Add(cDem1);
    c.CustomerDemographics.Add(cDem2);
    ctx.Customers.Add(c);

    ctx.SaveChanges();

}
```

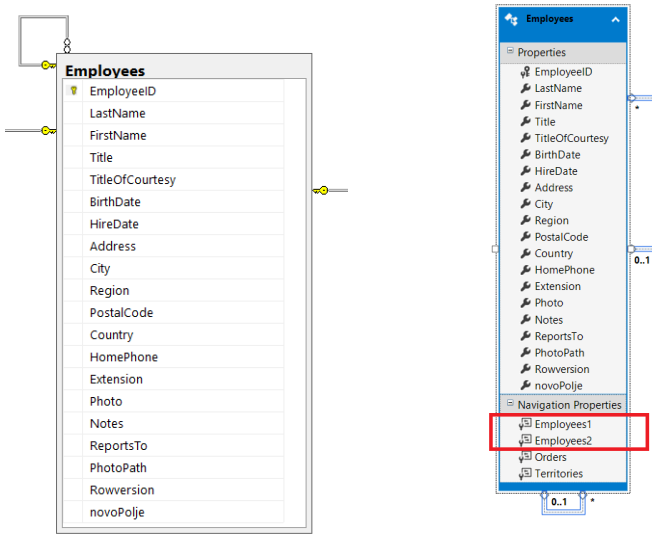
Ukoliko vezna tabela sadrži bar još jedno polje, osim ključeva unakrsnih tabela, onda se ta tabela mora naći u modelu pa samim tim mora biti uključena pri pisanju koda.



Slika 5.3. Povezivanje više na više preko pomoćne tabele u bazi

Tabela sa referencom na samu sebe

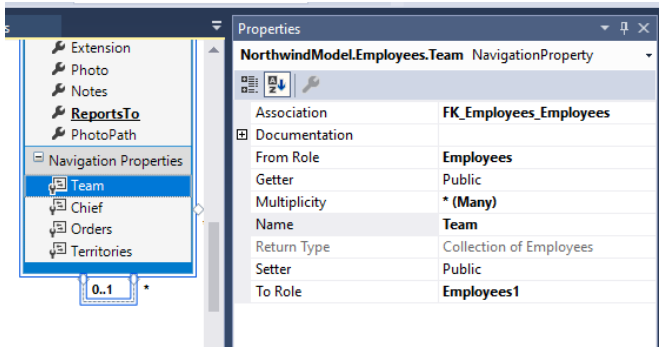
Ukoliko neka tabela sadrži strani ključ koji ukazuje na istu tabelu, relacijske veze u bazi odnosno asocijacije u modelu su date na slici.



Slika 5.4. Veza sa sopstvenom tabelom odnosno entitetom

Pri tome se formiraju dva navigaciona svojstva. Jedno se odnosi na roditeljsku kategoriju i ima **`0..1`** stranu relacije. Drugo se odnosi na dete i ima * uz relaciju. U konkretnom slučaju, tabela *Employees* ima takvu vezu koja je ostvarena preko polja *ReportTo*. Ova veza ukazuje vezu sa roditeljskim objektom koji označava zaposlenog koji je u nekoj strukturi iznad. Isto tako, ako se preko svojstva ostvaruje veza do više zaposlenih onda je tekući objekat nadređen grupi zaposlenih.

Važna preporuka je da se uradi promena imena ovih svojstava i na taj način olakša upotreba, pogledajte primer.



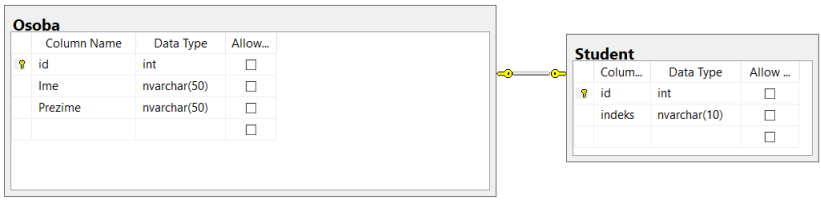
Slika 5.5. Promenjena navigaciona svojstva

```
using (var context = new NorthwindEntities1())
{
    DbSet<Employees> employees = context.Employees;
    foreach (var employee in employees)
    {
        Employees sef = employee.Chief;
        int brojPodredjenih = employee.Team.Count();
    }
}
```

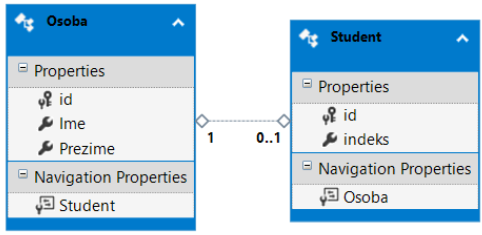
Spajanje tabela

Ukoliko se sadržaj neke tabele proširi drugom tabelom i relacijom „jedan na jedan“, moguće je uraditi sažimanje ovih tabela preko EDM modela. Razmotrimo primer povezivanja tabela *Osoba* i *Student* pri čemu su

povezani preko primarnog ključa. Odgovarajuća šema i EDM model nakon generisanja su prikazani na naredne dve slike



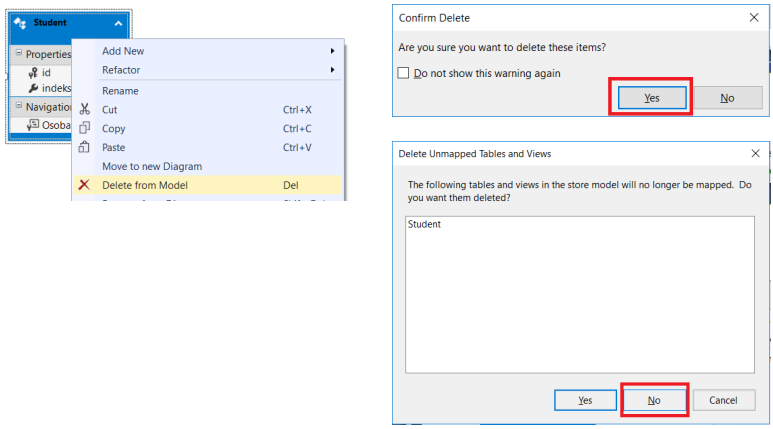
Slika 5.6. Relacija „jedan na jedan“



Slika 5.7. Entiteti i asocijacije u modelu nakon importovanja modela

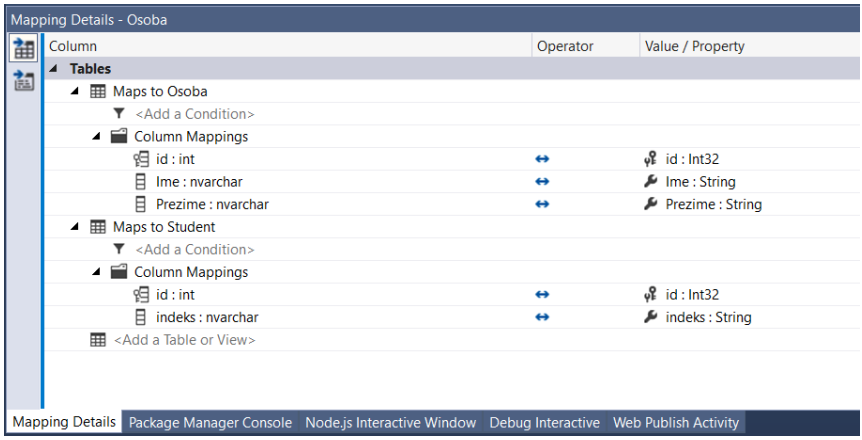
Postupak spajanja tabela:

- 1.Kopirati *indeks* iz tabele *Student* u *Osoba*
- 2.Desni klik na *Student*, izabrati brisanje, nakon što se pojavi dijalog za brisanje entiteta. Obavezno **odabrati No, čime se čuva definicija u Store modelu.**
  Dakle, naš model koji je zadužena za neposredni rad sa bazom podataka neće biti promenjen i preko njega će biti moguće ažurirati polje u tabeli *Student*. Ono što menjamo je konceptni model i naravno mapiranje. Na slici su prikazane slike koje ilustruju ovaj korak.



Slika 5.8. Izbacivanje svojstva iz entiteta ali ne i brisanje iz *Store* modela

- 3.Otvoriti prozor za editovanje mapiranja na entitetu *Osoba*. Odabrati dodatno *Student* tabelu za mapiranje



Slika 5.9. Dodavanje mapiranja za novu promenu modela

Nakon promene modela u kodu koristimo samo jedan entitet, na primer:

```
using (EFknjigaEntities ctx = new EFknjigaEntities())
{
    var o1 = new Osoba { id = 150, Ime = "Mita", Prezime =
"Jovic", indeks="HPT-1/22" };
    ctx.Osoba.Add(o1);
```

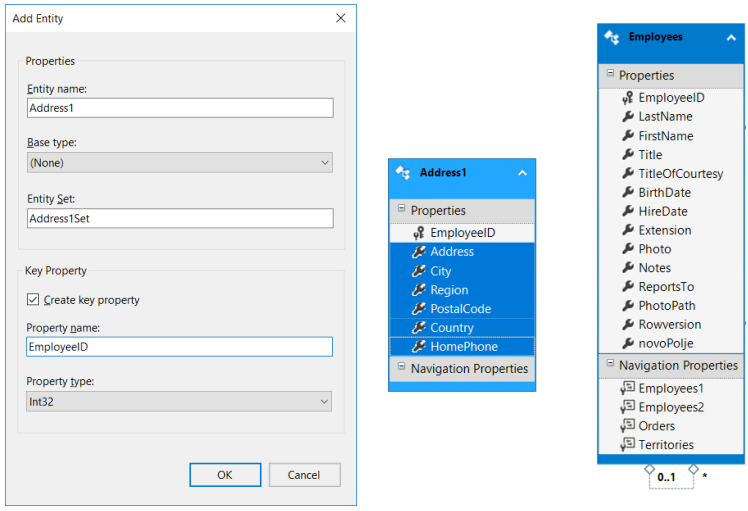
```
    ctx.SaveChanges();
}
```

Podela jednog entiteta

Ako imamo u modelu entitet koji odgovara tabeli sa nekoliko često korišćenih polja i sa nekoliko velikih, ali retko potrebnih polja, onda se preporučuje podelia na više entiteta. Pogledajmo entitet *Employees* i polja: *Address*, *City*, *Region*, *PostalCode*, *Country* i *HomePhone*.

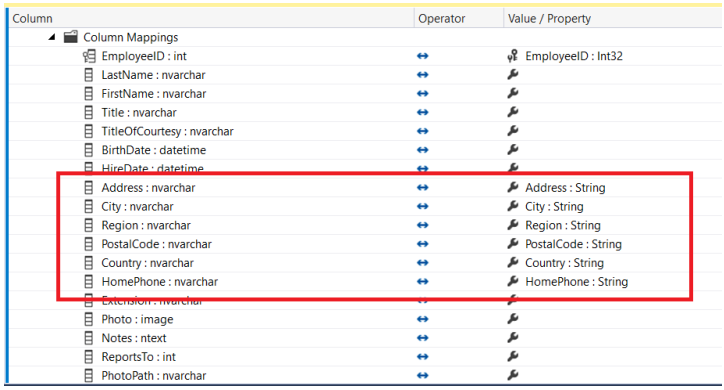
1. Dodaje se novi entitet sa ključem i poljem koje se dobija kopiranjem iz postojeće tabele.

2. Prebaciti navedena polja u novi entitet.



Slika 5.10. Izdvajanje polja u poseban entitet

3. Zatim uradimo mapiranje polja entiteta na tabelu Employees.

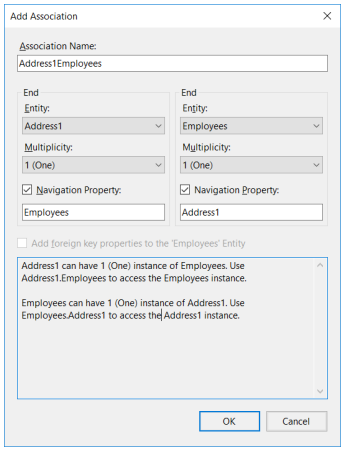


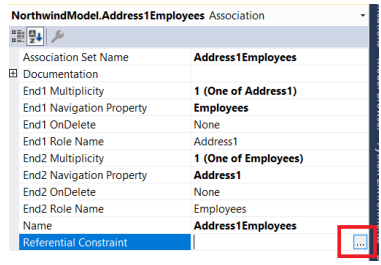
Slika 5.11. Dodavanje mapiranja na novi entitet

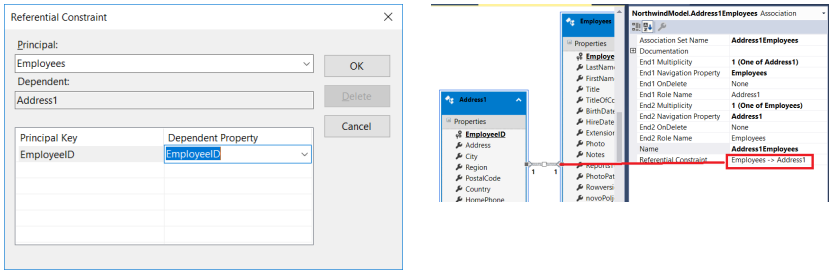
4. Dodajemo asocijaciju između dva entiteta.

5. Zatim se podesi svojstvo nove asocijacije, desnim klikom na nju, pa izborom opcije *Referential Constraint.*

Na narednim slikama je prikazano nekoliko pogleda pri kreiranju ove asocijacija.







Slika 5.12. Izdvajanje polja u novi entitet

Na kraju, pogledajmo šta smo dobili sa kreiranjem novog entiteta. Rad sa podacima adrese sada se svodi na rad sa posebnim entitetom.

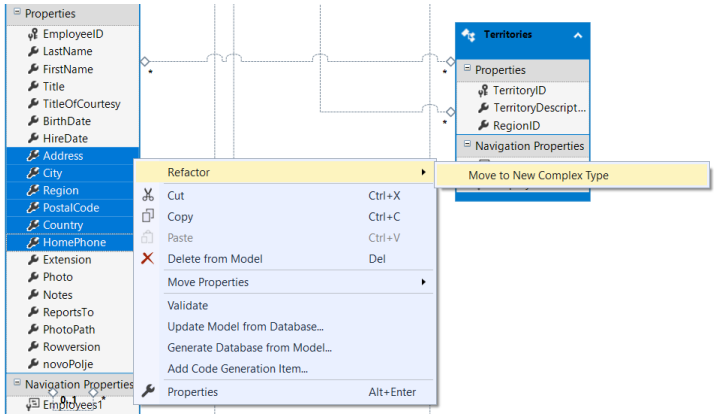
```
using (var ctx = new NorthwindEntities())
{
    var zap1 = ctx.Employees.First();
    Address1 adr1 = zap1.Address1;
    adr1.Country = "Serbia";
    ctx.SaveChanges();
}
```

Kreiranje kompleksnih tipova

Ukoliko se grupa polja ponavlja u više entiteta možete da razmislite o formiranju kompleksnih tipova. Kompleksni tip liči na izdvojeni entitet tj. na prethodni primer, ali svakako nije isto. U našem primeru *Northwind* baze možemo da primenimo kompleksni tip na istu grupu polja koje označavaju adresu. Postupak je sledeći:

1.Selektuje se grupa polja jednog entiteta za koju treba da formiramo kompleksni tip.

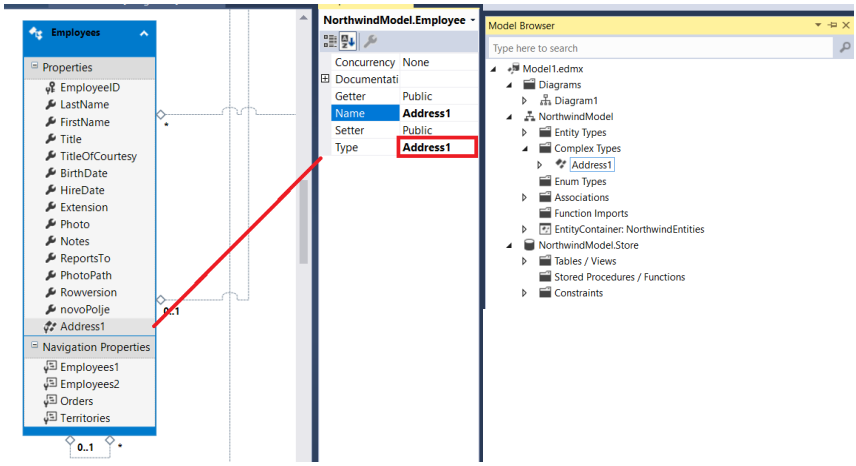
2.Desnim klikom, dok je grupa selektovana, otvara se konteksni meni na mom biramo opciju *Refactor*, a zatim *Move to New Complex Type*, pogledajte sliku.



Slika 5.13. Izdvajanje grupe polja za kompleksni tip

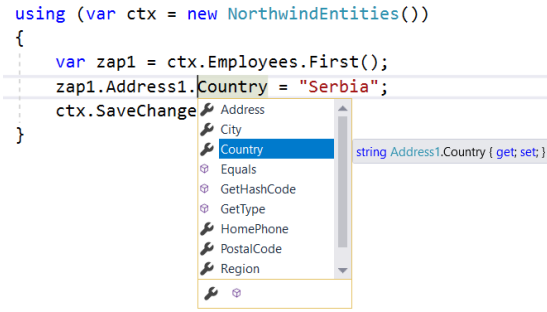
Nakon formiranja kompleksnog tipa, grupa polja u entitetu se sažima u jedno polje čiji tip je označen novim imenom.

Istovremeno, u prozoru *Model Browser*, može se videti novi tip u grupi *Complex Types.*



Slika 5.14. Formiranje kompleksnog tipa

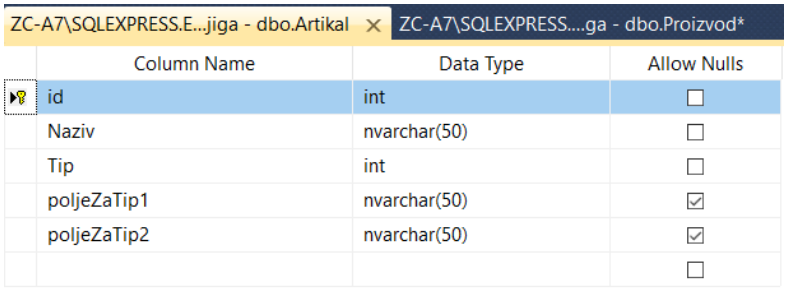
Novi kompleksni tip podataka omogućava sažeto prikazivanje grupe podataka. Ovako skraćeno zapisivanje može se koristiti u svim vrstama operacija, a upotrebu prati i *IntelliSense* u IDE okruženju.



Slika 5.15. Rad sa kompleksnim tipovima

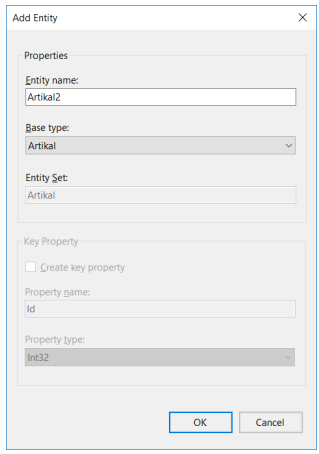
Diskriminativna polja

Nekada tabela poseduje polja koja mogu da se iskoriste za razdvajanje podataka u nove grupe. Pogledajmo jedan takav primer entiteta *Artikal* koji ima dva polja *poljeZaTip1* odnosno *poljeZaTip2* čije vrednosti mogu biti postojati ili ne, *Allow Nulls* je označeno. Vrednosti ovih polja su definisana zavisno od vrednosti polja *Tip*.



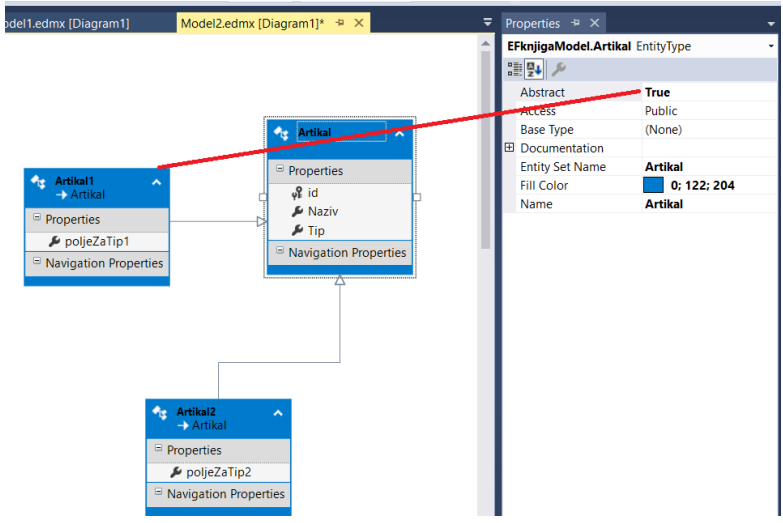
Slika 5.16. Prikaz tabele Artikal koji ćemo razdvojiti u više entiteta

1.Dodajemo dva nova entiteta u model. Neka su to entiteti: *Proizvod1* odnosno *Proizvod2*, koji će odgovarati vrednostima dva polja, oba izvedena iz entiteta *Proizvod*.



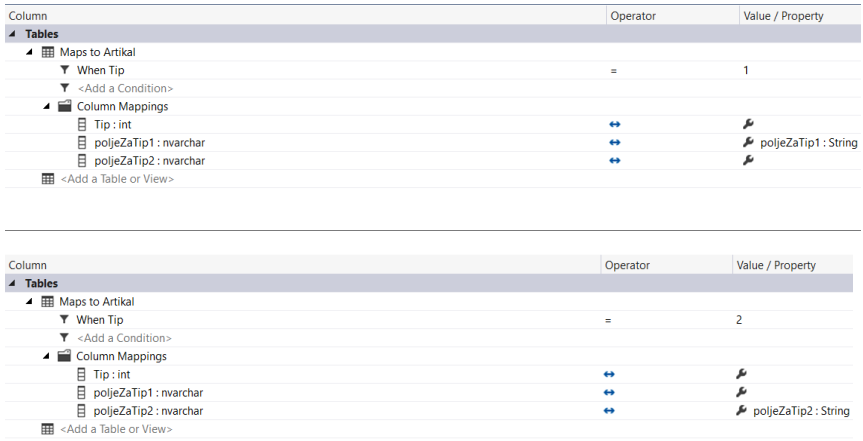
Slika 5.17. Dodavanje novog izvedenog entiteta

2.Koristeći opciju *Cut/Paste* prebaciti *poljeZaTip1* iz *Proizvod* u *Proizvod1*, a *poljeZaTip2* iz *Proizvod* u *Proizvod2*.



Slika 5.18. Šema izvedenih entiteta i njihove veze

3.Mapirati prebačene kolone u novim entitetima u odgovarajuće u tabelama, ali koristiti uslov na osnovu polja *EmployeeType*.



Slika 5.19. Prikaz mapiranja polja u novim entitetima

4.Izmeniti svojstvo **Abstract** entiteta *Proizvod* u **true**.

5.Izbrisati *Tip* iz *Proizvod* entiteta.

Umesto jednog entiteta sa *Tip* poljem i dva polja koja se koriste u zavisnosti od *Tip*-a imamo dva odvojena entiteta sa kojima radimo na standardan način. Na primer:

```
using (var ctx = new EFknjigaEntities())
{
    var a1 = new Artikal1 { id=1, Naziv = "pr1", poljeZaTip1
= "tipicno za 1." };
    var a2 = new Artikal2 { id=2, Naziv = "pr2", poljeZaTip2
= "tipicno za 2." };
    ctx.Artikal.Add(a1);
    ctx.Artikal.Add(a2);

    ctx.SaveChanges();
}
```

Eksplicitno izvršavanje sql upita

Primena EDM modela znači primenu i rad sa objektima tj. entitetima modela. Ipak, treba znati da je moguće i eksplicitno zadavanje sql komandi. Evo kako to izgleda u slučaju jedne parametrizovane komande.

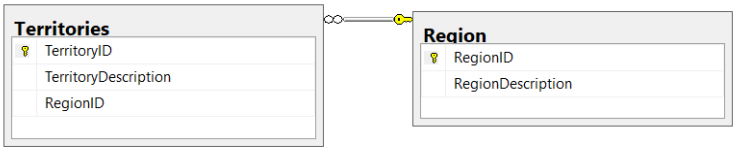
```
using (var ctx = new EFknjigaEntities())
{
    var sql = @"insert into Artikal(id, Naziv,Tip) values
(@id, @naziv, @tip)";
    var parameters = new DbParameter[]
    {
        new SqlParameter {ParameterName = "id", Value = 3},
        new SqlParameter {ParameterName = "naziv", Value =
"pr3"},
        new SqlParameter {ParameterName = "tip", Value = "1"}
    };

    var rowCount = ctx.Database.ExecuteSqlCommand(sql,
parameters);


}
```

Rezultat sql upita može biti bilo koji slog. Ipak, ako je slog koji se vraća odgovarajući slog nekog entiteta, onda se rezultat može prihvatiti kao tip tog entiteta. Sledi primer za slučaj kada se vraća rezultati tipa `Osoba`.

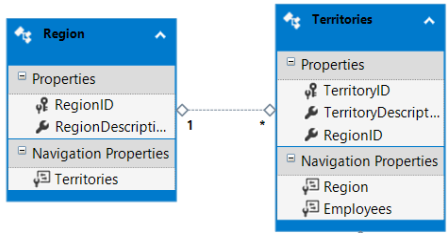
```
using (var ctx = new EFknjigaEntities())

{
    var sql = "select * from Osoba where ime = @ime";
    var parameters = new DbParameter[1];
    parameters[0] =  new SqlParameter { ParameterName =
"ime", Value = "Mita" };
    var osobe = ctx.Database.SqlQuery<Osoba>(sql,
parameters);
    foreach(var o in osobe)
    {
        var punoIme = o.Ime + " " + o.Prezime;
    }
}
```

Svi zapisi za „jedan na više“

Asocijacija „jedan na više“ je tipična veza entiteta u EDM modelu koja odgovara tabelama u bazi koje su povezane vezom „jedan na više“. Razmotrimo konkretan primer sa slike i rad sa povezanim podacima:



Slika 5.20. Veza „jedan na više“ u bazi



Slika 5.21. Veza „jedan na više“ u modelu

Sve zapise iz obe tabele moguće je dobiti na više načina. Recimo da želimo da dobijemo vrednost `RegionDescription` iz tabele `Region` odnosno `TerritoryDescription` iz tabele `Territory za RegionID==1`.

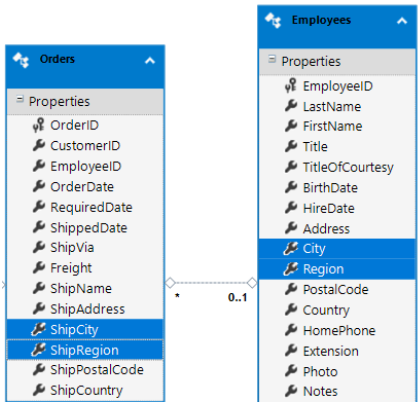
```
class myRegTer
{
    public string rdesc;
    public string tdesc;

}

using (var ctx = new NorthwindEntities())
{
    var q =
ctx.Territories.Include("Region").Where(x=>x.RegionID==1);
    var rez0 = from el in q
               select new myRegTer
               {
                   rdesc = el.Region.RegionDescription,
                   tdesc = el.TerritoryDescription
```

```
                };

    var rez1 = from t in ctx.Territories
               join r in ctx.Region
               on t.RegionID equals r.RegionID
               where r.RegionID==1
               select new myRegTer
               {
                   rdesc = r.RegionDescription,
                   tdesc = t.TerritoryDescription
               };

    var rez2 = from r in ctx.Region
               from t in r.Territories
               where r.RegionID==1
               orderby r.RegionID
               select new myRegTer
               {
                   rdesc = r.RegionDescription,
                   tdesc = t.TerritoryDescription
               };

    int cnt0 = rez0.Count();
    int cnt1 = rez1.Count();
    int cnt2 = rez2.Count();
}
```

Složeni *join* po više kolona

Dve tabele mogu biti povezani međusobno preko više kolona ili naš upit može biti zasnovan na povezivanju preko više kolona. Ovaj slučaj podseća na složeni *Join* upit koji se zasniva na povezivanju više vrednosti dve tabele.

U sledećem primeru razmotrićemo povezivanje dva entiteta: `Orders` odnosno `Employees` koristeći odgovarajuća polja: `ShipCity`, `ShipRegion` odnosno `City`, `Region`. Šema entiteta data je na narednoj slici, a primer izdvajanja rezultata nakon slike.



Slika 5.22. Šema povezanih entiteta

```
using (var ctx = new NorthwindEntities())
{
    var specOr = from or in ctx.Orders
                 join em in ctx.Employees on
                 new {
                      City = or.ShipCity,
                      State = or.ShipCountry }
                 equals
                 new { City = em.City, State = em.Country }
                 select or;
}
```

Grupisanje po više svojstava

Ako vršimo grupisanje po više svojstava nekog entiteta, onda ne možemo jednostavno izjednačiti vrednosti kao kada grupišemo po jednom svojstvu. Umesto toga, formira se novi anonimni objekat na osnovu koga se vrši grupisanje. Ovaj postupak liči na prethodni primer. Pogledajmo Linq upit koji bi grupisao zaposlene iz tabele `Emplyees` po osnovu dva polja: `City` i `Country`, a rezultat je kolekcija koja sadrži: `City`, `Country` ali listu svih koji pripadaju istom gradu i zemlji – `details`.

```
var results = from em in ctx.Employees
              group em by new { em.City, em.Country } into g
              select new
              {
                  Country = g.Key.Country,
```

```
                City = g.Key.City,
                details = g
            };
foreach (var r in results)
{
    string city = r.City;
    string state = r.Country;
    foreach (var d in r.details)
    {
        string ime = d.FirstName;
        string prezime = d.LastName;
    }
}
```

Pitanja i zadaci za proveru znanja

1. Napišite kod za dodavanje jednog više `CustomerDemographics` zapisa za jedan `Customer`.
2. Napišite kod za prikaz svih zaposlenih (zapisi u `Employees` tabeli) koji su rukovodioci i listu pripadajući zaposlenih u njegovom timu.
3. Kako se vrši spajanje više tabela? Kreirajte tabele u bazi i napišite jedan sopstveni primer koji ilustruje ovaj postupak.
4. Kreirajte novu tabelu `Student` sa poljima: `indeks`, `status`, `datumDiplomiranja`, `godinaStudija`. Ako vrednost poja `status` 1 student studira i ima `godinuStudija`. Ako je `status` jednak 2 onda ima `datumDiplomiranja`, a ako je 0 onda nema `status`. Pokušajte da ovakvu tabelu razdvojite na odgovarajuće entitete u modelu i napišete primere za rad sa podacima.
5. Napravite kompleksni tip za Adresu u svim tabelama Northwind modela.

6. MVC aplikacije

U ovom poglavlju najpre ćemo dati opis arhitekture koja je jedna od osnovnih u projektovanju veb aplikacija, a zatim ćemo uraditi postupak kreiranja i testiranja prve veb aplikacije. Na početku je dat opšti opis MVC arhitekture kao i prednosti u projektovanju veb aplikacija. Nakon opšteg opisa i principa koji su bazični u različitim primenama, poglavlje se bavi formiranjem ovakvih projekata pomoću alata Visual Studio.

ASP MVC je jedan šablon u razvojnom okviru Microsoft tehnologija za veb aplikacije. ASP MVC kombinuje karakteristike MVC arhitekture, najsavremenije ideje i tehnike iz *Agile* razvoja i najbolje delove postojeće ASP.NET platforme. ASP MVC nije nova tehnologija. To je potpuna alternativa za tradicionalne ASP.NET Veb Form aplikacije. Na steku .NET klasa ASP.NET je na vrhu tako da programeri imaju na raspolaganju gotovo sve .NET funkcije prilikom izgradnje MVC aplikacija.

MVC arhitektura

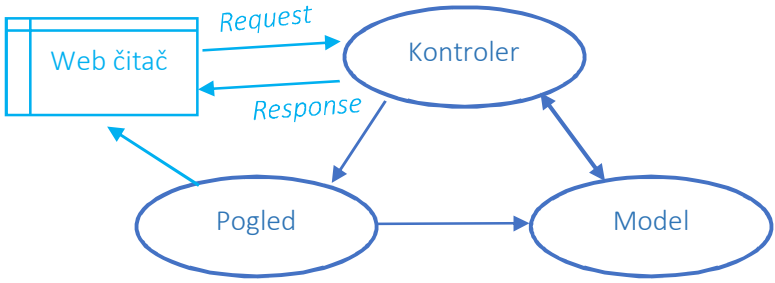
MVC označava skraćenicu engleskih reči za model (eng. **M**odel), pogled (eng. **V**iew) i kontroler (eng. **C**ontroler). Ove tri komponente kao i njihova povezanost jesu osnova MVC arhitekture na kojoj se bazira razvoj veb aplikacija u nastavku.

Model – predstavlja programsku reprezentaciju podataka i poslovne logike. Model čuva podatke u aplikaciji. Pomoću modela podaci se dobijaju iz baze, menjaju, a zatim ponovo snimaju u bazi podataka.

Pogled – predstavlja prikaz u aplikaciji. Pomoću pogleda korisniku se podaci predstavljaju, ali i pomoću pogleda korisnik može da menja podatke. Pogled je korisnički interfejs.

Kontroler – programska komponenta koja upravlja korisničkim zahtevima. Korisnik posredstvom komponenata za pogled upućuje serveru odgovarajući URL zahtev. Ovaj zahtev se obrađuje u kontroleru. Kontroler inicira novi pogled sa odgovarajućim podacima iz modela kao odgovor na zahtev korisnika.

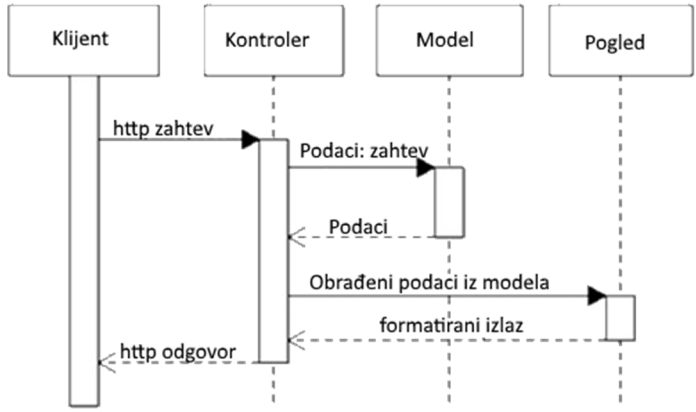
Šematski prikaz veze povezanosti ovih objekata data je na slici.



Slika 6.1. Šematski prikaz MVC arhitekture

Primer jedne obrade zahteva

Korisnik unosom u adresni deo veb čitača, kreira URL zahtev. Taj zahtev se prihvata na strani servera i to radi komponenta kontroler. Kontroler na osnovu vrste zahteva kreira podatke i odgovarajući pogled koji vraća kao odgovor korisniku.



Slika 6.2. Dijagram sekvence za MVC komponenata

Prednosti MVC arhitekture

MVC arhitektura nudi nekoliko prednosti:

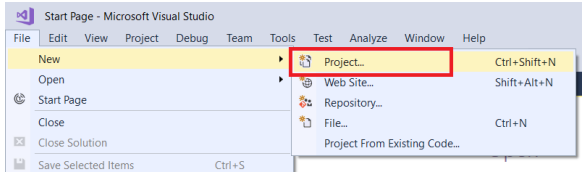
- olakšava upravljanje složenim aplikacijama uvođenjem slojeva odnosno deljenjem aplikacije na komponente,
- omogućava potpunu kontrolu nad prikazanim HTML-om i pruža jasno razdvajanje nadležnosti,
- potpuna i direktna kontrola nad generisanim HTML stranicama znači bolje mogućnosti u primeni i usklađenost sa standardima,
- omogućava više interaktivnosti i prilagođavanje za postojeće aplikacije,
- ima bolju podršku za razvoj upravljan testovima (TDD).

Praksa je pokazala veliku popularnost MVC arhitekture u primeni. Pokazuje se da je često koriste timovi programera i dizajnera kojima je potreban visok stepen kontrole u aplikaciji, modularnost, odvojenost testiranja.

Kreiranje projekta

Kreiranje prve veb aplikacije zasnovane na MVC arhitekturi uz pomoć razvojnog alata Visual Studio vršimo u svega nekoliko koraka.

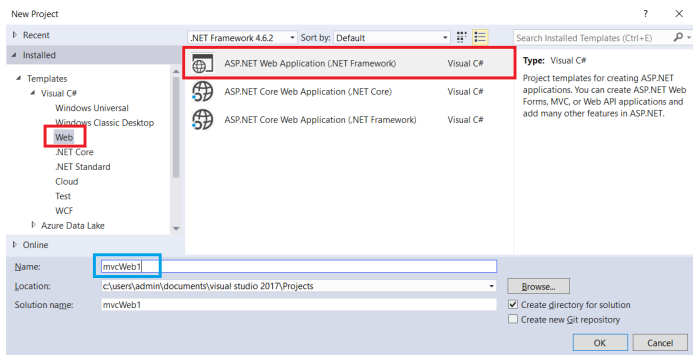
Korak 1. Otvorite Visual Studio. Odaberite opciju: *File -> New -> Project*.



Slika 6.3. Kreiranje projekta

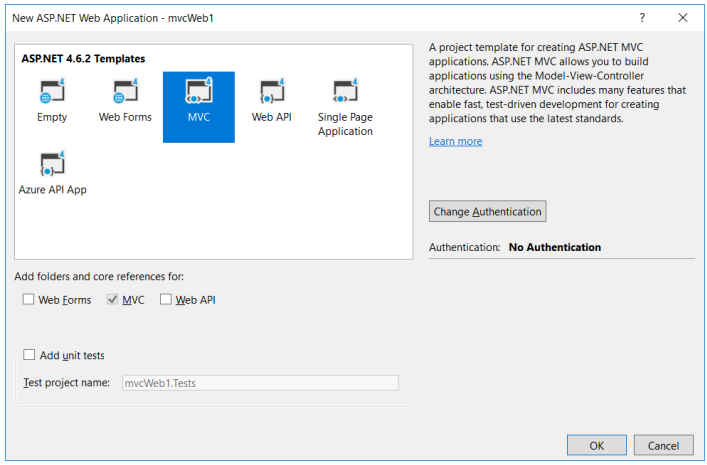
Korak 2. U prozoru *New Project* u levom panelu odaberite opciju: *Templates-> Visual C# -> Web*, pogledajte sliku.

U desnom prozoru selektujte *ASP.NET Veb Application*.



Slika 6.4. Izbor šablona projekta: ASP.NET

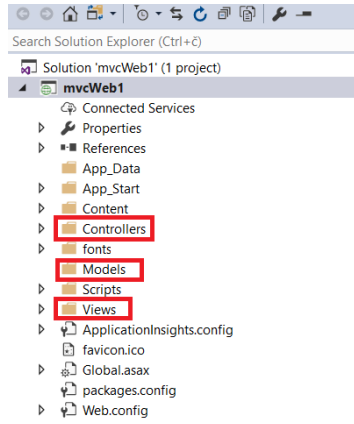
Korak 3. U ovom koraku definiše se šablon veb aplikacije. Biramo MVC šablon, kao na slici.



Slika 6.5. Izbor MVC šablona

Nakon ovog koraka Visual Studio automatski kreira MVC veb aplikaciju.

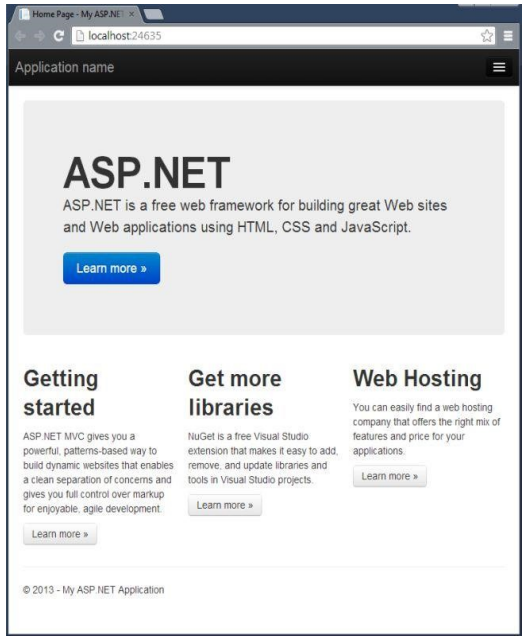
Visual Studio IDE (eng. Integrated Development Environment) nudi više pogleda na projekat pomoću posrednih prozora. Tipičan pogled je preko *Solution Explorer* prozora, gde možete videti kreirane fajlove i foldere u okviru aplikacije. Tu ćete naći i foldere: **Models**, **Controllers** i **Views**.



Slika 6.6. Osnovna struktura projekta

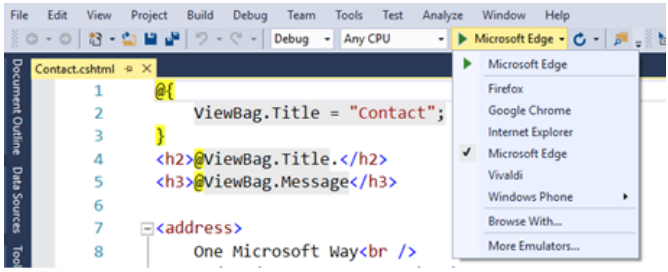
Dobijena mini aplikacija je ujedno šablon za dalje izmene i programiranje.

Važno: Visual Studio IDE nudi lak način za testiranje kreirane veb aplikacije, bez potrebe da ceo projekat ili fajlove prebacujete na veb server. Zahvaljujući ugrađenom IIS (Internet Information Services) veb serveru koji Visual Studio koristi, veb aplikacija se pokreće koristeći taj server. Pritiskom na F5 vrši se prevođenje koda, a zatim i učitavanje početne stranica na veb server odnosno pokretanje aplikacije na nekom od veb čitača, kao na slici.



Slika 6.7. Početna stranica aplikacije

Obratite pažnju da Visual Studio automatski otvara neki veb čitač da bi prikazao projekat. Kao podrazumevani veb čitač se može izabrati bilo koji koji ste instalirali na računaru, koristeći traku sa alatkama, kao što je prikazano na slici:



Slika 6.8. Izbor veb čitača za testiranje

Obično se pri proveri ispravnosti koristi više različitih čitača. Na slici se vidi da je instalirano nekoliko najpoznatijih veb čitača, što može biti korisno za testiranje veb aplikacija tokom razvoja.

Napomena: U polju adrese veb čitača nalazi se URL i broj porta aplikacije koja se hostuje. *Localhost* znači da je pokrenuta aplikacija na lokalnom računaru. Broj porta je generisan od strane okruženja Visual Studio i koristi se pri startu.

Struktura foldera

Kreirani projekat sastoji se od foldera organizovanih tako da sadrže fajlove određene namene. U projektu su sledeći folderi:

App_Data – Obično sadrži podatke aplikacije, kao što su LocalDB, .mdf, .xml fajlovi koji čuvaju podatke od značaja za rad aplikacije.

App_Start – Sadrži klase koje se koriste pri startu aplikacije. Tipično su tu fajlovi: BundleConfig.cs, FilterConfig.cs, RouteConfig.cs.

Content - Ovde su smešteni fajlovi stilova, slika, ikona. Od verzije MVC 5 kao osnovni stil aplikacije koristi se Bootstrap pa su ovde uključeni fajlovi koji omogućavaju primenu ovog radnog okvira namenjenog za lako i

efikasno kreiranje prilagođenih stranica: bootstrap.css, bootstrap.min.css i site.css.

Controllers - Sadrži `cs` fajlove koje sadrže klase kontrolera. Kontroleri upravljaju korisničkim zahtevima i vraćaju odgovarajući odziv na zahtev korisnika. MVC projekat zahteva da se naziv svih fajlova i klasa kontrolera završava sa `Controller`.

fonts – Sadrži specifične fontove za aplikaciju.

Models – Sadrži sve klase modela tj. sadrži klase koje opisuju podatke u aplikaciji. Najčešće se modeli sastoje od javnih svojstava preko kojih se dobavljaju odnosno menjaju podaci.

Scripts – Čuva JavaScript fajlove za rad aplikacije. MVC 5 u ovom folderu smešta JS skripte u okviru Bootstrap radnog okvira, *jQuery* i *modernizer* biblioteke.

Views – Sadrži sve html fajlove aplikacije. Najčešće se na ovom mestu čuvaju fajlovi ekstenzije .cshtml koji predstavljaju komponente pogleda. Pogledi imaju posebne foldere za svaki kontroler. Takođe u ovom folderu je i Shared folder. U njemu su svi pogledi koji se mogu deliti između svih kontrolera.

HTML stranice

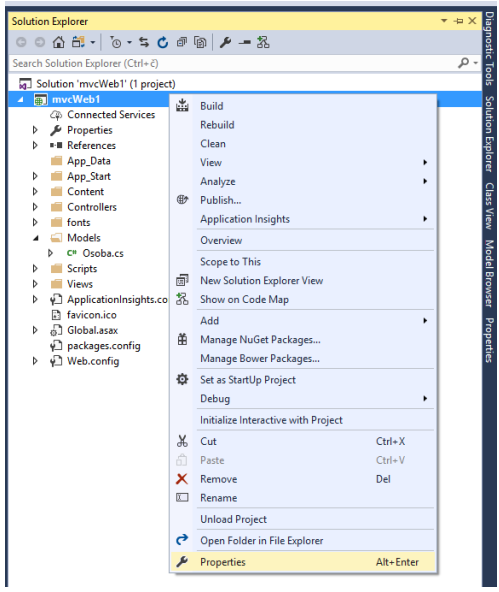
Pokušajte da nađete gde se nalaze HTML stranice u projektu?

Odgovor na ovo pitanje pokazuje tehniku rada u ovom razvojnom okruženju. Umesto fiksnih HTML stranica nalaze se fajlovi ekstenzije cshtml, standardni cs fajlovi i naravno biblioteka JavaScript funkcija i CSS stilova koju ćemo koristiti za izradu stranica prilagođenih različitim uređajima Bootstrap.

Ako se pitate šta je sa fiksnim HTML stranicama ako je sve podređeno dinamičkim sadržajima tj. da li je moguće koristiti fiksni HTML u projektu? Naravno da jeste moguće koristiti potpuno fiksne unapred pripremljene HTML stranice. Međutim, čak i za takve stranice, važno je da se fiksni sadržaj može lako uklopiti u osnovni šablon sajta. To, znači da zaglavlje, navigacija, futer i eventualno bočni elementi obično ostaju nepromenljivi, a centralni deo se menja, bilo dinamičkim ili fiksnim sadržajem.

Podešavanja

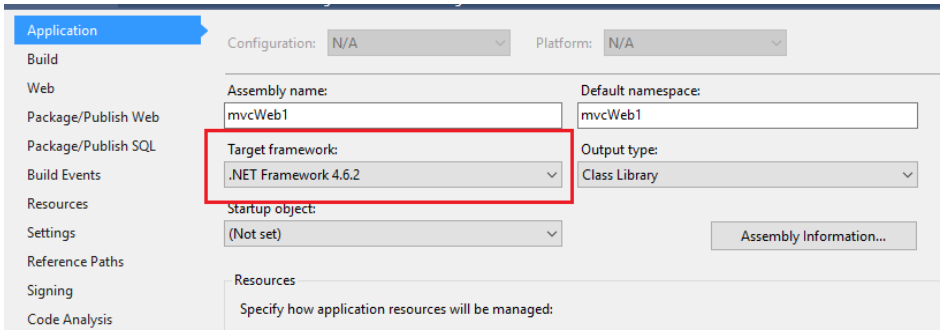
Podešavanja na nivou projekta se obavljaju tako što se desnim klikom na projekat u *Solution Explorer* prozoru izabere opcija *Properties*.



Slika 6.9. Izbor opcija za podešavanje

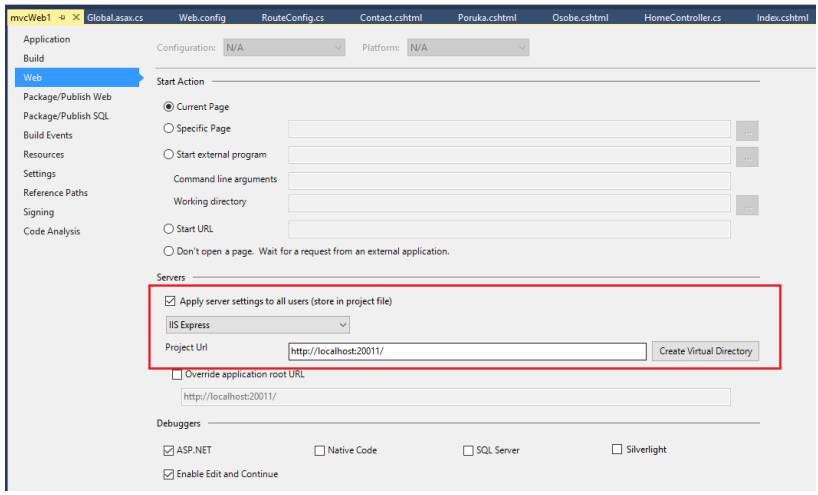
U ovom delu se nalaze sva neophodna podešavanja za vašu aplikaciju. Pogledajmo nekoliko tipičnih svojstava u okviru ovih podešavanja.

Na kartici *Application* možete naći korišćeni .NET Framework.



Slika 6.10. Kartica Application u delu za podešavanja

Na kartici *Web* možete odabrati način podešavanja korišćenog internog servera kao i važeći URL vaše aplikacije za potrebe razvoja odnosno testiranja.



Slika 6.10. Kartica *Web* u delu za podešavanja

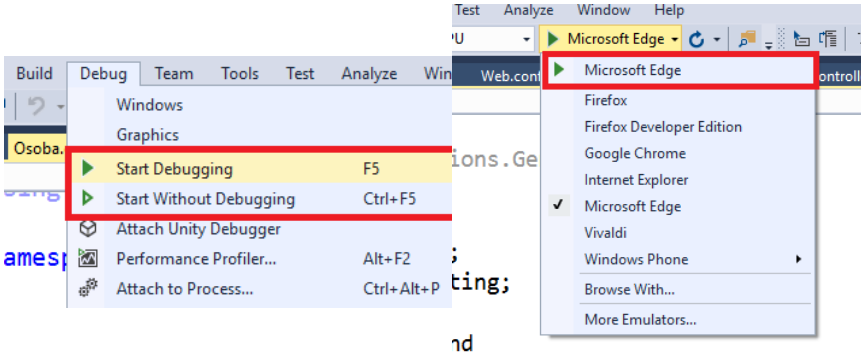
Razvojni server

Visual Studio uključuje IIS Express, lokalnu razvojnu verziju veb servera, koja se koristi za pokretanje veb stranica. Pri tome se za potrebe testa koristi slobodni "port" tj. broj koji zajedno za IP adrsom definiše URL adresu aplikacije. Na slici 6.7, vidi se da aplikacija radi na adresi http://localhost:24635/, tj. da koristi port 24635. Broj porta je definisan podešavanjima, a može biti naknadno promenjen.

U našim primerima, kada govorimo o adresama tj. URL-ovima kao što su /Student ili /Home radi se o dodatku na osnovni URL aplikacije, tj. na deo koji ide nakon broja porta. Pretpostavljajući da je broj porta 24635, adrese: /Student ili /Home znače zapravo pune adrese: http://localhost:26641/Student ili http://localhost:26641/Home.

Testiranje

Videli smo kako se pokreće aplikacija koja se uređuje preko Visual Studio-a, i to na dva načina. Izborom skraćenice F5 pokreće se projekat uz debagovanje. F5 inicira Visual Studio da pokrene *IIS Express* i startuje veb aplikaciju na tom serveru. Visual Studio zatim pokreće neki veb čitač i otvara početnu stranicu kreiranog projekta. U konkretnom slučaju to je stranica *Index* kontrolera *Home*.

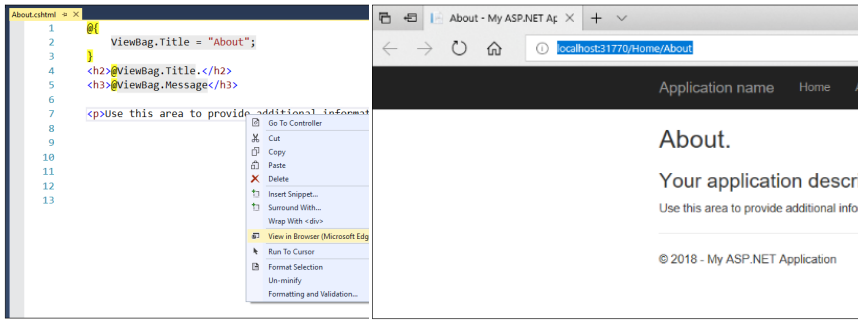


Slika 6.11. Dva načina pokretanja projekta

Međutim, otvaranje pogleda koji se trenutno uređuje u editoru Visual Studio-a može se jednostavnije obaviti preko kontekstnog menija.

Na ovaj način otvara se URL adresa pogleda za određeni kontroler. U našem slučaju Home je kontroler, a metoda *About* ima svoj pogled *About.chtml,* pa se na lokalnom IIS serveru za testiranje otvara stranica: http://localhost:31770/Home/About.

Na slici je prikazan način otvaranja pogleda i sam pogled. Vidi se da se i bez eksplicitnog prevođenja vrši pozivanje lokalnog servera koji otvara pogled od interesa.



Slika 6.12. Otvaranje pogleda u veb čitaču

Pitanja i zadaci za proveru znanja

1. Koje komponente čine MVC arhitekturu? Objasni njihovu ulogu.
2. Napravite jednu veb aplikaciju MVC arhitekture. Opiši strukturu foldera i odgovori u kojim folderima se nalaze osnovne komponente?
3. Gde se čuvaju JavaScript, CSS odnosno Bootstrap fajlovi?
4. Kako se vrši podešavanje u projektu?
5. Koja podešavanja možete da koristite? Pokušajte da nađete gde se nalazi zapis ovih podešavanja.
6. Kako se obavlja testiranje projekta?
7. Testirajte početni projekat u različitim veb čitačima.

7. Kontroleri

U ovom poglavlju razmatramo komponentu kontroler u okviru MVC arhitekture i njenu konkretnu realizaciju u veb aplikaciji.

Klase kontrolera smeštene su u direktorijumu ***Controllers***. U ovom folderu kontroleri se nalaze u `cs` fajlovima. Po pravilu svaki kontroler je u posebnom folderu. MVC arhitektura zahteva poštovanje naziva fajlova odnosno klasa kontrolera, više o tome u nastavku.

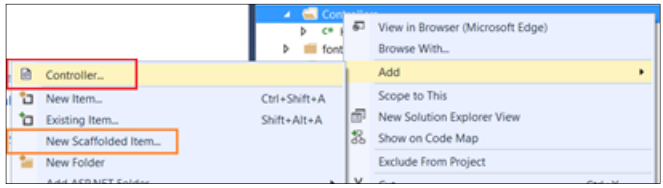
Kontroleri upravljaju korisničkim zahtevima i vraćaju odgovarajući odgovor. Dakle, to su objekti odgovorni za prihvatanje korisničkih zahteva i iniciranje novih prikaza. Takođe, preko kontrolera se vrše izmene u modelima podataka kao odgovor na korisnički unos. Oni su odgovorni za tok aplikacije i rad sa unetim podacima i podacima koji se prikazuju. Kao konačan ishod korisničke akcije, kontroleri obezbeđuju odgovarajući pogled.

U zavisnosti od opcija koje su odabrane pomoću 'čarobnjaka' pri početnom kreiranju aplikacije, formirano je nekoliko kontrolera automatski. Otvaranjem foldera *Controllers* iz prozora *Solution Explorer* mogu se pogledati već napravljeni kontroleri.

Kreiranje

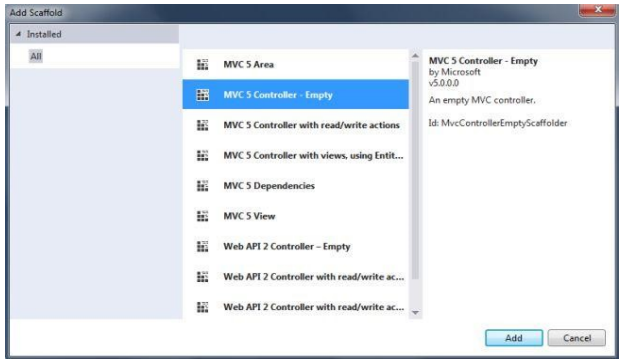
Postupak za kreiranje novog kontrolera je sledeći:

Korak 1: Desni klik na folder `Controllers` ili `New Scafolded Item`.



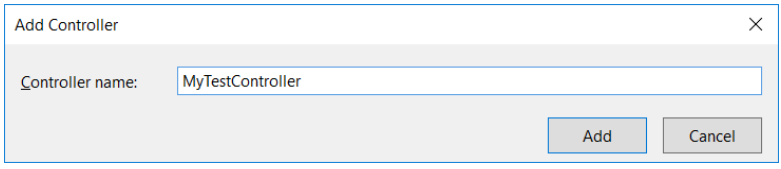
Slika 7.1. Opcija iz menija za kreiranje kontrolera

Korak2: Zatim se vrši izbor opcije kontrolera (prazan, sa opcijama čitanja i pisanja, sa pogledom) u formi `Add Scaffold`



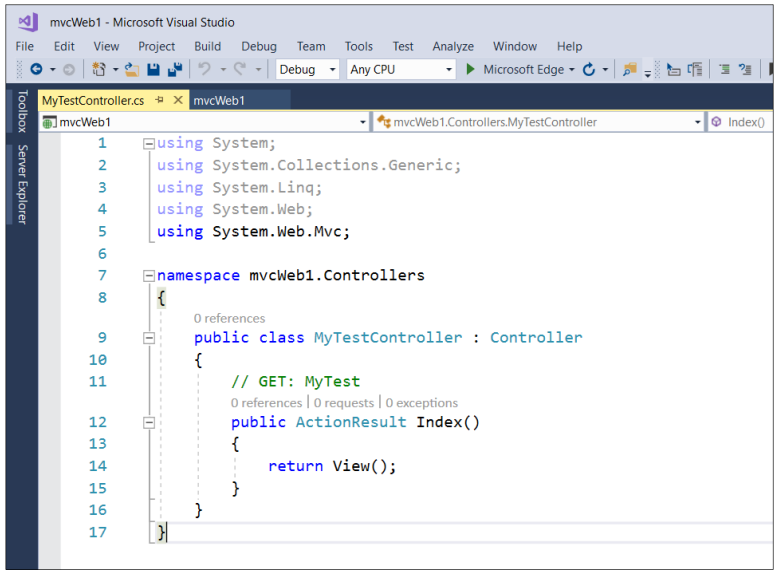
Slika 7.2. Opcija iz menija za kreiranje kontrolera

Korak 3: Zatim se otvara pomoćna forma u kojoj se mora uneti ime kontrolera. Na primer "MyTestController", kao na slici.



Slika 7.3. Izbor naziva kontrolera

Visual Studio kreira fajl `MyTestController.cs` i istovremeno prikazuje novi kontroler, kao na slici.



Slika 7.4. Novi kontroler

Kao što se vidi, novi kontroler je kreiran kao klasa izvedena iz klase `Controller`. Takođe, automatski je dodata jedna metoda po njegovom kreiranju. Ta metoda je `Index` i ona predstavlja podrazumevanu metodu kontrolora koja će se koristiti ako se eksplicitno ne navede metoda.

Testiranje

Novi kontroler možemo odmah i testirati. Ipak, uradićemo neke izmene koje će bolje ilustrovati rad kontrolera u ovom tipu aplikacija.

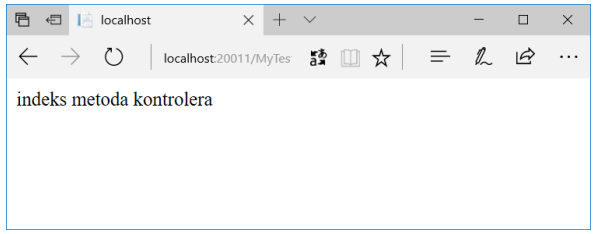
Promenite kod kontrolera koji je automatski kreiran na način kako je to u nastavu prikazano. Obratite pažnju na povratnu vrednost metoda.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace mvcWeb1.Controllers
{
    public class MyTestController : Controller
    {
        //public ActionResult Index(){
        //    return View();
        //}
        public string Index()
        {
            return "indeks metoda kontrolera";
        }
        public string Welcome()
        {
            return "Welcome metoda kontrolera";
        }
    }
}
```

Nove metode **Index**, **Welcome** vraćaju objekat koji je po tipu običan string, u odnosu na inicijalno generisanu metodu koja vraća objekat pogleda tipa **ActionResult**. Pogledajmo ishod ove izmene.

Adresiranje specifičnog kontrolera preko URL zahteva vrši se navodeći naziv kontrolera iza kose crte, na primer: *localhost:1234/MyTest*

Ovaj URL aktivira poziv kontroler **MyTestController**, tačnije njegovu metodu Index koja je podrazumevana pošto se nije eksplicitno zahtevala druga metoda. Odziv na ovaj URL zahtev je:

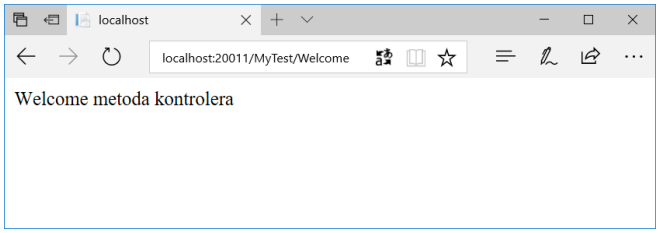


Slika 7.5. Odziv metode Index kontrolera **MyTest**

Na sličan način može se ostvariti poziv `Welcome` metode kontrolera na osnovu drugog URL zahteva:

`localhost:1234/MyTest/Welcome`

Odgovarajući odziv je:



Slika 7.6. Odziv metode Welcome kontrolera MyTest

Parametri u metodama

Metoda kontrolera može imati parametre koji se se prenose od korisnika preko URL zahteva u vidu parametara. Za prenos parametara koristi se sintaksa:

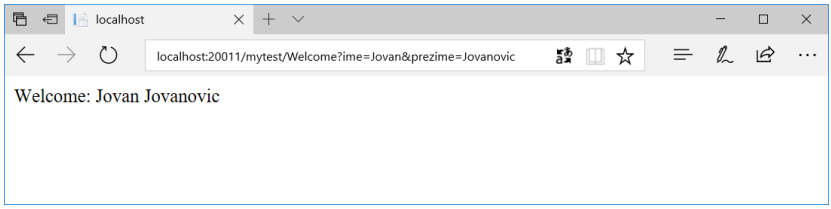
```
metoda?ime1=vrednost1&ime2=vrednost2
```

Na primer, ako promenimo metodu na sledeći način:

```
public string Welcome(string ime, string prezime)
{
    return "Welcome: " + ime + " " + prezime;
}
```

Odgovarajući URL zahtev za poziv ove metode odnosno prateći odgovor u veb čitaču, za port 20011, bio bi:

```
http://localhost:20011/mytest/Welcome?ime=Jovan&prezime=Jovanovic
```



Slika 7.7. Odziv metode Welcome sa pratećim parametrima

Zaštita od zlonamernih podataka (eng. *JS injecting*)

U praktičnoj primeni nije dobra praksa da se korisniku omogući upotreba stringa za aktivaciju nekog prikaza bez pomoćne konverzije u HTML kod. Razlog je bezbednosni. Umesto direktne upotrebe stringa obavezno se koristi pomoćna metodu `HttpUtility.HtmlEncode`. Ova metoda onemogućava *JavaScript injecting* ili *HTML markup* u prihvatni kod. Dakle:

```
public string Welcome(string ime, string prezime)
{
    return "Welcome: " + HttpUtility.HtmlEncode(ime) + " "
           + HttpUtility.HtmlEncode(prezime);
}
```

Adresiranje kontrolera

Kada se pokrene MVC aplikacija prva se poziva metoda `Applicaton_Start()`.

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);
    BundleConfig.RegisterBundles(BundleTable.Bundles);
}
```

U ovoj metodi poziv se metode `RegisterRoutes` koja kreira tabelu rutiranja za određene kontrolere. MVC poziva klase kontrolera i akcije koje mu pripadaju u zavisnosti od dolazećeg URL-a. Podrazumevana logika za URL rutiranja koristi sledeći format:

```
/[Controller]/[ActionName]/[Parameters]
```

Podešavanje formata rutiranja vrši se u pomenutoj metodi koja je definisana u klasi/fajlu `RouteConfig.cs` u folderu `App_Start`:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
id = UrlParameter.Optional }
);
```

Podrazumevana tabela rutiranja sadrži jednu rutu (naziva Default). Podrazumevana ruta mapira prvi segment URL-a u naziv kontrolera, drugi segment u nazivu akcije, a treći u parametar koji se naziva *id*.

Zamislimo da je uneta URL adresa preko polja za unos adrese nekog veb čitača:

```
/Home/Index/3
```

Mapa rutiranja ove adrese podrazumevana i to je:

- controller = Home
- action = Index
- id = 3

Na ovaj način kada se adresira `/Home/Index/3`, izvršavaće se metoda:

```
HomeController.Index(3)
```

Ako jedan URL ne sadrži ništa osim naziva domena onda je važeći podrazumevani kontroler odnosno metoda. Na primer, http://lcoalhost:1234 znači odgovor do klase `HomeController` odnosno metode `Index`.

Višestruke rute

Višestruke rute znači da se u objekat `routes` dodaju posebna mapiranja. Nekada je moguće i preklapanje između njih, odnosno istim adresiranjem izazvati iste akcije. Pogledajte sledeći primer:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

routes.MapRoute(
    name: "Student",
    url: "Student/{id}",
    defaults: new { controller = "Home", action =
"Student",
                    id = UrlParameter.Optional }
);
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional }
);
```

Za ovako podešeno rutiranje najpre se proverava da li URL započinje sa `Student`, pa tek ako ne, onda se ide na podrazumevano mapiranje. Ako jeste vrši se mapiranje na navedeni `Home` kontroler odnosno `Student` metodu.

Definisane rute mogu imati i definisana ograničenja u pogledu vrednosti parametara rute. Na primer, ako u ruti koristimo `id` parametar i ako je `id` uvek broj, onda se može postaviti takvo ograničenje, koristeći regularne izraze. U prethodnom primeru za `id` je navedeno da je opcioni parametar koristeći vrednost iz nabrojive liste `UrlParameter.Optional` kojom se to definiše.

Treba imati u vidu da je metoda `Index` podrazumevana metoda i da se eventualni parametar u toj metodi prosleđuje na očekivani način, čak i kada je opcioni. Na primer, ako je metod:

```
public ActionResult Index(int id)...
```

onda se uvek u URL adrese mora navesti argument koji je broj, odnosno:

`localhost:1234/Home?id=33` bio bi ispravan poziv tj adresiranje, dok bi

`localhost:1234/Home` ili `localhost:1234` bilo neispravno.

Šta se događa ako je argument metode `string id?`

Ako argument metode mora da bude ceo broj ali istovremeno može da ne postoji, onda bi se mogao koristiti sledeći potpis metode:

```
public ActionResult Index(int? id)...
```

Pitanja i zadaci za proveru znanja

1. Šta su kontroleri?
2. Kreirajte prazan kontroler `Products`?
3. Testirajte metodu `Index`.
4. Koji tip podataka vraća metoda kontrolera?
5. Kako se adresira određeni kontroler odnosno metoda kontrolera?
6. Kako se preko adrese zadaju parametri metode kontrolera?
7. Objasni primenu pomoćne metode `HttpUtility.HtmlEncode?`
8. Gde i kako se definiše način logika rutiranja poziva?
9. Kako se koriste višestruke rute?

8. Pogledi

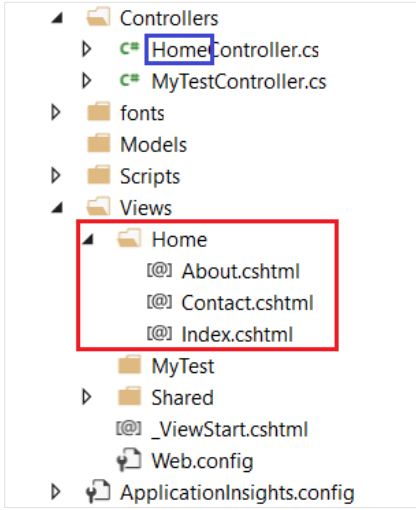
U prethodnom poglavlju pokazano je kako kontrolori mogu da vrate podatke tipa `string`, koji se zatim prikazuju u veb čitaču. To je korisno za razumevanje rada kontrolera, ali u bilo kojoj netrivijalnoj veb aplikaciji, primenjuje se šablon: većina aktivnosti kontrolora treba da prikažu dinamične informacije u HTML formatu. Ako akcije kontrolera samo vrate stringove, oni bi morali da rade sa mnogo stringova odnosno zamena. Dakle, neophodno je da postoji neki sistem šabloniranja. A to je zapravo pogled (eng. View).

Uvod

Pogled je odgovoran za definisanje korisničkog interfejsa (UI). Pošto kontroler izvrši odgovarajuću logiku za traženi URL, prosleđuje podatke za prikaz odgovarajućem pogledu. Dakle, to su programske komponente koje su zadužene za definisanje prikaza, istovremeno komponente kojima upravljaju kontroleri.

Pogledi u ASP MVC arhitekturi nisu direktno dostupni, što ih čini drugačijim od tzv. fajl-baziranih (eng. *file-based*) okvira kao što su ASP.NET Web Forms ili PHP. Dakle, poglede nije moguće direktno prikazati u veb čitaču tj. njihov prikaz se uvek generiše (kaže se još i renderuje) od strane kontrolera.

Za formiranje prikaza pogled dobija informacije od kontrolera. Najčešće pogled dobija podatke za prikaz tako što ih u vidu objekata modela prosleđuje kontroler. Pogled transformiše model u format spreman za prezentaciju korisniku. MVC model koristi šablonski kod pa je zbog jednostavnosti povezivanje naziv pogleda isti kao i naziv klase kontrolera. Na primer, za postojeći Home kontroler vide se kreirani pogledi u folderu Home.



Slika 8.1. Pogledi u početnoj MVC aplikaciji

Vidi se da ovi fajlovi imaju ekstenziju **cshtml**. Radi se o programskim komponentama koje se izvršavaju na serverskoj strani i koje formiraju HTML koji se prosleđuje korisniku. Samostalno, bez renderovanja, ove komponente se ne mogu prikazati. Pogledajmo unutrašnjost ovih komponenata i razumećemo i zašto. Otvorimo jednostavni **About.cshtml** i prateću sliku prikaza:

```
@{
ViewBag.Title = "About";
}
<h2>@ViewBag.Title.</h2>
<h3>@ViewBag.Message</h3>
```

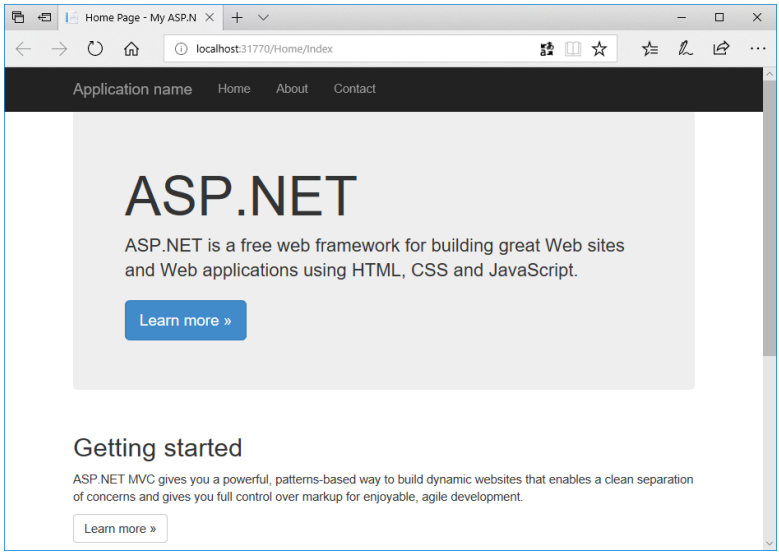
```
<p>Use this area to provide additional
information.</p>
```

Slika 8.2. Izgled About pogleda i prateći deo koda

Kao što se vidi, pogled definisan fajlom **About.cshtml** definiše unutrašnji deo prikaza celog ekrana, bez menija i futera. Ovo nije slučajno tj. tako je definisano s namerom u okviru aplikacije. Kasnije ćemo videti gde i kako. Dakle, svi pogledi predstavljaju sadržaj koji se definiše u unutrašnjem delu i predstavlja odgovor na korisničku akciju. Takođe, zapaža se primena sintakse koja započinje sa znakom **@**. Ovaj znak predstavlja deo sintakse poznate kao **Razor** i koristi se kao oznaka za izvršavanje koda na serverskoj strani pri generisanje HTML sadržaja. Razor sintaksu prestavićemo detaljno u narednom poglavlju.

Podrazumevani pogled

Ako unesemo URL adresu korena sajta, vidi se da se prikazuje pogled metode Index odgovarajućeg kontrolera – **Home**, pogledati sliku.



Slika 8.3. Početni pogled sajta odgovara Index metodi kontrolera

Prikaz predstavlja renderovani HTML sadržaj generisan na osnovu `Index.cshtml` pogleda uz uklapanje u glavni šablon prikaza koji uključuje zaglavlje i podnožje sajta.

Sada ćemo razmotriti prenos podataka od kontrolera do jednog prikaza. Najpre ćemo pogledati najlakši način.

ViewBag

Ovaj objekat posreduje u razmeni manje količine podataka kontrolera i pogleda. Na strani kontrolera, u fajlu `HomeController.cs` u metodi `About`, naći ćete kod:

```
ViewBag.Message = "Your application description page.";
```

kojim se dodeljuje vrednost stringa svojstvu `Message` objekta `ViewBag`, koji se zatim koristi u pogledu.

Upotreba **ViewBag** objekta je potpuno analogna upotrebi objekta **ViewDataDictionary** klase. To je specijalizovana klasa tipa rečnika namenjena skladištenju podataka za prikaz. Zapravo **ViewBag** je omotač oko ove klase. Na primer:

```
ViewData["CurrentTime"] = DateTime.Now;
```

je potpuno analogno sa:

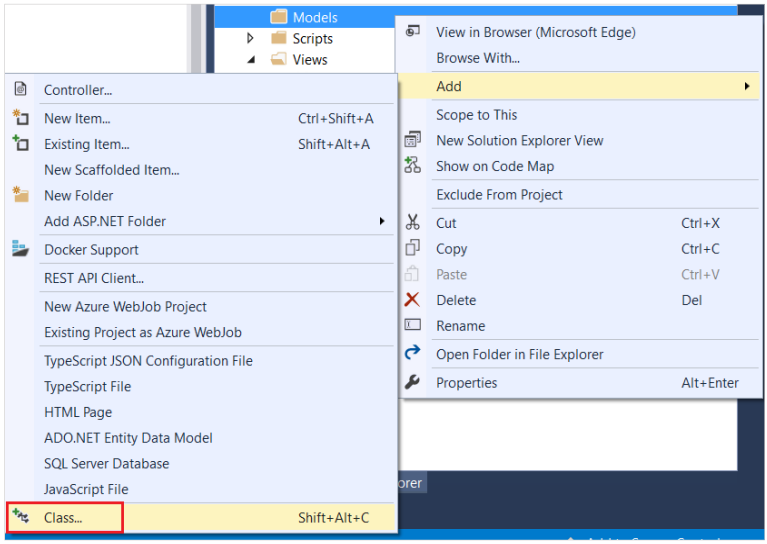
```
ViewBag.CurrentTime = DateTime.Now;
```

Mada je kod u oba slučaja iste funkcionalnosti, primena **ViewBag** objekta je lakša pa se i češće koristi. Dakle, ovaj objekat čuva vrednost za neki jedinstveni naziv, a nazive čuva kao svojstva objekta. Međutim postoji izuzetak kada to nije moguće. Ako je naziv sastavljen od više reči, tada se vrednost ne može adresirati preko **ViewBag** objekta. Za to se koristi **ViewData** na sledeći način:

Kontroler: `ViewData["kljuc sa razmacima"] = "Neki podaci.";`

Pogled: `<h3>@ViewData["kljuc sa razmacima"]</h3>`

Ovaj objekat se može iskoristiti i za prenos liste drugih objekata. Neka u našoj aplikaciji postoji klasa Osoba za opis podataka. U skladu sa MVC arhitekturom ova klasa se smešta u folder **Models**, kao na slici



Slika 8.4. Dodavanje klase modela

Pretpostavimo da objekat **Osoba** ima dva polja i za svako postavimo **set** i **get** metodu za svojstva istog imena, kao u kodu:

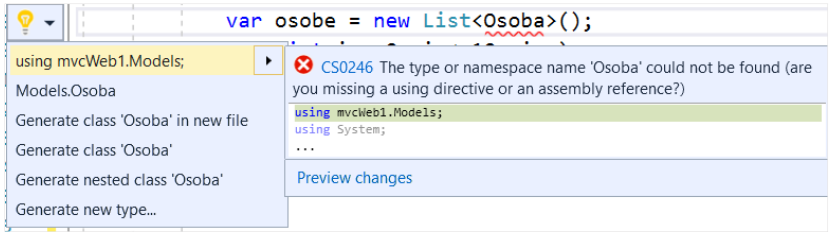
```
namespace mvcWeb1.Models
{
    public class Osoba
    {
        public string jmbg { get; set; }
        public string ime { get; set; }
    }
}
```

Zatim kreirajmo listu ovih objekata sa vrednostima. Ova lista objekata zamenjuje podatke koji će se kasnije čitati iz nekog izvora podataka, na primer relacione baze. Listu formiramo u kontroleru i prosledimo pogledu preko **ViewBag** objekta, kao u kodu:

```
public ActionResult About()
{
    var osobe = new List<Osoba>{
        new Osoba{ jmbg="1111111111112",
                   ime = "Jovan Jovanović" },
```

```
            new Osoba{ jmbg="1111111111113",
                       ime = "Petar Petrović" }
        };
        ViewBag.Osobe = osobe;
        return View();
}
```

Pri dodavanju koda voditi računa da se mora uključiti putanja do foldera Model kako bi se klase modela koristile, pogledajte sliku.



Slika 8.5. Uključivanje putanje do klasa u folderu Modela

Na strani pogleda potrebno je, takođe, uključiti klasu modela i pristupiti listi podataka preko `Razor` sintakse. Na primer:

```
@using mvcWeb1.Models;

<h3>Broj osoba u modelu je: @ViewBag.Osobe.Count</h3>
<h3>Imena osoba su:</h3>
    <ol>
        @foreach (Osoba o in ViewBag.Osobe)
        {
            <li>@o.ime</li>
        }
    </ol>
```

Ovo će formirati sledeći izgled:

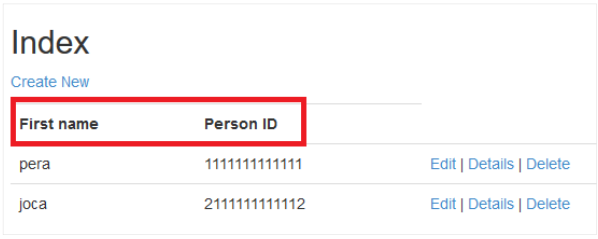
Slika 8.6. Izgled modifikovanog pogleda na osnovu liste Osoba

Anotacije

ASP MVC nudi dodatne mogućnosti za podešavanje svojstava modela u prikazu primenom odgovarajućih **anotacija**. Anotacijama nazivamo dodatni opis smešten u uglaste zagrade a koji stoji ispred naziva metode ili klase. U nekim tekstovima koristi se reč **dekorator**. Na primer, ako je potrebno promeniti naziv svojstva pri prikazu, zbog upotrebe ćirilice, razmaka ili neki drugi razlog onda se za odgovarajuća svojstva modela postavljaju odgovarajuće anotacije.

```
public class Osoba
{
    [Display(Name ="First name")]
    public string ime { get; set; }

    [Display(Name = "Person ID")]
    [Key]
    public string jmbg { get; set; }
}
```

Pomoćna funkcija `@Html.DisplayNameFor` prikazivaće navedeno ime.

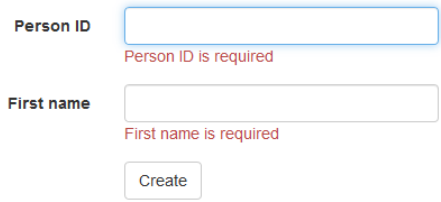


Slika 8.7. Izgled liste nakon izmene naziva polja

Drugo, važno je da, pri unosu ili izmeni podataka, postoji odgovarajuća **validacija** pre snimanja. Određena ograničenja se postavljaju takođe primenom anotacija. Na primer, ako uz svako polje dodamo uslov da je **vrednost polja obavezna**:

```
public class Osoba
{
  [Required(ErrorMessage ="First name is required")]
  [Display(Name ="First name")]
  public string ime { get; set; }

  [Required(ErrorMessage = "Person ID is required")]
  [Display(Name = "Person ID")]
  [Key]
  public string jmbg { get; set; }
}
```

U prikazu, dodata validacija znači i istovremenu proveru ispunjenosti uslova. Ukoliko nije zadovoljeno dobija se izgled kao na slici.



Slika 8.8. Prikaz unosa Osobe

Neki primeri češće korišćenih ograničenja su:

- **Dužina stringa**. Definiše se anotacijom:

  ```
  [StringLength(13, MinimumLength=13 )]
  ```

  U slučaju kada je maksimalna odnosno minimalna dužina 13 karaktera.

- U opštem slučaju može se primeniti **regularni izraz** proizvoljnog zapisa, na primer:
  ```
  [RegularExpression(@"\d{3,13}")]
  ```

Definiše primenu validacije zasnovanu na regularnom izrazu `"\d{3,13}"` kojim se vrednost polja koja se sastoji od cifara dužine od 3 do 13.

Povezivanje sa kontrolerima

Već smo rekli da jednom kontroleru odgovara folder istog imena sa jednim ili više pogleda. Na primer, na startu **Home** kontroler ima tri metode koje renderuje tri pogleda: podrazumevani – `Index`, `Contact` i `About`. Poziv odgovarajućeg pogleda vrši se automatski prosleđivanjem objekta `View()`. Kontroler može eksplicitno zadati poseban pogled navodeći njegovu adresu, na primer:

```
public ActionResult About()
{
    //ViewBag.Message = "Your application...";
    //return View();
    return View("~/Views/Shared/Error.cshtml");
}
```

U ovom slučaju poziva se eksplicitno postojći pogled `Error` u folderu **Views/Shared**. Na ovaj način se, pri izboru opcije `About`, otvara stranica o grešci.

Kada je reč o povezivanju kontrolera i pogleda važno je napomenuti i način prenosa veće količine podataka. Već smo videli da se manja količina podataka veoma lako prenosi pomoću `ViewBag` objekta. U ovaj objekat je moguće smestiti listu drugih objekata.

Međutim, kako se prenosi više podataka?

Podaci se mogu preneti kao argument objekta `View` pri napuštanju akcijskih metoda. Takvi podaci predstavljaju modele. Na primer, metoda kontrolera bi mogla da izgleda ovako:

```
public ActionResult About()
```

```
{
    var osobe = new List<Osoba>{
        new Osoba{ jmbg="1111111111112",
                   ime = "Jovan Jovanović" },
        new Osoba{ jmbg="1111111111113",
                   ime = "Petar Petrović" }
    };
    return View(osobe);
}
```

dok bi odgovarajući pogled bio zasnovan na modelu.

Na vrhu fajla pogleda koristi se naredba `@model` kojom se specificira tip objekta, tačnije tip modela, koji se očekuje kao model. Kada se automatski kreira kontroler onda Visual Studio automatski uključuje naredbu `@model` na vrhu fajla. Na ovaj način kontroler prosleđuje tipizirani objekat `Model`.

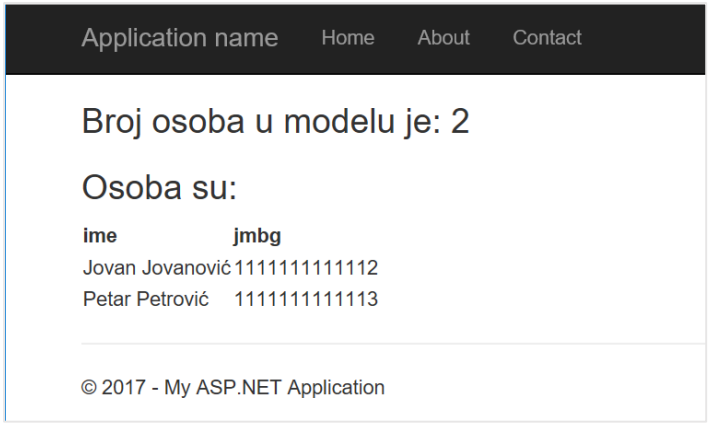
```
@model List<mvcWeb1.Models.Osoba>

<h3>Broj osoba u modelu je: model.Count</h3>
<h3>Imena osoba su:</h3>
    <ol>
        @foreach (var o in Model)
        {
            <li>@o.ime</li>
        }
    </ol>
```

Umesto upotrebe `List` klase za prenos više objekata obično se koristi uopštenje primenom interfejsa `IEnumerable`. Ova generalizacija se primenjuje pri automatskom kreiranju koda od strane Visual Studio-a. U tom slučaju kod na strani pogleda bi bio sledeći:

```
@model IEnumerable <mvcWeb1.Models.Osoba>

<h3>Broj osoba u modelu je: @Model.Count()</h3>
<h3>Osoba su:</h3>
<table>
    <tr>
        <th>
            @Html.DisplayNameFor(x => x.ime)
        </th>
        <th>
            @Html.DisplayNameFor(x => x.jmbg)
```

```
            </th>
        </tr>
        @foreach (var item in Model)
        {
            <tr>
                <td>
                    @Html.DisplayFor(xx => item.ime)
                </td>
                <td>
                    @Html.DisplayFor(xx => item.jmbg)
                </td>
            </tr>
        }
</table>
```

Obratiti pažnju kako se dobija broj osoba iz prosleđene kolekcije. Takođe u kodu je prikazano kako se za dobijanje HTML koda na osnovu vrednosti svojstva objekta odnosno njegovog imena koriste se metode: `@Html.DisplayFor` odnosno `@Html.DisplayNameFor`. Konačan izgled stranice bio bi:



Slika 8.9. Konačan izgled pogleda Osoba

Dodavanje novog pogleda

Dodavanje novog pogleda prikazaćemo za postojeći kontroler Home. Umesto modifikacije About pogleda dodaćemo novi pogled za prikaz liste osoba. Krenimo redom. Najpre je važno **modifikovati šablon stranicu** `_Layout.cshtml` tako da se ubaci stavka menija Osobe.

Na osnovu postojećeg koda, i prethodnog znanja o Razor sintaksi, lako je prepoznati gde su kreirane stavke postojećeg menija, pa ćemo, na sličan način dodati novu stavku.

```
<div class="navbar-collapse collapse">
 <ul class="nav navbar-nav">
  <li>@Html.ActionLink("Home","Index","Home")</li>
  <li>@Html.ActionLink("About","About","Home")</li>
  <li>@Html.ActionLink("Contact", "Contact", "Home", null,
new        {@class="btn        btn-default"        })</li>
<li>@Html.ActionLink("Osobe","Osobe","Home")</li></ul>
</div>
```

Obratiti pažnju da je korišćena jedna pomoćna metoda Razor-a - `@Html.ActionLink()`. Ova pomoćna metoda kreira link na stranici pogleda koji poziva navedenu metodu određenog kontrolera.
Zatim se proširuje Home kontroler:

```
public class HomeController: Controller
{
    public ActionResult Osobe()
    {
        var osobe = new List<Osoba>{
            new Osoba{ jmbg="1111111111112",
                       ime = "Jovan Jovanović" },
            new Osoba{ jmbg="1111111111113",
                       ime = "Petar Petrović" }
        };
        return View(osobe);
    }
    public ActionResult Index()
. . . . . .
```

Osim @Html.ActionLink() pomoćne metode koristi se i metoda @Html.Action() na sličan način ali bez kreiranja linka. Na primer, ako imamo Test1 metodu u kontroleru Home

```
public ActionResult Test1(string coName)
{
    if (coName == "myCoo")
        return Content("Company name is " + coName);

    return RedirectToAction("Contact");
}
```

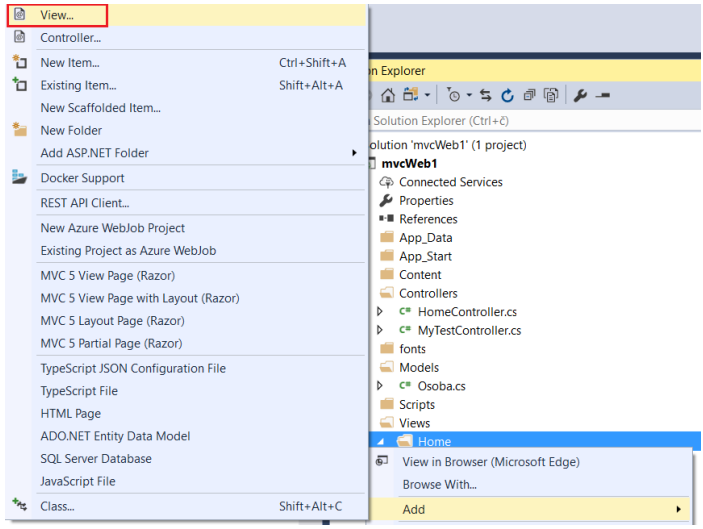
Ovu metodu možemo iskoristiti za ispisivanje poruke u podnožju u _Layout.cshtml dokumentu i pri tome proslediti argument:

```
<footer>
    <p>&copy; @DateTime.Now.Year - My ASP.NET Application
- @Html.Action("Test1", "Home", new { coName = "myCoo"
})</p>
</footer>
```

Obratite pažnju na **način dodavanja parametara metodi**, što se vrši kreiranjem anonimnog objekta sa svojstvom tačno definisanog imena kao ime argumenta.

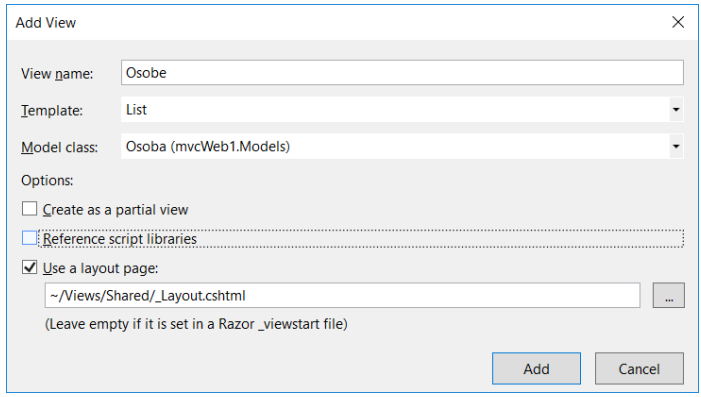
Na kraju se dodaje novi pogled za postojeći kontroler. Procedura je sledeća:

Korak 1. Desnim klik na folder Home u folderu Views i izbor opcije kao na slici:



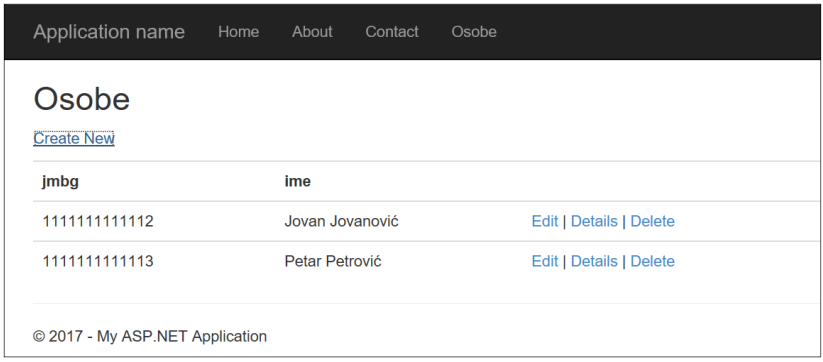
Slika 8.10. Kreiranje novog pogleda

Pošto smo odlučili da novi pogled prikazuje listu objekata Osoba, a ovu klasu imamo već definisanu među modelom, podešavamo opcije kao na sledećoj slici:



Slika 8.11. Podešavanje opcija za pogled Osobe

Sada možemo pokrenuti aplikaciju (F5 ili Ctrl+F5). Ukoliko smo u pogledu Osobe automatski će biti otvoren taj pogled, inače se otvara početna stranica aplikacije. U drugom slučaju izborom opcije menija otvaramo željenu stranicu:



Slika 8.12. Stranica pogleda na listu objekata Osoba

Zapaža se nekoliko linkova: `Create New`, `Edit`, `Details` i `Delete`. Stranice za ove operacije nisu još uvek generisane. Ono što je očigledno jeste da će razvojem više modela generisanje svih ovih stranica postati prilično teško, a da će pri tome struktura koda u njima ostati prilično slična.

Podmeni. Mada je kreiranje podmenija odnosno podstavki jedne stavke menija tipično u desktop aplikacijama, ipak njihova upotreba u veb aplikacijama može predstavljati uvod u komplikovanu korisničku navigaciju. Uz ovo upozorenje prikazujemo izmenu u deljenom pogledu `_Layout.cshtml`, primenom `Bootstrap` klasa.

```
<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact", "Contact",
"Home")</li>
    <li>@Html.ActionLink("To Do","Index","ToDoEmpty")</li>
    <li class="dropdown">
      <a class="dropdown-toggle" data-toggle="dropdown"
href="#">
        Slozena opcija
        <b class="caret"></b>
      </a>
      <ul class="dropdown-menu">
        <li class="active"><a href="#">. . .</a></li>
        <li><a href="#">. . .</a></li>
        <li><a href="#">. . .</a></li>
```

```
        <li><a href="#">. . .</a></li>
      </ul>
    </li>
  </ul>
</div>
```

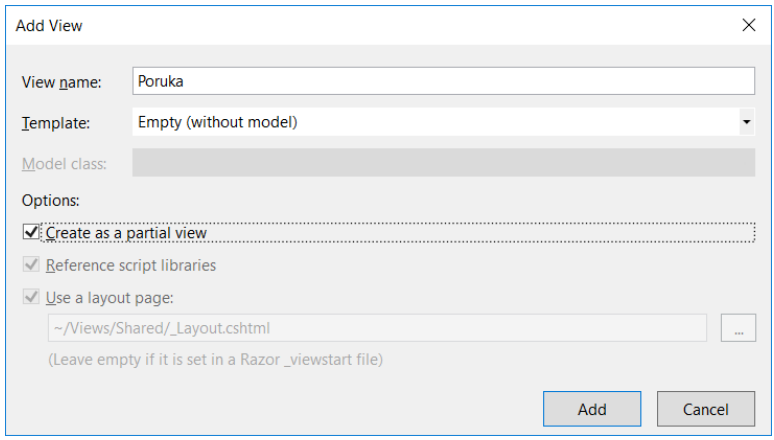
Parcijalni pogled

Parcijalni pogled (eng. Partial View) je pogled koji se renderuje unutar drugog pogleda. Kao i ostali pogledi parcijalni pogled ima file ekstenziju `cshtml`. Ne postoji semantička razlika između parcijalnog i bilo kog drugog pogleda. Osnovna razlika je što se parcijalni pogled ne startuje kroz `_ViewStart.cshtml`.

Na primer ako se u Home kontroler doda metoda Poruka koja ima povratnu vrednost `PartialView()`:

```
public ActionResult Poruka()
{
    ViewBag.Poruka = "Pažnja! Koristi se PartialView!!";
    return PartialView();
}
```

A zatim se parcijalni pogled kreira podešavanjem opcija kao na slici:



Slika 8.13. Podešavanje opcija za parcijalni pogled Poruka

Dobija se sledeći izgled ekrana kao potvrda da parcijalni pogled ne koristi šablon prikaza.

Slika 8.14. Vizuelni prikaz parcijalnog pogleda Poruka

Pitanja i zadaci za proveru znanja

1. Koji su pogledi kreirani pošto se kreira početna aplikacija na osnovu šablona tj. čarobnjaka? Koji pogled je početni tj. podrazumevani?
2. Kako bi preneli neku poruku iz metode kontrolera u pogled?
3. Objekat koje klasa se može koristiti za istu svrhu kao i objekat `ViewBag`? Objasnite razlike ako postoje.
4. Kako bi preneli listu objekata u pogled iz metode kontrolera? A kako bi prikazali listu objekata u pogledu.
5. Šta je to anotacija? Navedite primere anotacija.
6. Kako bi pridružili regularan izraz za unos nekog poštanskog broja za unos u neko polje forme?
7. Kako bi ograničili unos na maksimalno 15 karaktera?

8. Da li je moguće iz neke metode kontrolera eksplicitno pozvati određeni pogled? Objasni kako?
9. Šta je to parcijalni pogled?
10. Kako se koriste parcijalni pogledi? Napiši jedan primer.

9. *Razor*

Uvod

Ova glava je posvećena opisu mehanizma za generisanje HTML stranica na osnovu `cshtml` dokumenata tj šablona pogleda naziva se Razor. Razor je uveden počev od ASP.NET MVC 3 i predstavlja podrazumevani mehanizam za kreiranje pogleda na dalje. Ovo poglavlje se bavi Razor mehanizmom. Razor pruža čist, lagan i uprošćen šematski prikaz pogleda koji ne sadrži sintaktičku krutost.

Razor je jednostavna markup sintaksa za ugrađivanje serverskog koda u ASP.NET veb stranice. U prethodnim primerima neke Razor komande su već objašnjene pošto je njihova primena bila neophodna za početno predstavljanje rada komponenata pogleda. Ova sintaksa je slična drugim sintaksama generisanja HTML stranica na serverskoj strani, ali je njegova upotreba još više pojednostavljena i olakšana. Uporedni prikaz nekoliko sintaksi je dat u tabeli:

Tabela 9.1. Uporedni prikaz sintaksi za generisanje HTML-a

Razor	PHP	Klasični ASP:

`<ul>`
`@for (int i=0; i<5; i++)`
`{`
`  <li>@i</li>`
	`<ul>`
	`<?php`
	`for ($i=0; $i<5;$i++){`
	`  echo("<li>$i</li>");`
	`}`
		`<ul>`
		`<%for i=0 to 5%>`
		`  <li><%=i%></li>`
		`<%next%>`
		`</ul>`

```
}                          ?>
</ul>                      </ul>
```

Osnovne karakteristike *razor* sintakse su:

- C# blokovi koda su uokvireni sa @{ ... }
- Linijski izrazi (promenljive ili funkcije) počinju sa @
- Promenljive su deklarisane sa **var**
- Stringovi su zatvoreni sa znacima navoda
- C# fajlovi imaju ekstenziju **.cshtml**

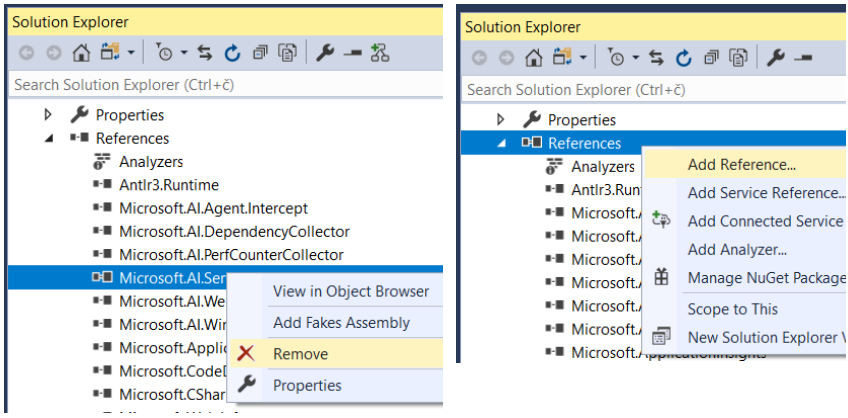
Umetanje vrednosti

Razor implementira umetanje vrednosti neke promenljive ili metode na veoma jednostavan način. U HTML kod ubacuje se znak @ koji prethodi promenljivoj odnosno metodi. Na primer, ako je potrebno prikazati serversko vreme u veb stranici koja se prikazuje klijentu, kod je:

```
<html>
   <body>
       <p>Vreme na serveru je: @DateTime.Now</p>
   </body>
</html>
```

U prethodnom primeru umetnuta vrednost predstavlja tekuće vreme dobijene iz standardne .NET strukture za rada sa podacima tipa datum/vreme `DateTime`. Naravno, primena nekih klasa i metoda u Razor sintaksi znači istovremeno dostupnost tih klasa odnosno prisustvo odgovarajućih biblioteka pri povezivanju aplikacije. U ovom slučaju to je: biblioteka mscorlib.dll odnosno asembli **`mscorlib`**, a imenski prostor (eng. Namespace) je **`System`**.

Biblioteke koje su uključene u projekat mogu se pogledati preko sekcije `References` u prozoru `Solution Explorer`. Preko tog prozora, koristeći konteksni meni dobijen desnim klikom, može se neka biblioteka izbaciti ili dodati projektu, kao na slici:



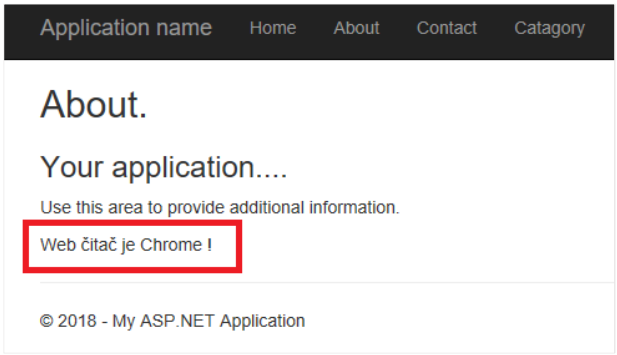
Slika 9.1. Brisanje i dodavanje biblioteka

Umetnuta vrednost može biti i podatak koji se dobija od klijentske strane a koji sadrži podatke o korišćenom veb čitaču. Ovi podaci se koristi, na primer, pri statističkim izveštajima o pristupu stranicama. Za to se koristi klasa `HttpBrowserCapabilities` u već prisutnoj biblioteci `System.Web.WebPages` koristeći Razor sintaksu `@Request.Browser.`

Izmenite neki postojeći pogled dodajući sledeći kod:

```
<p>Veb čitač je @Request.Browser.Browser !</p>
```

Kao rezultat dobijate novi podatak u pogledu:



Slika 9.2. Umetnuta vrednost u prikazu

Blok serverskog koda

Blok C# koda koji se izvršava na serveru, može se umetnuti u pogled preko Razor sintakse tako što se @ a zatim sledi blok koda. Pri tome se uvek koriste vitičaste zagrade, bez obzira da li je jedna ili više linija koda. Na primer:

```
<!-- Deklarisanje promenljive na serveru -->
@{ var zdravoSvete = "Zdravo, zdravo"; }

<p>Vrednost serverske promenljive zdravoSvete:
@zdravoSvete</p>

<!-- Blok od nekoliko naredbi  -->
@{
    var poz = "Dobro došli!";
    var dan = DateTime.Now.DayOfWeek;
    var poruka = poz + " Danas je: " + da;
}
<p>Serverska poruka: @poruka</p>
```

Izrazi

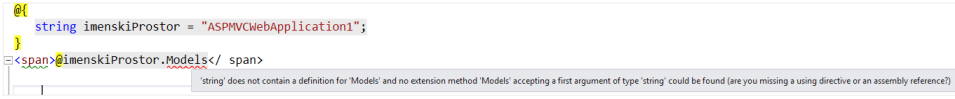
Osobina Razor sintakse jeste da se automatski prebacuje sa koda na markup i obrnuto što ga čini kompaktnim i čistim pri pisanju. Međutim, ova mogućnost može napraviti neke nejasnoće. Pogledajte sledeći izraz:

```
@{
    string imenskiProstor = "ASPMVCWebApplication1";
}
<span>@imenskiProstor.Models</ span>
```

Verovatno očekujete da rezultat bude nadovezivanje stringova, odnosno:

```
<span>ASPMVCWebApplication1.Models</span>
```

Umesto toga dobija se greška da ne postoji svojstvo odnosno ekstenzija koja se naziva **Models** za tip podatka **string**, pogledati sliku.



Slika 9.3. Primer greške usled nerazumevanja izraza u Razor-u

U ovom slučaju Razor nije mogao razumeti nameru napisanog izraza, odnosno tumači **@imenskiProstor.Models** kao izraz u celosti, pokušavajući da odredi da li za tip string promenljive **imenskiProstor** moguće odrediti značenje .**Models**.

Međutim, imajući ovo u vidu, postoji drugo rešenje. Razor sintaksa podržava eksplicitne izraze pri kodovanju tako što ih uvlači u zagrade:

```
<span> @(imenskiProstor).Models </span>
```

Na ovaj način Razor se daje na znanje da je **imenskiProstor** poseban izraz a .**Models** jeste doslovno tekst, van Razor sintakse, koji nije deo kodnog izraza.

Na veb stranicama često se prikazuje mail adresa koja predstavlja string čiji je sastavni deo znak @. Razmotrimo sledeći primer:

```
<span>zoran.cirovic@gmail.com</span>
```

Na prvi pogled, ovo izgleda kao da bi moglo izazvati grešku jer `@gmail.com` izgleda kao ispravan kod izraza gde pokušavamo da odštampamo `com` svojstvo `gmail` varijable. Na sreću, Razor je dovoljno pametan da prepozna opšti obrazac e-mail adrese tako da slobodno možete ostavite ovaj izraz jer je ispravan.

Pomoćne metode

U praksi Razor pogledi često se baziraju na primeni određenih pomoćnih metoda koje formiraju određeni HTML kod uz odgovarajuću logiku. Ove metode se nazivaju **Helper** metodama, a mogu se koristiti naknadno bilo gde u projektu, a zasnovane su na Razor sintaksi za uređivanje prikaza. Pokazaćemo primenu jedne ovakve metode u konkretnom slučaju.

Napisaćemo Razor kod za prikaz vrednosti uz obavezno formatiranje. Za formatiranje prikaza vrednosti koristimo drugu vrednost kao graničnu tako da, ako je vrednost manja od granične prikaz će biti crvene boje, u suprotnom je zelene boje.

Podsetimo se da se definisanje boje teksta na HTML stranici tipično realizuje primenom CSS stila odnosno atributa `style`. Ukoliko definisanje boje treba da nadjača tj. preklopi spoljašnje stilove, što je u ovom slučaju očigledno, onda treba definisati ugradni stil za sam element. Na primer za zelenu boju teksta HTML kod bi bio:

```
<span style="color:green">tekst...</span>
```

Zato ćemo definisati Helper metodu `showValue` koja ima dva argumenta. Prvi argument je vrednost koja se ujedno i prikazuje, a drugi argument je

granična vrednost u odnosu na koju se određuje koji će stil biti aktivan. Helper metoda počinje oznakom `@helper`. U bloku sa kodom može se ugraditi HTML markap kod. Prebacivanje iz C# u HTML markap vrši se automatski, ali se u markap bloku pozivanje pozadinskog koda vrši umetanjem vrednosti promenljive ili funkcije. U našem slučaju ceo kod izgleda:

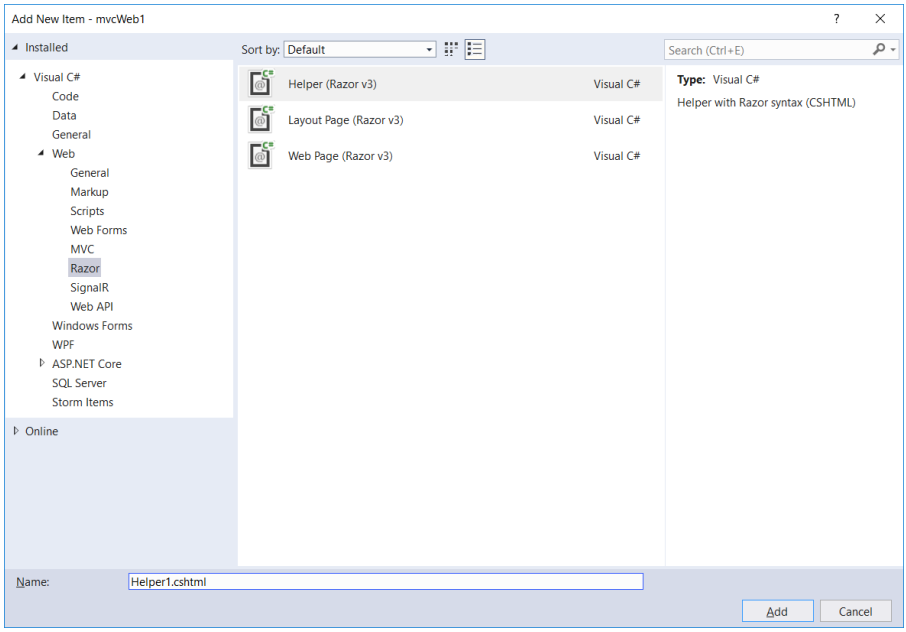
```
@{var x = 33.44; var y = 44.33; }
@showValue(x, 40) <br/>
@showValue(y, 40)

@helper showValue(double v, double g){
    var c = "green";
        if (v < g)
        {
            c = "red";
        }
    <span style="color:@c">@v.ToString("n")</span>
}
```

Kreiranje zajedničkih Helper metoda

Ovde ćemo pokazati kako se kreira Helper metoda u zasebnom fajlu sa kodom a koji kasnije možemo koristiti na više mesta u aplikaciji.

U korenski folder sajta kreirajmo folder imena `App_Code`. Ovo izvodim tako što u prozoru `Solution Explorer` desnim klikom na projekat biramo opcije `Add -> New Folder`. `App_Code` je rezervisano ime za folder u ASP.NET u kome se smešta kod za komponente kao što su Helper metode.

Zatim, u ovaj folder kreirati jedan fajl ekstenzije cshtml. Na sličan način kao u prethodnom slučaju, u prozoru `Solution Explorer` desnim klikom na folder `App_Code` biramo opcije `Add -> New Item`, a zatim se popunjava forma kao na slici.



Slika 9.4. Kreiranje Helper fajla preko VS okruženja

Odaberimo naziv fajla **HelperOsoba.cshtml**.

Sada možemo da ubacimo pomoćne metode. Neka prva metoda bude za uobličavanje ispisa teksta.

```
@helper naslov(string naslov)
{
        <div style="border: 1px solid black; width:90%;
margin-top:20px;">
            <h2 style="color: blueviolet;
width:90%;">@naslov</h2>
        </div>
}
```

Druga metoda služi za tabelarni prikaz podataka iz objekata klase Osoba. Klasu Osoba smo ranije dodali u folder Models. Obratite pažnju da će ova metoda koristi klasu Osoba pa se mora uključiti upotreba ove klase, dakle kod bi bio:

```
@using mvcWeb1.Models @*uključivanje imenskog prostora za
klasu Osoba*@

@helper testOsobe(IEnumerable<Osoba> osobe)
{
    <h2>Osobe za testiranje</h2>
    <table class="table table-striped">
        <thead>
            <tr>
                <th>Ime</th>
                <th>JMBG</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var o in osobe)
            {
                <tr>
                    <td>@o.ime</td>
                    <td>@o.jmbg</td>
                </tr>
            }
        </tbody>
    </table>
}
```

Zapazite da sada u jednom dokumentu imate više pomoćnih metoda.

Dalje, objekte ove klase treba negde u kodu kreirati i proslediti pogledu. Zato modifikujmo metodu Index kontrolera Home, koja je ujedno podrazumevana metoda inicijalnog kontrolera.

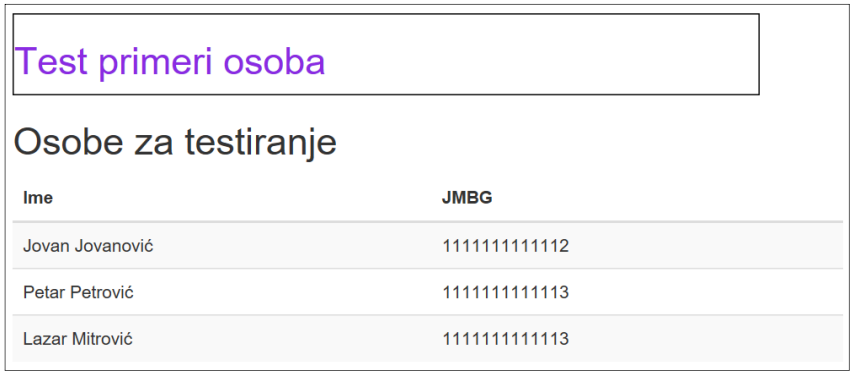
```
//public ActionResult Index()
//{
//    return View();
//}
public ActionResult Index()
{
    var osobe = new List<Osoba>{
        new Osoba{ jmbg="1111111111112", ime = "Jovan
Jovanović" },
        new Osoba{ jmbg="1111111111113", ime = "Petar
Petrović" },
```

```
        new Osoba{ jmbg="1111111111113", ime = "Lazar
Mitrović" }
    };
    return View(osobe);
}
```

Ovom izmenom kreirano je tri objekta i smeštena su u jednu listu. Lista je prosleđena odgovarajućem pogledu **Index.cshtml** koji pripada Home kontroleru. Upotreba **Helper** metoda zasebnog dokumenta izvodi se na način da se navodi ime fajla a u nastavku iza tačke sledi ime pomoćne metode, kao u primeru koji sledi:

```
@{
    ViewBag.Title = "Home Page";
}

@HelperOsoba.naslov("Test primeri osoba")  @*poziv helper
metode*@
@HelperOsoba.testOsobe(Model)              @*poziv helper
metode*@
```

Na ovaj način, dobijamo sopstvene delove prikaza koje ugrađujemo u naše stranice. Dobijeni rezultat ove primene je dat na sledećoj slici:



Slika 9.5. Prikaz primene dve kreirane Helper metode

Nizovi

Rad sa nizovima je dosledan načinu rada sa ostalim referencnim objektima. Nizovi su gotovo obavezni u radu sa podacima, pa treba obratiti posebnu pažnju. Na primer:

```
@{
    string[] voce={"jabuka","kruska","breskva","kajsija"};
    int posK = Array.IndexOf(voce, "kruska");
    int duzinaNiza = voce.Length;
    string x = voce[2];
}
<html>
<body>
    <h3>Members</h3>
    @foreach (var v in voce)
    {
        <p>@v</p>
    }
    <p>Pozicija kruske u nizu od indeksa 0 je: @posK</p>
    <p>Broj elemenata niza: @duzinaNiza</p>
    <p>Vocka na poziciji 2 is @x</p>
</body>
</html>
```

U ovom kodu je kreiran niz **voce** kao niz objekata tipa **string**. Objekti su istovremeno i inicijalizovani u kodu. Nizovi u .NET-u predstavljaju istovremeno i tipove koji imaju sopstvene metode i svojstva, kao na primer svostvo **Length** koje vraća broj elemenata niza.

Na sličan način mogu da se organizuju podaci ali u vidu listi. Na primer:

```
@{
    List<string> voce = new List<string>(); //
deklarisanje i inic. prazne liste
    // dodavanje više elemenata istovremeno u listu
    voce.AddRange( new string[] { "jabuka", "kruska",
"breskva" });
    voce.Add("kajsija"); // dodavanje elementa u listu
    int posK = voce.IndexOf("kruska");
```

```
    int brElListe = voce.Count;
    string x = voce[2];  //  string x = voce.ElementAt(2);
}
```

Čitanje iz fajlova

Pri čitanju podataka iz fajla koriste se standardne klase za rad sa fajlovima, najčešće iz prostora imena **System.IO**. U primeru koji sledi korišćena je metoda **MapPath** za prevođenje relativne putanje do odgovarajuće fizičke putanje na serveru.

```
@{
    var filePath = Server.MapPath("~/App_Data/Osobe.txt");
    Array linijeTeksta = File.ReadAllLines(filePath);
}
<!DOCTYPE html>
<html>
<body>
    <h1>Čitanje teksta iz fajla Osobe.txt</h1>
    @foreach (string linija1 in linijeTeksta)
    {
        @linija1 <text> </text> <br />
    }
</body>
</html>
```

Forme

Forme predstavljaju delove HTML stranice za slanje podataka serveru. Podaci se na formi prikupljaju iz kontrola koje služe za unos. Ove kontrole, osim vrednosti podataka, sadrže i naziv odnosno id koji se šalje serveru. Na serverskoj strani podaci se izdvajaju i obrađuju.

Sledeći primer prikazuje kreiranje forme i prikupljanje podataka koji su prosleđeni iz nje. Za prikupljanje podataka koristi se objekat tipa `HttpRequest`. U konkretnom slučaju taj objekat predstavlja svojstvo `Request` klase `Page` tj. tekućeg objekta. Pojednostavljena sintaksa je `Request["key"]`.

Drugo svojstvo koje je korišćeno u primeru je `IsPost`. Ovim svojstvom se razdvaja da li je zahtev od stranice tipa `GET` ili `POST`. Ako je zahtev tipa `POST`, svojstvo `IsPost` će vratiti vrednost `true`.

```
@{
    var imagePath = "";
    if (IsPost)
    {
        string ime = Request["ime"];
        string jmbg = Request["jmbg"];
        if (Request["cbSlika"] != null)
        { imagePath = "img/" + Request["cbSlika"]; }
        else { imagePath = ""; }
        <p>
            Uneli ste: <br />
            Ime: @ime <br />
            JMBG: @jmbg <br />
            slika: @imagePath
        </p>
    }
}

<form method="post" action="">
    Ime:<br />
    <input type="text" name="ime" value="" /><br />
    JMBG:<br />
    <input type="text" name="jmbg" value="" /><br /><br />
    Slika:<br />
    <select name="cbSlika">
        <option value="Slika 1.jpg"> Slika 1</option>
        <option value="Slika 2.jpg"> Slika 2</option>
        <option value="Slika.3.jpg"> Slika 3</option>
    </select>
    <input type="submit" value="Send" class="submit" />
    @if (imagePath != ""){
        <p>
            <img src="@imagePath" alt="Sample" />
```

```
        </p>
    }
</form>
```

Skrivanje osetljivih informacija

U ASP.NET fajlovi čiji naziv počinje sa podcrtom ne mogu da se prikažu na vebu ili čitaču. Dakle, ako želite da sakrijete sadržaj koda ili prikaz od korisnika potrebno je da se to koristi. Ovo je način da se sačuvaju lozinke, mail adrese i slično. Njihovim čuvanjem u fajlu `_AppStart`:

```
@{
    WebMail.SmtpServer = "mailserver.firmaxyz.com";
    WebMail.EnableSsl = true;
    WebMail.UserName = "mpetrovic@firmaxyz.com";
    WebMail.Password = "mojalozinka";
}
```

podaci ostaju sigurni bez načina za njihov prikaz krajnjem korisniku.

Metode za organizaciju pogleda

Razor sintaksa obiluje velikim brojem korisnih metoda. Pomoću ove sintakse lako se formira šablon prikaza u MVC projektu u kome se pojedini pogledi koriste za prikaz unutrašnjeg dela stranice dok zaglavlje, meni i futer ostaju nepromenljivi. Neke važne metode korišćene za to su:

- `@RenderBody()`– metoda koja se nalazi u glavnoj stranici aplikacije. To je stranica koja se koristi za generisanje osnovnog izgleda ostalih stranica. U standardnom šablonu smeštena je u folder `Shared` i naziva se `_Layout.cshtml`. Postoji samo jedan metod `RenderBody` za jednu `Layout` stranicu. Označava mesto na kome se generiše HTML kod pogleda koji se ugrađuje u celu stranicu.
  Na primer, ako je šablon `Layout.cshtml`:

  ```
  <html>
  ```

```
<body>
    <p>Paragraf 1</p>
    @RenderBody()
    <p>Paragraf 2</p>
</body>
</html>
```

onda bi sledeći kod:

```
@{Layout = "Layout.cshtml";}
<h1>NASLOV</h1>
<p>Lorem ipsum...</p>
```

generisao stranicu sa ugrađenim sadržajem:

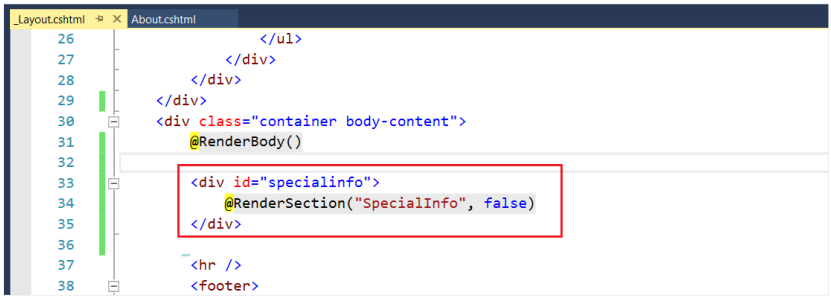
```
<html>
<body>
    <p>Paragraf 1</p>
    <h1>NASLOV</h1>
    <p>Lorem ipsum...</p>
    <p>Paragraf 2</p>
</body>
</html>
```

- `@RenderPage()`– metoda koja importuje sadržaj iz drugog fajla i ugrađuje HTML kod u novi fajl koji se prikazuje. Ovaj metod može imati jedan ili dva argumenta. Prvi je fizička lokacija fajla, drugi je opcioni i predstavlja niz objekata koji se prosleđuje stranici.

```
<html>
<body>
    @RenderPage("header.cshtml")
    <h1>Naslov Veb stranice </h1>
    <p>Ovo je paragraf</p>
    @RenderPage("footer.cshtml")
</body>
</html>
```

- Stranica `_Layout` sadrži koncept sekcija za prikaz, ali može imati samo jedan `RenderBody` metod. Međutim, moguće je da postoji više sekcija tj. odeljaka. Da bi se prikazala jedna sekcija koristi se metod `RenderSection`. Razlika između `RenderSection` i `RenderPage` metode je što `RenderPage` čita sadržaj iz datoteke, dok metoda

`RenderSection` pokreće blok koda. Prvi parametar metode `RenderSection` definiše ime sekcije u Layout šablonu. Drugi parametar je opcioni i definiše da li je renderovanje obavezno ili ne. Ako je renderovanje obavezno onda Razor šalje izuzetak ako sekcija nije realizovana u šablonu prikaza. Ubacivanje sekcije je prikazano na slici ispod:



Slika 9.6. Ubacivanje nove sekcije u osnovni prikaz

dok se kod u pogledu koristi na sledeći način:

```
@section SpecialInfo{
    <p>Ovo je odeljak za posebne informacije!</p>
    <p>Vreme je: @DateTime.Now.ToShortTimeString()</p>
}
```

Pitanja i zadaci za proveru znanja

1. Šta je Razor? Koja je namena ove sintakse?
2. Kako se ubacuje serverski kod?
3. Napišite primer u kom ubacujete u pogled tekući datum, vreme kao i dan u nedelji.
4. Šta je to eksplicitni izraz u Razor sintaksi? Navedite primer.
5. Kako se pišu pomoćne metode?

6. Napišite primer pomoćne metode koja prikazuje padajuću listu nekih podataka.
7. Napišite primer pomoćne metode koja prikazuje listu padajućih lista nekih podataka.
8. Napišite sve načine prosleđivanja podataka od forme ka nekoj metodi kontrolera.
9. Kako se mogu sakriti osetljivi podaci ili delovi koda od korisnika?
10. Koje metode za organizaciju pogleda poznajete?
11. Objasnite upotrebu metode `RenderBody`.
12. Objasnite upotrebu metode `RenderPage` odnosno `RenderSection`. Koja je razlika između njih?

10. Uvod u *Bootstrap*

Dizajn pogleda zasniva se na primeni gotovog okruženja **Bootstrap - BS**. Bootstrap je jedno kompletno, moćno i često korišćeno okruženje za brzu, laku izradu veb orijentisanih aplikacija pre svega orijentisanih ka mobilnim uređajima (eng. mobile first). Ovo poglavlje je posvećeno prikazu osnova BS okruženja. BS je zasnovan na primeni HTML, CSS odnosno JavaScript-a. Bootsrap su razvili Mark Otto i Jacob Thornton za Twitter aplikaciju. Objavljen je kao projekat otvorenog koda u avgustu 2011 na repozitorijumu GitHub.

Stranice zasnovane na ovom okruženju su prilagodljive, tačnije pripadajući CSS stilovi se podešavaju za različite širine prikaza krajnjih uređaja: desktop, tablet i mobilne uređaje).

Osnovni elementi pogleda

Struktura

Struktura stranice zasniva se na mrežastom sistemu (eng. Grid). Mreža sadrži redove i kolone. Za svaki red definiše se od 1 do 12 ćelija tj kolona. Mrežasta struktura obezbeđuje različit prikaz za različite širine ekrana.

CSS

Bootstrap sadrži CSS podešavanja od osnova sa globalnim CSS podešavanjima, stilizacijom osnovnih HTML elemenata i dodacima na osnovni izgled koja su izvedena na osnovu proširenja osnovnih klasa.

Komponente

Bootstrap sadrži na desetine komponenata namenjenih za višestruko korišćenje. Ove komponente omogućavaju na primer sopstvenu ikonografiju, padajuće liste, navigaciju, pomoćne prozore (eng. alert), promene na osnovu pozicije miša itd.

JavaScript

Bootstrap sadrži i na desetina posebnih komponenata iz biblioteke jQuery. Lako se uključuju sve ili jedna po jedna.

Prilagođenje

Prilagođenje Bootstrap komponenata je moguće kao i jQuery plugin-ova i tako dobijanje sopstvenih komponenata.

Poslednja verzija Bootstrap-a može se dobiti na lokaciji: http://getbootstrap.com/.

Slika 10.1. Logo Bootstrap okruženja na stranici http://getbootstrap.com/

Primena BS

Struktura dokumenata

Postoje dva osnovna pristupa korišćenja okruženja. Prvi je putem preuzimanja fajlova i njihovo postavljanje na odgovarajući veb server. Drugi je povezivanje preko odgovarajućeg URL-a. Ukoliko se vrši preuzimanje fajlova moguće je preuzeti izvorni kod ili već pripremljene tj. kompajlirane fajlove.

Ako se koristi **kompajlirana** verzija Bootstrapa, vrši se najpre preuzimanje ZIP fajla, a zatim ekstrakcija svih fajlova. Dobija se struktura fajlova/direktorijuma kao na slici:

```
bootstrap/
├── css/
│   ├── bootstrap.css
│   ├── bootstrap.min.css
├── js/
│   ├── bootstrap.js
│   ├── bootstrap.min.js
└── img/
    ├── glyphicons-halflings.png
    └── glyphicons-halflings-white.png
```

Slika 10.2. Struktura fajlova u slučaju kompajliranih fajlova

Kao što se vidi, postoje verzije CSS odnosno JavaScript fajlova (bootstrap.*), kao i umanjene verzije CSS odnosno JS (bootstrap.min.*). U strukturu su takođe uključeni fontovi (ikone) *Glyphicons*.

Ako se preuzme izvorni kod okruženja Bootstrap tada je struktura fajlova kao na sledećoj slici:

```
├── less/
├── js/
├── fonts/
├── dis/
│   ├── css/
│   ├── js/
│   └── fonts/
├── docs-assets/
├── examples/
└── *.html/
```

Slika 10.3. Struktura fajlova u slučaju fajlova izvornog koda

Fajlovi u folderima **less/**, **js/**, odnosno **fonts/** su izvorni kod za Bootstrap CSS, JS, odnosno icon fontove respektivno.

Folder **dist/** sadrži sve što je navedeno u prekompajliranoj verziji koja je već opisana.

Folderi **docs-assets/**, **examples/**, odnosno ***.html** sadrže Bootstrap dokumentaciju.

Šabloni povezivanja

Osnovni šablon za izradu HTML stranica predstavlja način povezivanja preuzete ili linkovane verzije Bootstrap-a.

HTML šablon za preuzete fajlove

```
<!DOCTYPE html>
<html>
   <head>
      <title>Bootstrap 101 Template</title>
      <meta name="viewport" content="width=device-width,
initial-scale=1.0">
      <!-- Bootstrap -->
```

```
      <link href="css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
      <h1>Hello, world!</h1>
      <!-- jQuery -->
      <script src="https://code.jquery.com/jquery.js">
      </script>
      <!-- Include all compiled plugins -->
      <script src="js/bootstrap.min.js">
      </script>
   </body>
</html>
```

HTML šablon za linkovani Bootstrap

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>naslov stranice</title>
    <meta charset="utf-8">
    <meta    name="viewport"    content="width=device-width,
initial-scale=1">
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bo
otstrap.min.css">
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jque
ry.min.js"></script>
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/boot
strap.min.js"></script>
</head>
<body>
</body>
</html>
```

Projektovanje pogleda

Mrežasta struktura

Osnovna struktura stranice zasnovane na Bootstrap sistemu uključuje prilagodljivu fluidnu mrežu sastavljenu od maksimalno 12 kolona. Sistem uključuje predefinisane klase za lako definisanje izgleda.

Mrežasti sistem se koristi za kreiranje vizuelnog prikaza zasnovanog na redovima i kolonama u kojima je smešten sadržaj za prikaz.

- Redovi (eng. rows) se koriste za kreiranje horizontalne grupe kolona.
- Kolone (eng. cols) čuvaju sadržaj koji se prikazuje. Kolone su sadržane u redovima.
- Formatiranje i poravnavanje redova obavlja se u klasi *.container* ako je definisane širine odnosno u *.container-fluid* ako je pune širine.

Mrežasta struktura je definisana primenom predefinisanih klasa. Tako klasa .row ili .col-4 predstavljaju predefinisane klase za red odnosno kolonu širine 4 od 12 jedinica. Međutim, koristeći određenu sintaksu mogu se dobiti i drugačiji prikazi.

Kolone sadrže u sebi i rastojanje između kolona preko pedding-a.

```
<div class="container">
    <div class="row">
        <div class="col-??-*"></div>
        <div class="col-??-*"></div>
    </div>
    <div class="row">
        <div class="col-??-*"></div>
        <div class="col-??-*"></div>
        <div class="col-??-*"></div>
    </div>
```

```
    <div class="row">
        ...
    </div>
</div>
```

U primeru koji je dat neke klase su zapisane su pomoću opštih oznaka. Znakovi pitanja definišu širinu izlaznog uređaja za koji se definiše struktura, a zvezdice definišu širinu. Konkretne nazive klasa videćemo u narednim poglavljima. Najpre, pogledaćemo primenu medija upita kroz Bootstrap okruženje.

Medija upiti

Bazira se na korišćenju `@media` pravila. Standardno pravilo primene upita je:

```
@media = mediji and/or uslovZaOsobinuMedija
```

Na primer:

```
<style>
   @media screen and (max-width: 500px) {
        body {
           background-color: yellow;
        }
   }
</style>
```

Bootstrap definiše nekoliko prelomnih tačaka. Za uređaje kojima je širina ekrana definisana u određenom opsegu definišu se klase prikaza. Ove klase imaju veoma dosledan način definisanja naziva, tačnije razlikuju se u specifičnim dodacima na osnovni naziv. Za mrežastu strukturu koriste se 4 tipa kasa. To su:

Koriste se 4 klase:

xs – za telefone. Ova klasa je **podrazumevana**

sm – za tablete

md – za desktop računare

lg – za velike desktop računare

Napomena: Svaka klasa koja se primenjuje odnosi se ne samo za definisanu veličinu već i za veličine ekrane koje su veće. Na primer ako je definisana klasa xs (eng. extra small) a nije definisana sm (eng. small) onda se klasa xs primenjuje za ekrane na koje bi se odnosila klasa sm.

Ovde su već definisane prelomne tačke i primena sopstvenih upita bi bila sledeća:

```
/* (tablets, 768px i širi) */
@media (min-width: @screen-sm-min) { ... }

/* (desktops, 992px i širi) */
@media (min-width: @screen-md-min) { ... }

/* (large desktops, 1200px i širi) */
@media (min-width: @screen-lg-min) { ... }

/* više prelomnih tačaka! */
@media (min-width: @screen-sm-m) and (max-width:
@screen-sm-max) { ... }
@media (min-width: @screen-md-min) and (max-width:
@screen-md-max) { ... }
```

Primer

Napisati HTML kod koji će prikazati promenu vertikalnog u horizontalni raspored za tri kolone jednog reda, različite boje u slučaju širih ekrana od tablet računara.

CSS:

```
[class|="col"]
{
  border: solid 1px black;
```

HTML:

```
<div class="container-fluid">
 <div class="row">
  <div id="crveno" class="col-sm-4">
```

```
}                                    Crveno
#crveno{                           </div>
  background-color:red;            <div id="plavo" class="col-sm-4">
}                                    Plavo
#plavo{                            </div>
  background-color:blue;           <div id="belo" class="col-sm-4">
}                                    Belo
#belo{                             </div>
  background-color:white;        </div>
}                                </div>
```

Rezultat:

Slika 10.4. Prikaz prilagođenog prikaza

Obrazloženje:

Kreirane su tri kolone. Svaka kolona širine 4 jedinice. Ukupan broj jedinica po kolonama za jedan red je 12. `class="col-sm-4"`.

Klasa `sm` je odabrana tako da se 3 kolone vide u prikazu predviđanom za tablet ekrane i šire, dok za uže prikaz će biti promenjen tako da svaka stavka bude jedna ispod druge.

Više prelomnih tačaka

Kada je potrebno dodati više prelomnih tačaka, odnosno kreirati posebne prikaze za mobilne, table, desktop onda se dodaje više klasa odnosno uslova. Na primer, napraviti mrežastu strukturu koja će imati tri kolone u razmeri:

- 3:4:5 – u slučaju mobilnih uređaja
- 4:4:4 – u slučaju tablet računa
- 5:4:3 – za desktop (i šire ekrane)

```
<div class="container-fluid">
    <div class="row">
```

```
        <div id="crveno" class="col-xs-3 col-sm-4 col-md-
5">
            Crveno
        </div>
        <div id="plavo" class="col-xs-4 col-sm-4 col-md-
4">
            Plavo
        </div>
        <div id="belo" class="col-xs-5 col-sm-4 col-md-3">
            Belo
        </div>
    </div>
</div>
```

Ofset klase

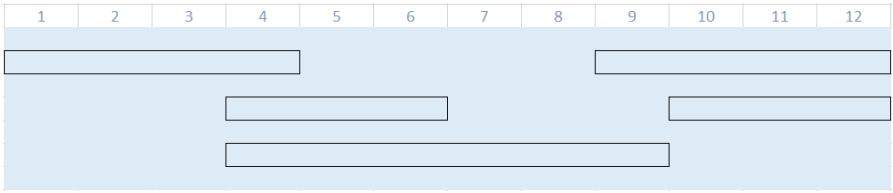
U mrežastom prikazu moguće je i preskočiti određene kolone tj. ćelije mreže za određeni stil prikaza. Za ovo koristi se tzv. ofset (eng. offset) klase. Sintaksa je:

`col-??-offset-*`

gde ?? predstavlja oznaku xs, sm, md, lg, xl a * broj kolona u jedinicama 1/12.

Primer

Napisati stranicu koja će imati 3 reda sa po dve ćelije za sadržaj u svakom redu, kao na slici.



Slika 10.5. Prikaz primene `ofset` klasa

Rešenje:

```
<div class="row">
  <div class="col-md-4">.col-md-4</div >
  <div class="col-md-4 col-md-offset-4">.col-md-4 .col-md-
offset-4</div >
</div >

<div class="row">
  <div class="col-md-3 col-md-offset-3">.col-md-3 .col-md-
offset-3</div >
  <div class="col-md-3 col-md-offset-3">.col-md-3 .col-md-
offset-3</div >
</div >

<div class="row">
  <div class="col-md-6 col-md-offset-3">.col-md-6 .col-md-
offset-3</div>
</div >
```

Ugnježdavanja

Jedna ćelija mrežaste strukture definisana je redom i kolonom. U jednu ćeliju može se upisati novi red sa više kolona i to nazivamo udnježdavanjem. Dakle, ugnježdavanjem sadržaja u već postojeću mrežu vrši se dodavanjem novog reda tj odeljka klase `.row` i postavljanjem `.col-??-*` kolona unutar postojeće `.col-??-*` kolone.

```
<div class="row">
  <div style="background-color: green" class="col-sm-3">
      Roditeljski red: .col-sm-3
  </div>
  <div style="background-color: yellow" class="col-sm-9">
      Roditeljski red: .col-sm-9
      <div class="row">
          <div class="col-xs-8 col-sm-6">
              Dete 1: .col-xs-8 .col-sm-6
          </div>
          <div class="col-xs-4 col-sm-6">
              Dete 2: .col-xs-4 .col-sm-6
          </div>
      </div>
```

```
  </div>
</div>
```

Promena redosleda u prikazu

Ako su kolone zapisane u jednom redosledu moguće ih je prikazati u drugačijem. U praktičnoj primeni promena redosleda se obavlja kada je to zaista neophodno. Ovo se postiže primenom klasa `.col-??-push-*` odnosno `.col-??-pull-*` gde je * u opsegu od **1** do **11**.

U narednom primeru vrši se zamena redosleda ovih kolona koristeći pomenuta svojstva.

```
Pre uredjivanja
<div class="row">

    <div class="col-md-3" style="background-color:
lightblue; border: solid 1px">
        Levo
    </div>
    <div class="col-md-6" style="background-color:
lightgreen; border: solid 1px">
        Desno
    </div>
</div><br>

Posle push i pull uredjivanja
<div class="row">
    <div class="col-md-3 col-md-push-6" style="background-
color: lightblue; border: solid 1px">
        Levo
    </div>
    <div class="col-md-6 col-md-pull-3" style="background-
color: lightgreen; border: solid 1px">
        Desno
    </div>
</div>
```

Međutim, sa novom verzijom Bootstrap je uveo pojednostavljenje promene redosleda primenom novih klasa **.order**. Tako bi u novoj verziji preuređivanje bilo:

```
<div class="col-md-3 order-2"
<div class="col-md-6 order-1"
```

Slika 10.6. Prikaz promene redosleda prikaza

Skrivanje sadržaja

Klase koje se koriste za prikaz ili skrivanje sadržaja u zavisnosti od širine ekrana (tj uređaja) prikazane su u tabeli. Koristi se neka od kombinacija.

Tabela 10.1. Klasa prikaza odnosno skrivanja

Klasa	Extra small (<768px)	Small (≥768px)	Medium (≥992px)	Large (≥1200px)
.visible-xs-*	Visible	Hidden	Hidden	Hidden
.visible-sm-*	Hidden	Visible	Hidden	Hidden
.visible-md-*	Hidden	Hidden	Visible	Hidden
.visible-lg-*	Hidden	Hidden	Hidden	Visible
.hidden-xs	Hidden	Visible	Visible	Visible
.hidden-sm	Visible	Hidden	Visible	Visible

.hidden-md	Visible	Visible	Hidden	Visible
.hidden-lg	Visible	Visible	Visible	Hidden

Primer

```
<h1 class="visible-xs"> EXTRA SMALL Ekran.</h1>
<h1 class="visible-sm"> SMALL Ekran.</h1>
<h1 class="visible-md"> MEDIUM Ekran.</h1>
<h1 class="visible-lg"> LARGE Ekran.</h1>
```

Vidljiv samo jedan tekst i to onaj koji odgovara širini ekrana

```
<span class="hidden-xs"> EXTRA SMALL Ekran.</span>
<span class="hidden-sm"> SMALL Ekran.</span>
<span class="hidden-md"> MEDIUM Ekran.</span>
<span class="hidden-lg"> LARGE Ekran.</span>
```

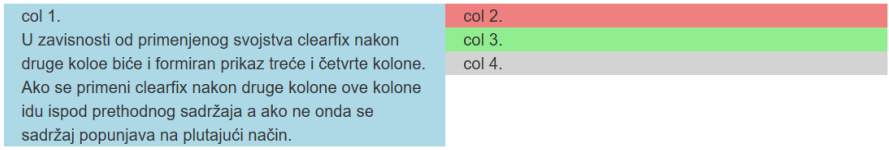
Nije vidljiv samo jedan tekst i to onaj koji odgovara širini ekrana

Isključivanje svojstva plutanja

BS ima sopstvenu klasu kojom obezbeđuje primenu isključivanja svojstva plutanja. Najbolje je objasniti kroz jedan primer.

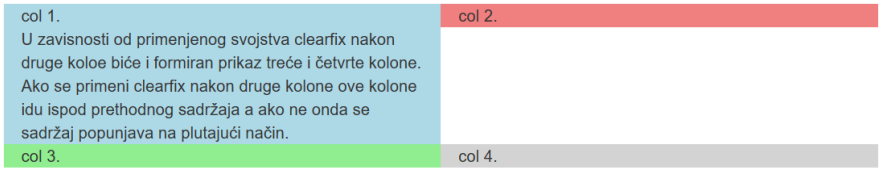
```
<div class="container">
<div class="row">
    <div class="col-xs-6 " style="background-
color:lightblue;">
        col 1.<br />
        U zavisnosti od primenjenog svojstva clearfix
nakon druge koloe biće i formiran prikaz treće i
        četvrte kolone. Ako se primeni clearfix nakon
druge kolone ove kolone idu ispod prethodnog sadržaja a
ako ne
        onda se sadržaj popunjava na plutajući način.
```

```
    </div>
    <div class="col-xs-6 " style="background-
color:lightcoral;">col 2.</div>
    <!--<div class="clearfix "></div>-->
    <div class="col-xs-6 " style="background-
color:lightgreen;">col 3.</div>
    <div class="col-xs-6 " style="background-
color:lightgray;">col 4.</div>
</div>
                                </div>
```



Slika 10.7. Standardan prikaz bez isključena plutanja

Nakon primene clearfix-a iza druge kolone, ubaciti komentarisani kod:



Slika 10.8. Prikaz nakon primene **`clearfix`** klase

Pitanja i zadaci za proveru znanja

1. Šta je Bootstrap (BS)?
2. Kako se BS može primeniti za neki HTML dokument?
3. Objasni mrežastu strukturu. Napiši kod koji prikazuje raspored časova koristeći mrežastu strukturu.
4. Koje vrste kontejnera postoje i koja je razlika između njih?
5. Šta su to medija upiti?

6. Koji tipovi klasa postoje u primeni za mrežastu strukturu?
7. Ako je na nekom odeljku primenjena samo klasa `sm`, za koje veličine ekrana važi klasa?
8. Kako se dodaje više prelomnih tačaka? Objasniti na primeru.
9. U koju svrhu se koriste ofset klase? Navedite primer.
10. Kako se može podeliti širina ekrana na 144 dela?
11. Koje BS klase i kako se one primenjuju za promenu redosleda prikaza?

11. *Database First*

Uvod

Gotovo je nemoguće zamisliti ozbiljniju veb aplikaciju a da nema posebnu celinu za rad sa podacima. Nekada je potrebno da se radi sa već postojećom bazom, a nekada sa lokalnom bazom koja je namenjena pre svega samoj veb aplikaciji. Razvojno okruženje nudi viši nivo apstrakcije u radu sa podacima i na taj način odvaja deo rada sa podacima od ostatka aplikacije. Zahvaljujući takvom odvajanju nije od posebnog interesa koja baza podataka je u pozadini aplikacije.

U ovom poglavlju bavimo se povezivanjem veb aplikacije sa postojećom bazom. U radu sa podacima koristi se skup klasa biblioteke Entity Framework. Ovaj skup klasa obezbeđuje rad sa podacima preko komponente Model u okviru MVC arhitekture. Koristeći *Entity Framework* moguće kreirati model na osnovu koga se zatim ostvaruje perzistencija podataka u bazi, a takođe je moguće je i na osnovu postojeće baze kreirati modele koji se kasnije koriste u kontrolerima za rad sa podacima.

Najpre ćemo razmatrati projektovanje aplikacija gde se pretpostavlja postojanje baze i podataka.

Najčešći vid čuvanja i rada s podacima je posredstvom nekog servera baze podataka. Danas postoji veliki broj različitih servera koji obezbeđuju organizaciju i višekorisnički pristup podacima. Sve više su u upotrebi i *nosql* baze podataka koje u novim sistemima zasnovanim na velikom broju uređaja vezanih na mrežu obezbeđuju bolje performanse.

Ukoliko baza podataka već postoji onda se aplikacija vezuje za takvu bazu i koristi modele koji se formiraju na osnovu modela baze podataka. Ovaj pristup je poznat kao *Database First* pristup u projektovanju. ASP.NET poseduje mehanizam poznat pod nazivom *Scaffolding* kojim se obezbeđuje efikasno generisanje operacija *Create*, *Read*, *Update* odnosno *Delete* (CRUD) kao i prateće poglede za određene modele.

Povezivanje aplikacije i baze

Kako bi mogli aplikaciju povezati sa nekim serverom baze podataka, potrebno je da imate pristupne parametre do neke postojeće baze ili da neku bazu instalirate. Pristupni parametri uključuju podatke o serveru:

- ip adresa (nekada je umesto ip adrese moguće koristiti ime računara na mreži),
- ime servera (pošto je moguće da postoji više servera na istom računaru),
- korisničko ime
- lozinka.

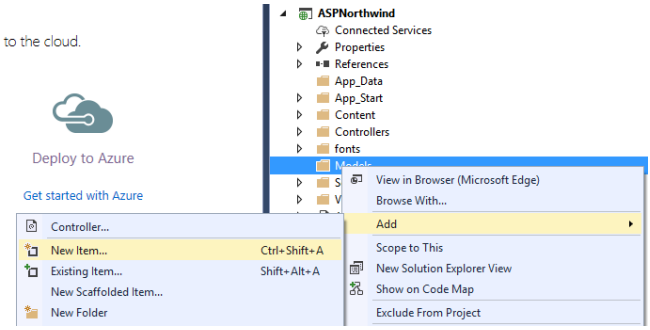
Pri izradi projekta nije neophodno da instalirate neku od verzija MS Sql Server-a. Razlika u postupku postoji samo u Koraku 5. tj. razlika je pri definisanju neposredne konekcije do izvora podataka.

Krenimo od samog početka i napravimo novi projekat.

Korak 1: Otvoriti Visual Studio i napraviti novi projekat ASPNorthwind.

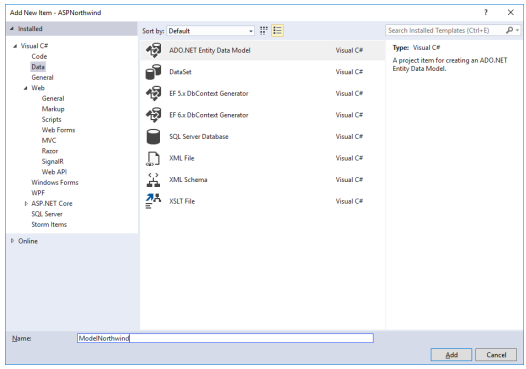
Korak 2: Označiti MVC project šablon.

Korak 3: U prozoru `Solution Explorer` dodajte novu stavku desnim klikom na Models pa onda New Item.

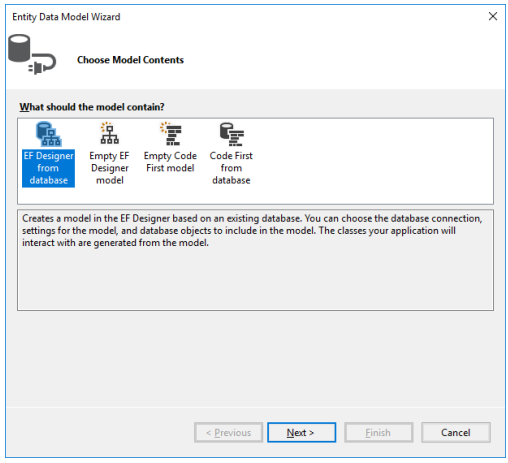


Slika 11.1. Dodavanje nove stavke u projektu

Korak 4: Zatim označiti ADO.NET Entity Data Model.

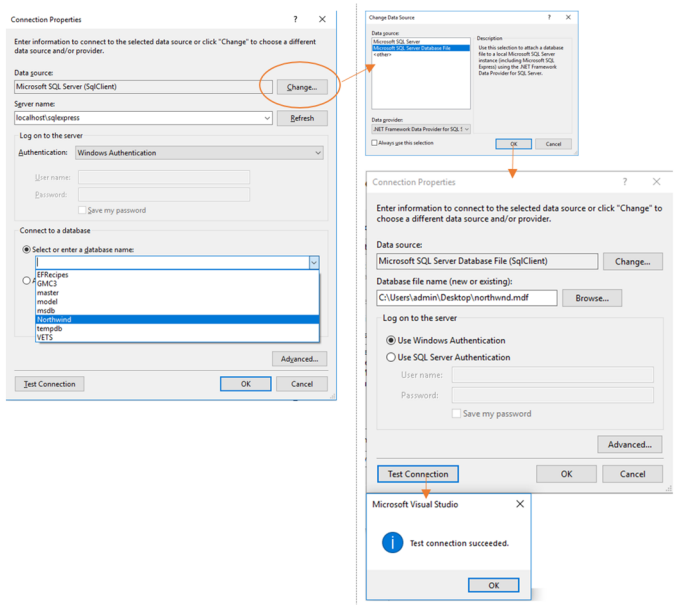


Slika 11.2. Izbor nove stavke – ADO.NET Entity Data Model



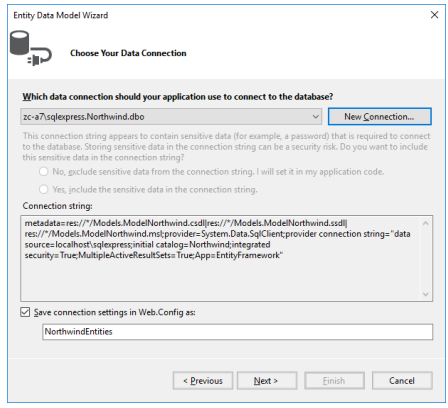
Slika 11.3. Način formiranja modela

Korak 5: U narednom koraku označiti postojeću konekciju ako postoji ili dodati novu. Dodavanje nove konekcije za lokalni server baze *SQL express* izgleda kao na slici. Na desnom delu slike prikazan je izbor druge vrste konekcije, na primer pri vezivanju za postojeći mdb fajl tj za lokalnu bazu.



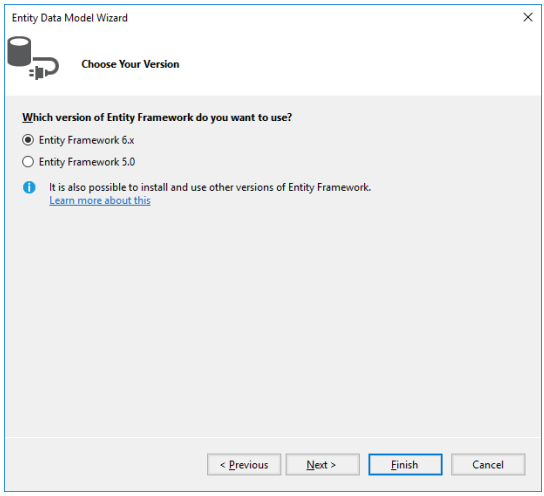
Slika 11.4. Definisanje svojstva konekcije

A zatim se bira konekcija do servera:



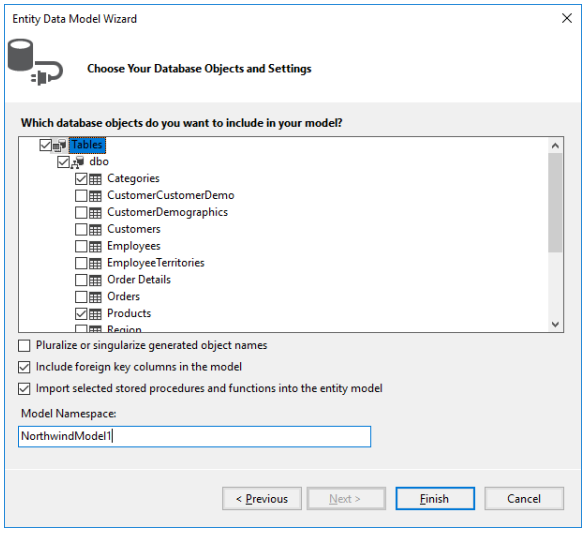
Slika 11.5. Izbor konekcije

Korak 6: Ako postoji više instaliranih verzija Entity Framework biblioteke za rad sa podacima, vrši se izbor odgovarajuće.



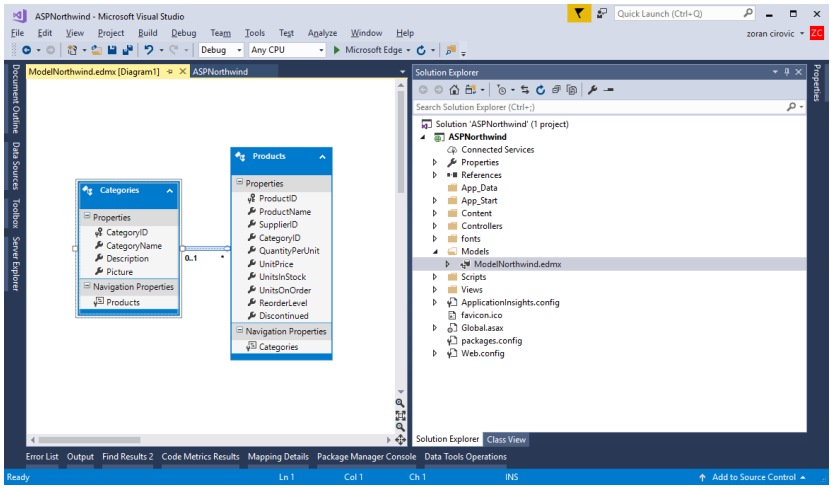
Slika 11.6. Definisanje svojstva konekcije

Korak 7: Biraju se objekti za formiranje modela. Na primer odaberimo Products i Categories tabele. Takođe u polje Model Namespace definiše se imenski prostor za klase koje će biti generisane.



Slika 11.7. Izbor tabela za kreiranje modela

Korak 8: Visual Studio automatski kreira Model. Šema formiranog modela je data na narednoj slici. U pozadini modela formirano je nekoliko fajlova koji odgovaraju konceptualnom sloju, odnosno sloju skladištenja (prema izvoru podataka) kao i sloju koji povezuje ova dva. Sve eventualne izmene na modelu treba da se vrše preko IDE okruženja.



Slika 11.8. Vizuelni prikaz kreiranog modela

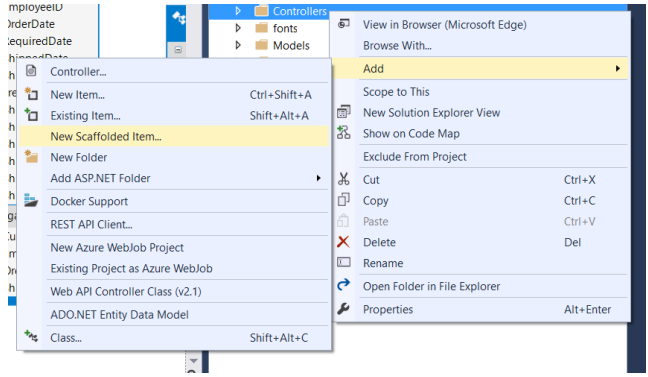
Na ovaj način automatizovano je kreiran jedan `Entity Data Model` za pristup podacima u bazi.

Nakon ovog postupka prelazimo da dodavanje kontrolera i pogleda i rad sa pogledima.

Dodavanje kontrolera i pogleda

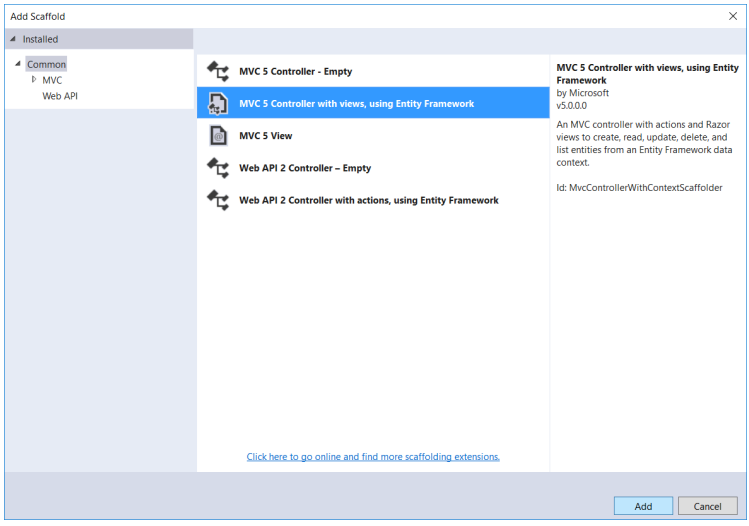
MVC razvojno okruženje Visual Studio nudi lak način za kreiranje kontrolera koji će formirati različite poglede koristeći Entity Framework. Imajući u vidu da je postupak veoma sličan za različite modele koristi se tzv. `Scaffolding`. Pokrenimo sledeću proceduru.

Korak 1: U prozoru `Solution Explorer` desnim klikom na folder `Controller`, birate stavku `Add`, a zatim i stavku `New Scaffolded Item` (ili skraćeno samo birate stavku `Controller`)



Slika 11.9. Dodavanje novog kontrolera

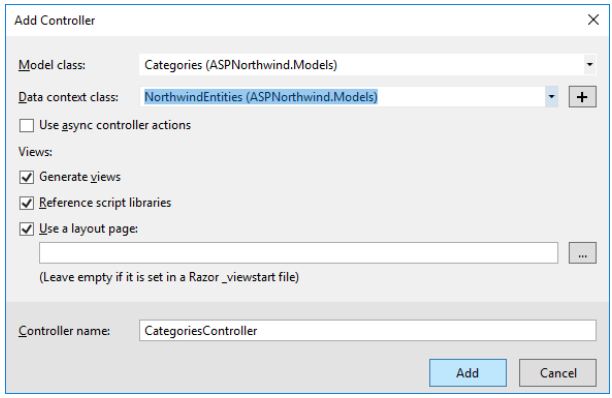
Korak 2: U sledećoj formi za popunjavanje, označiti kontroler sa opcijama pogleda kako je prikazano na slici:



Slika 11.10. Izbor stavke

Korak 3: U narednoj formi unesite ime kontrolera za odabrani model odnosno **DbContext** klasu, kako je prikazano. Obratite pažnju da će se u padajućoj listi klasa modela naći sve klase modeli vaše aplikacije.

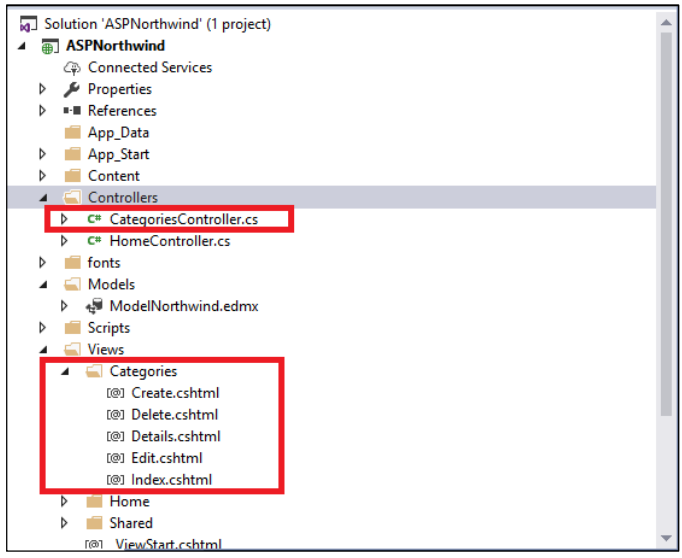
Konteksna klasa za podatke je EDM klasa koja je formirana i takođe se nalazi u odgovarajućoj padajućoj listi.



Slika 11.11. Popunjavanje podataka pre kreiranja kontrolera

Važno. Pristup do klasa podrazumeva validne asemblije tj. biće prijavljena greška prevodioca ukoliko pre ovog koraka niste kompajlirali vaš projekat.

Nakon dodavanja kontrolera razvojno okruženje automatski kreira `CategoriesController` i odgovarajuće poglede:



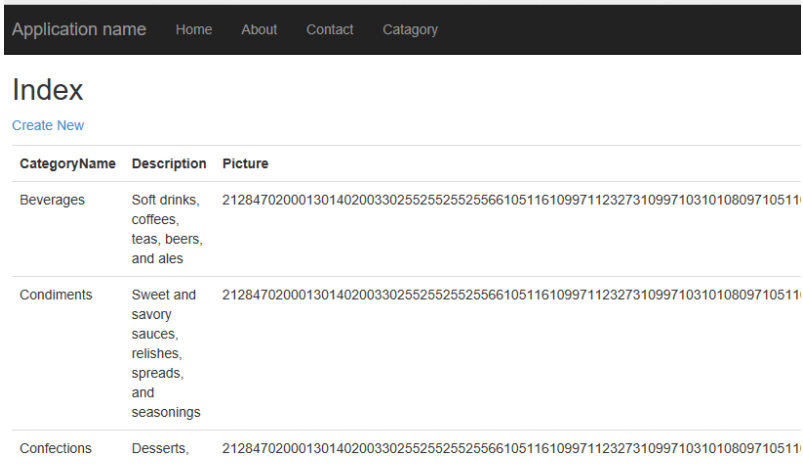
Slika 11.12. Stablo fajlova i foldera nakon dodavanja kontrolera

Ne zalazeći u detalje već sa namerom da se vidi šta je to što je već napravljeno, promenićemo postojeći meni tako da možemo pristupiti svakom od novih pogleda za kategorije posebno. Naravno, pri tome treba imati na umu da se komande Delete, Details i Edit tiču određene stavke odnosno da se mora prethodno definisati red u tabeli Categories nad koji se izvršava komanda. Komanda Index prikazuje listu kategorija, a Create kreira novu kategoriju.

Na primer, promenimo osnovni meni koristeći fajl **_Layout.cshtml** dodajući stavku za kategorije koja će biti povezana sa Index metodom kontrolera:

```
<div class="navbar-collapse collapse">
  <ul class="nav navbar-nav">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
    <li>@Html.ActionLink("Contact",              "Contact",
"Home")</li>
    <li>@Html.ActionLink("Cat…","Index","Categories")</li>
    </ul>
</div>
```

Ove izmene u kodu formiraju novi padajući meni za kategorije kao na slici:



Slika 11.13. Izgled automatski kreiranog pogleda i nove stavke u meniju

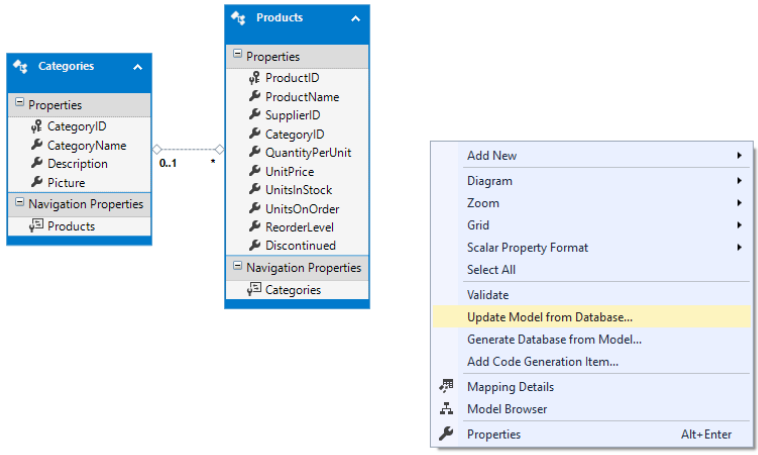
Naravno, pre nego što nastavimo dalje, neophodno je izbaciti kolonu Picture ili je modifikovati na način da prikazuje sliku. Izbacivanje kolone tabele je trivijalno. Prikazivanje slike je nešto složenije i prikazaćemo jedno od mogućih rešenja. Potrebno je da modifikujete kod pogleda na način koji je prikazan:

```
@*@Html.DisplayFor(modelItem => item.Picture)*@
@if(item.Picture != null && item.Picture.Length>78)
{
    <img
src="@Convert.ToBase64String(item.Pi
cture,78,item.Picture.Length-78)" alt="Image" />
}
```

Izmena modela

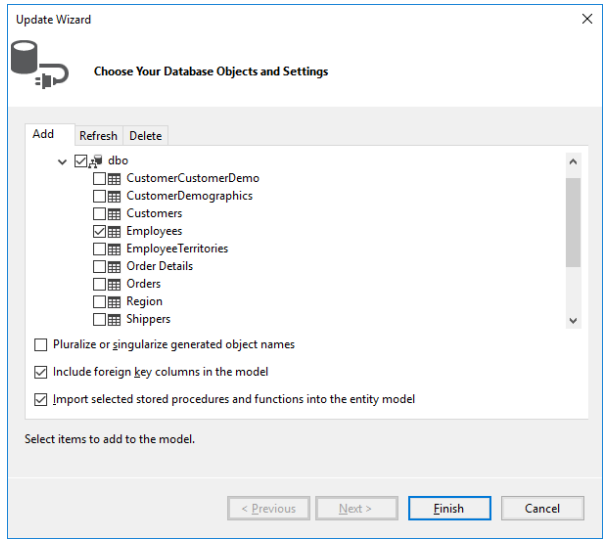
Postojeći model radi sa podacima iz tabela Products odnosno Categories. Proširenje modela koji je već povezan sa bazom vrši se ažuriranjem postojećeg modela. Postupa je sledeći:

Korak 1. Na kontekstnom meniju grafičkog editora modela, koji dobijamo desnim klikom na editor, biramo stavku Update Model from Database.



Slika 11.14. Pokretanje postupka ažuriranja modela

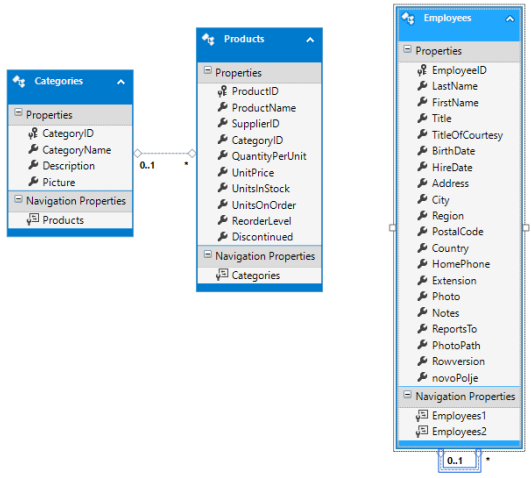
Korak 2. U novoj formi biramo karticu Add, a zatim u njoj tabelu iz baze koju hoćemo da dodamo modelu.



Slika 2. Izbor kartice za dodavanje entiteta u postupku ažuriranja modela

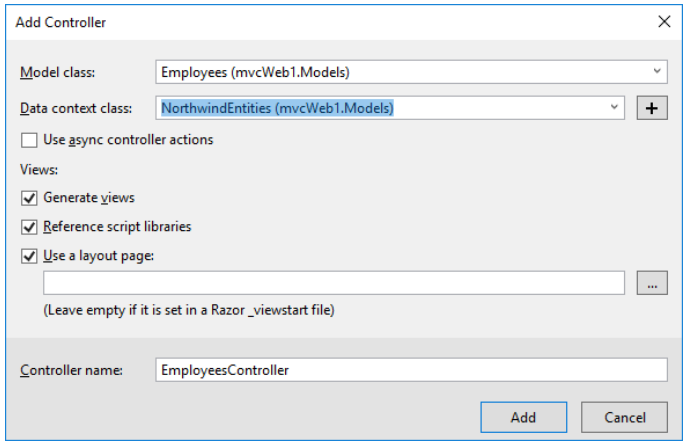
Napomena. Ukoliko je potrebno promeniti postojeći model tj. već kreirane entitete `Categories` odnosno `Product` onda bi trebalo preko kartice `Refresh` odabrati već postojeće entitete i njih osvežiti.

Dobija se sledeći model.



Slika 11.15. Izgled novog ažuriranog modela

Korak 3. Kompajlirati projekat, a zatim kreirati kontrolere i poglede na osnovu modela koristeći `Scaffolding`.



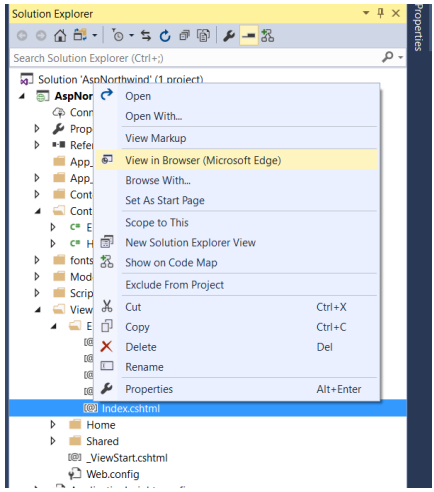
Slika 11.16. Dodavanje kontrolera i pogleda

Prilagođenje generisanih metoda

Sve metode i pogledi su generisani automatizovano primenom `Scaffolding` mehanizma a na osnovu podataka iz modela. Zato je neophodno pogledati prikaz i kontrolere i uraditi neophodne izmene.

Metode *Create* i *Read*

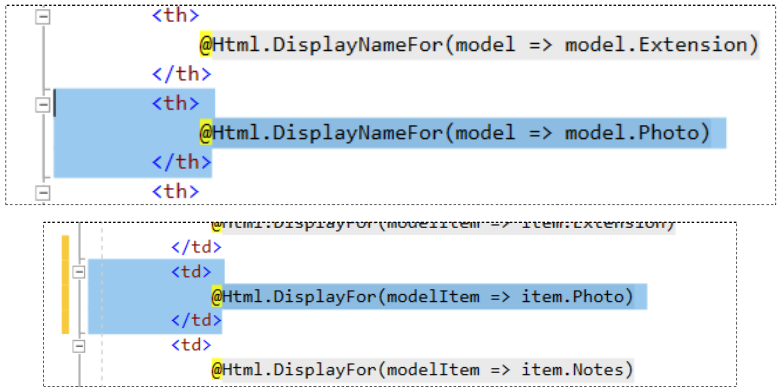
Korak 1: Označiti fajl `Index.cshtml` u pogledu `Employees` i otvoriti ga u veb čitaču preko kontekstnog menija.



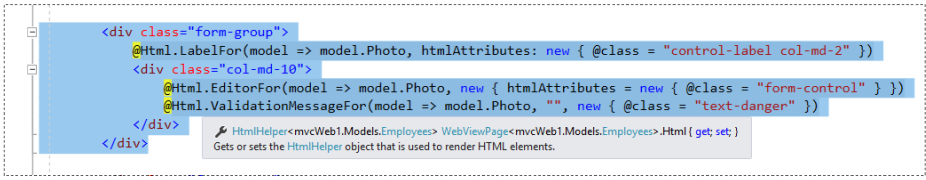
Slika 11.17. Pokretanje određenog pogleda iz kontekstnog menija

Korak 2: Testirajte kreiranje novog radnika klikom na link `Create New`. Obratite pažnju na polje `Photo`. Podaci ovog polja se prikazuju u vidu niza kodova, a ne kao slika. Razlog je u načinu skladištenja slika. Slike u Northwind bazi su male po veličini i čuvaju se po tabelama kao binarni nizovi, baš kao što mogu biti sačuvani bilo koji fajlovi. To su tzv. BLOB podaci. Zato se pri automatizovanom generisanju pogleda i kontrolera takvi podaci predstavljaju kao niz kodova. Dekodovanjem tih podataka može se izdvojiti odgovarajući format.

Dakle, prilagođenje treba da se uradi u svim pogledima koji sadrže polje **Photo**. U našem slučaju pokazaćemo na slikama jednostavno izbacivanje ovih polja iz **Create.cshtml** odnosno **Index.cshtml** prikaza.



Slika 11.18. Izbacivanje **Photo** podataka iz **Create.cshtml**



Slika 11.19. Izbacivanje **Photo** podataka iz **Index.cshtml**

Zatim, nakon izmena u svim pogledima, unesite nekoliko novih zapisa, otvorite svaki modifikovan pogled i verifikujte da li ste ispravno uradili ovaj deo.

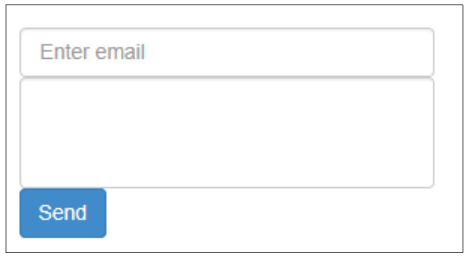
Slanje poruke

Ubacimo u pogled namenjen za prikaz kontakta, **Contact.cshtml**, u okviru kontrolera **Home**, jednu formu za slanje poruke. Na primer, na početku pogleda dodajte sledeći kod:

```
<h3>@ViewBag.Poruka</h3>
<form method="post">
```

```
    <div class="form-group">
        <input type="email" class="form-control"
name="inputMail" placeholder="Enter email">
        <textarea class="form-control" name="inputText"
rows="3"></textarea>
        <button type="submit" class="btn btn-
primary">Send</button>
    </div>
</form>
```

Ako pokrenete i pogledate ovaj prikaz dobija se novi izgled sa formom za slanje poruke:



Slika 11.20. Kreirana forma za slanje

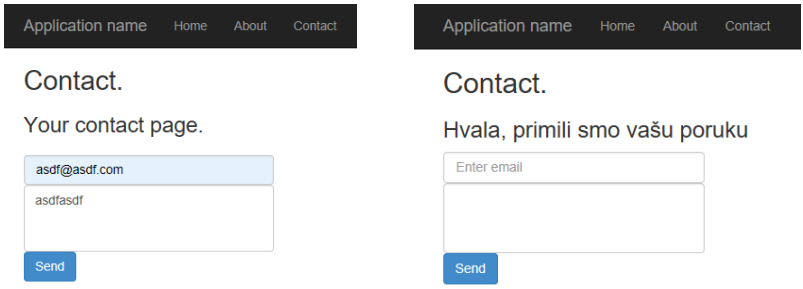
Forma je definisana da šalje podatke metodom `Post` ,`<form method="post">`, ali nije navedeno eksplicitno kojoj metodi se poziv prosleđuje. Podrazumevano, naziv metode odgovara istoimenoj metodi kontrolera koja odgovara pogledu. Dakle, ako je pogled `Contact.html` onda će poziv sa dva argumenta biti prosleđen istoimenoj metodi kontrolera `Home`. Ono što se još zapaža jeste da se šalju dve tekstualne vrednosti sa forme preko ulaznih polja: `name="inputMail"` odnosno `name="inputText"`. Svaka od dve vrednosti jedinstveno je određena svojim nazivom.

Dakle, metodu za prihvatanje post poruke kontrolera Contact napisaćemo sa dva argumenta. Argumenti u toj metodi **moraju** svojim **imenom** da ogovaraju imenima kontrola koje šalju podatke, dakle dodajemo u `HomeController.cs` fajl novu `Contact` metodu:

```
public ActionResult Contact()
{
```

```
    ViewBag.Message = "Your contact page.";
    ViewBag.Poruka = "Tekst poruke...";
    return View();
}

[HttpPost]
public ActionResult Contact(string inputMail, string
inputText)
{
    ViewBag.Poruka = "Hvala, primili smo vašu poruku";
    return View();
}
```

U ovom primeru samo smo naveli jednu fiksnu tekstualnu poruku koju ćemo vratiti pri generisanju pogleda koji nastaje pozivom iz nove metode. Takvu poruku samo prikazujemo na vrhu novog pogleda.



Slika 11.21. Slanje i odgovor na poruku.

Promena url adrese

Podrazumevano ime pogleda odgovara metodi u kontroleru. Ukoliko imamo više različitih metoda za koje vezujemo isti pogled ili želimo da url adresa do tog pogleda sadrži neke posebne znake poput crtice „-“ onda se posebno zadaje ime akcije atributom **ActionName** uz metodu, na primer:

```
[ActionName ("about")]
```

```
public ActionResult Abouttttt()
{
    ViewBag.Message = "Your application....";
    return View();

}
```

Ili

```
[ActionName ("about-asp-mvc")]
public ActionResult About()
{
    ViewBag.Message = "Your application....";
    return View("About");

}
```

Ove metode možete testirati direktnim pozivom metode kontrolera:

…/Home/About-asp-mvc

Filteri

Jedan od načina da se modifikuju metode jeste primenom filtera za metode. Filteri se dodaju u uglastim zagradama iznad metode. Takvo označavanje se naziva još i anotacija, dekorator ili spoljašnji atribut. Postoji više vrsta filtera a najčešće korišćeni su:

Autorizacioni filer

`Authorize` – definiše koje grupe korisnika imaju pristup do posebnih metoda. Ovaj atribut ima skup parametara uključujući `Role` i `Users`, pogledajmo primer:

```
[Authorize(Roles = "administrator", Users = "kondor")]
[HttpPost]
public ActionResult Contact(string inputMail, string
inputText){    }
```

U ovom primeru, samo korisnik „kondor“ koji je „administrator“ može pristupiti ovoj metodi. Ako je filter samo `[Authorize]` pristup ima svaki ulogovani korisnik.

Filter može biti pridružen pojedinoj metodi, kao što je prikazano, ili svim metodama jedne klase, ako se postavi iznad deklaracije klase, na primer:

```
[Authorize]
public class HomeController : Controller{     }
```

U ovom slučaju, moguće je napraviti izuzetak preklapajući autorizacioni filter cele klase posebnim filerima za pojedine metode, na primer:

```
[Authorize]
public class HomeController : Controller{
    public ActionResult Index(){
        return View();
    }

    [AllowAnonymous]
    public ActionResult About(){
        ViewBag.Message = "Your application....";
        return View("About");
    }
```

Sopstveni filter

Stoji uz odgovarajuće metode. Inače ovaj filer nasleđuje klasu **ActionFilterAttribute** i preklapa metodu **OnActionExecuting**.

```
public class myFilterAttr : ActionFilterAttribute

{
  public override void
OnActionExecuted(ActionExecutedContext filterContext){
    var request = filterContext.HttpContext.Request;
    //Logger.logRequest(request.UserHostAddress);
    base.OnActionExecuted(filterContext);
  }
}
```

Ovako napisana klasa može se kasnije koristiti kao novi filter:

```
[myFilterAttr]
```

```
public ActionResult Contact(){}
```

Da bi se novi filter koristio globalno potrebno je uraditi registraciju u **FilterConfig** klasi u folderu **App_Start**.

```
public static void
RegisterGlobalFilters(GlobalFilterCollection filters){
    filters.Add(new HandleErrorAttribute());
    filters.Add(new myFilterAttr());
}
```

Rezultujući filter

Dodaje osobine na već urađene operacije. Tipičan primer je **OutputCache** filter koji obezbeđuje keširanje vrednosti za određenu metodu na određeno vreme, iskazano u sekundama i na određenoj lokaciji:

```
[OutputCache(Duration = 3600, VaryByParam = "none")]
public ActionResult Index(){   }

[OutputCache(Duration = 60, VaryByParam = "id")]
public ActionResult Details(int? id){   }
```

Filer izuzetaka

Stoji uz metodu na koju se odnosi, a definiše tip izuzetka i odgovarajući pogled za taj izuzetak, ukoliko se želi postaviti više različitih pogleda. Na primer:

```
[HandleError(View="MathError", ExceptionType =
typeof(DivideByZeroException))]
[HandleError(View = "StackOfError", ExceptionType =
typeof(StackOverflowException))]
public ActionResult Create(){…}
```

Povezani podaci

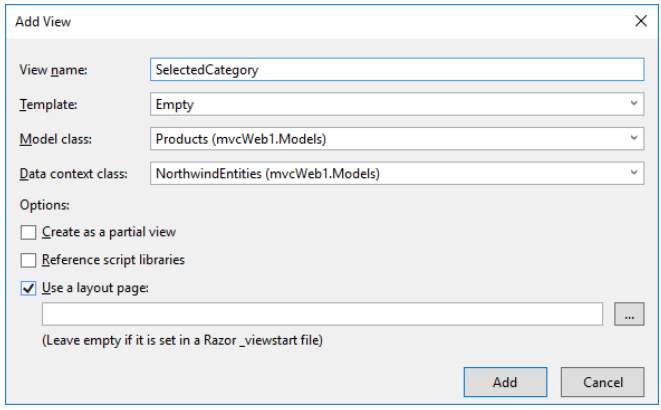
Na veb stranicama je čest prikaz povezanih podataka tj. podataka koji su međusobno uslovljeni. Na primer, pri filtriranju na osnovu zadatih uslova. Postoji više načina za realizaciju ovakvog povezivanja. U nastavku će biti prikazan jedna tehnika koja se oslanja uglavnom na primenu kroz ASP.NET.

Konkretno, pošto je podatak o kategoriji (entitet Categories) proizvoda sadržan u proizvodu odnosno u entitetu Products, onda bi novu interaktivnu veb stranice mogli da kreiramo tako da na prikazu imamo jednu padajuću listu za kategorije i tabelu za proizvode. U padajućoj listi se prikazuju podaci o kategorijama, a izborom jedne kategorije, u tabeli se prikazuju proizvodi koji pripadaju toj kategoriji.

Korak 1. Opredelimo se za neki postojeći kontroler ili kreirajmo novi. U ovom slučaju biram postojeći kontroler Home.

Korak 2. U odabranom kontroleru definišimo metodu koja će biti zadužena za formiranje ovog pogleda. Neka to bude metoda `SelectedCategory()`.

Korak 3. Koristeći IDE kreiramo odgovarajući pogled



Slika 11.22. Izbor opcija za generisanje pogleda

Novi pogled je generisan na osnovu praznog šablona. Dobijen je kod:

```
@model mvcWeb1.Models.Products

@{
    ViewBag.Title = "SelectedCategory";
}

<h2>SelectedCategory</h2>
```

Korak 4. Dopunimo pogled sa padajućom listom. Pri tome padajuću listu sve podatke koje korisnik unosi a koje je potrebno poslati nekoj metodi smeštamo u formu. Za kreiranje padajuće liste i forme koristićemo Helper metode.

```
@using (Html.BeginForm("SelectedCategorie", "Home"))
{
    // SelectedCategorie - naziv metode koja prima podatke
    // koji se šalju u okviru kontrolera Home

    @Html.DropDownList("categoryName",
(SelectList)ViewBag.Values, "Select a category", new {
onchange = "this.form.submit();" })
    // categoryName - naziv elementa koji šalje podatke
    // podaci se primaju na serverskoj strani pridruženi
ovom nazivu

}
```

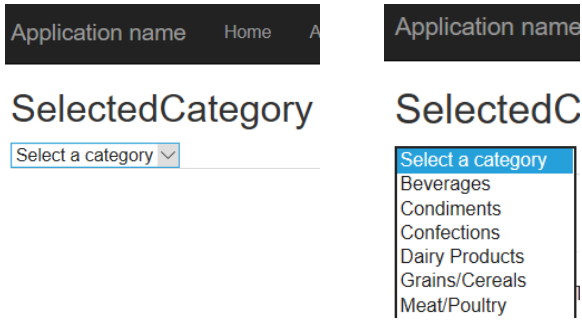
Obratiti pažnju na sve detalje u prethodnom kodu. Argumenti metode `DropDownList` su:

- `categoryName` – naziv html elementa koji šalje podatke, na serverskoj strani se na osnovu ovog imena određuju poslati podaci.
- `(SelectList)ViewBag.Values` – lista podataka koja se prikazuje, tipa `SelectList`
- `Select a category` – početni tekst koji se prikazuje
- `new{onchange = "this.form.submit();"}` – atribut koji se kreira i kome pridružujemo JavaScript kod. Kod znači slanje podataka istovremeno sa odabirom stavke. Forma za ovaj kod

koji pripada padajućoj listi se dobija referencom `this.form`, a slanje se inicira metodom submit, `this.form.submit()`

- Formi je pridružena metoda za obradu podataka `SelectedCategorie` u okviru kontrolera `Home`.

Nakon ovih izmena, kompajlirate aplikaciju i pokrenite sajt.



Slika 11.23. Padajuća lista na osnovu prethodnog koda

Sada je potrebno prihvatiti podatak od forme odnosno odabranu kategoriju, zatim uraditi filtriranje i na kraju te podatke prikazati u tabeli.

Podatak se može prihvatiti pomoću iste metode koja se koristila i pri formiranju prvog izgleda, ali metodu treba proširiti za argument koji se prima/šalje.

```
public ActionResult SelectedCategory(string categoryName)
{
    // priprema podataka za padajucu listu
    System.Web.Mvc.SelectList sl = new
SelectList(db.Categories.Select(x => x.CategoryName));
    ViewBag.Values = sl;

    // pronalazenje prve kategorije iz kolekcije
    // koja ima odabran naziv
    Categories odabrano = db.Categories.Where(x =>
x.CategoryName == categoryName).FirstOrDefault();
    if (odabrano != null)
    {
        // izdvajanje id za kategoriju
        int id = odabrano.CategoryID;
```

```
        // izdvajanje svih proizvoda čija je kategorija
izdvojeni id
        rez_Products = db.Products.Where(x => x.CategoryID
== id).ToList();
    }

    return View(rez_Products);
}
```

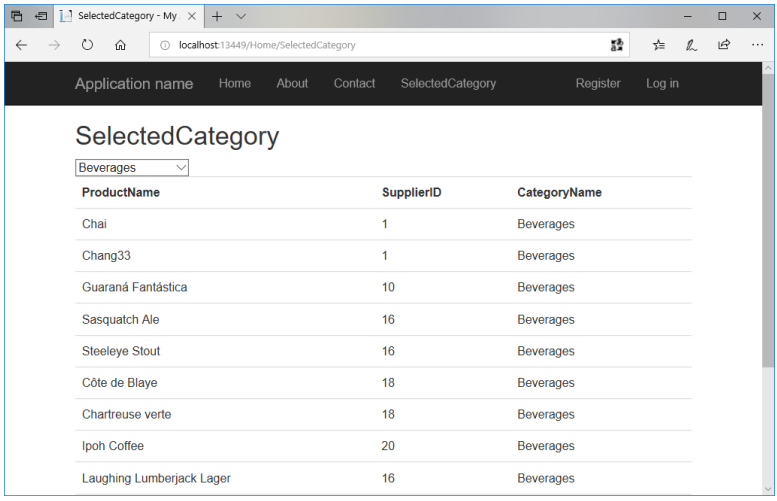
Treba obratiti pažnju na još jedan detalj. Ukoliko se ova metoda vezuje za meni u aplikaciji, onda se mora voditi računa šta se šalje kao argument. Ukoliko se to posebno ne naglasi dobija se vrednost **null**, koja u našem slučaju ne utiče na rezultate, pa je nismo morali posebno obrađivali u kodu.

Sada je važno dopuniti pogled za prikaz ovih podataka. Model smo prilagodili da prima `IEnumerable` tipove i tako omogućili prihvatanje svih listi i nizova u pogledu:

```
@model IEnumerable<mvcWeb1.Models.Products>
@{
    ViewBag.Title = "SelectedCategory";
}

<h2>SelectedCategory</h2>

@using (Html.BeginForm("SelectedCategory", "Home"))
{
    // . . . već navedeno . . .
}

<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model =>
model.ProductName)
        </th>
        <th>
            @Html.DisplayNameFor(model =>
model.SupplierID)
        </th>
        <th>
```

```
            @Html.DisplayNameFor(model =>
model.Categories.CategoryName)
        </th>
    </tr>

    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem =>
item.ProductName)
            </td>
            <td>
                @Html.DisplayFor(modelItem =>
item.SupplierID)
            </td>
            <td>
                @Html.DisplayFor(modelItem =>
item.Categories.CategoryName)
            </td>
        </tr>
    }

</table>
```

Za ovakav pogled dobija se sledeći prikaz.



Slika 11.24. Pogled za filtriranje proizvoda po kategoriji

Pitanja i zadaci za proveru znanja

1. Kreirajte vašu MVC veb aplikaciju i testirajte njen rad bez dodavanja vašeg koda.
2. Napravite sopstveni model u folder u Models kao ADO.NET Entity Data Model.
3. Koristeći Scaffolding napravite kontroler i odgovarajuće poglede za rad sa entitetom `Employees`. Ispitajte opciju `Use a layout page` u formi `Add Controller` pri dodavanju kontrolera.
4. Promeniti tabelu dodavanjem novog polja, po sopstvenom izboru. Šta je sve potrebno promeniti i gde kako bi aplikacija radila sa novim podacima?
5. Napisati sopstvenu prikaz i metodu kontrolera za unos podataka o zaposlenom. Kako se mogu podaci proslediti kontroleru?
6. Šta su to filteri?
7. Objasnite i pokažite primenu filtera `Authorize`.
8. Objasnite i pokažite primenu filtera `OutputCache`.
9. Objasnite i pokažite primenu filtera izuzetaka.
10. Napišite primer jednog pogleda koji će sadržati formu sa padajućom listom stavki tabele Products. Napisati metodu koja će prihvatati odabranu stavku. Na koji način se definiše slanje odabrane stavke.
11. Za odabrani proizvod prikazati sve narudžbenice uz još neki uslov po vašem izboru.

12. *Code first*

Tehnologija *Code First* omogućava da se ceo sajt, sa pratećom bazom u pozadini, formira u C# kodu uključujući: opise tabela, relacije i ograničenja. Dakle, opis modela se ne vrši na osnovu baze već se baza i tabele kreiraju na osnovu modela .NET klasa koristeći programski jezik C#.

Ovo poglavlje je posvećeno ovoj tehnologiji i kroz konkretan primer pokazujemo tehniku rada u razvojnom okruženju.

Osnovni oblik modela definiše se korišćenjem konvencija. Konvencije su pravila koja se koriste za automatsko konfigurisanje konceptualnog modela zasnovanog na definicijama klase. Konvencije su definisane u imenskom prostoru klasa System.Data.Entity.ModelConfiguration.Conventions. Dalje, konfigurisanje modela vrši se primenom anotacija podataka ili u kodu. Prednost se daje konfiguraciji kroz API metode praćene anotacijama podataka, a zatim konvencijama.

Razvoj

Primena tehnologije *Code First* zasniva se na formiranju modela u kodu pisanjem .NET Framework klasa koje definišu konceptualni model. Pored definisanja klasa entiteta, potrebno je da objekat `DbContext` dobije informacije koje tipove treba da uključi u model. To se radi definišući kontekstualnu klasu koja je izvedena iz `DbContext`-a i postavljanjem

`DbSet` svojstva za tipove za koji su deo modela. Na ovaj način će se uključiti tipovi ali i automatski se ubacuju i svi referentni tipovi, čak i ako su navedeni tipovi definisani u drugom skupu.

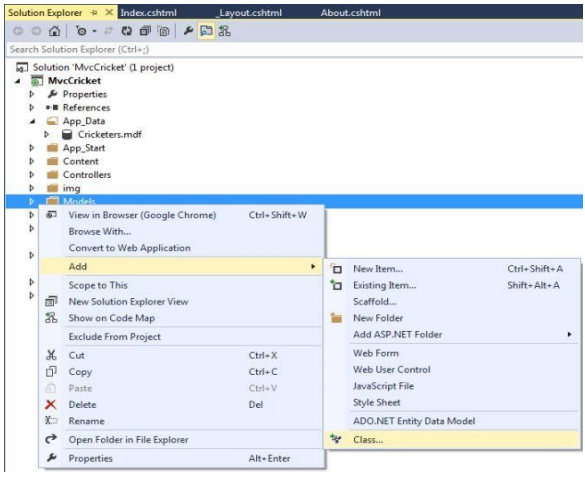
Ako korišćeni tipovi učestvuju u hijerarhiji klasa tj. nasleđivanju, dovoljno je definisati svojstvo `DbSet` za baznu klasu, a izvedeni tipovi će biti automatski uključeni.

Model

Ovde ćemo pokazati kako se dodaju klase za rad sa modelima i konekcijskim stringom. Konekcijskim stringom se obezbeđuje potrebni parametri za perzistencija podataka modela, konkretnije, pomoću konekcijskog stringa se omogućava rad sa bazom podataka.

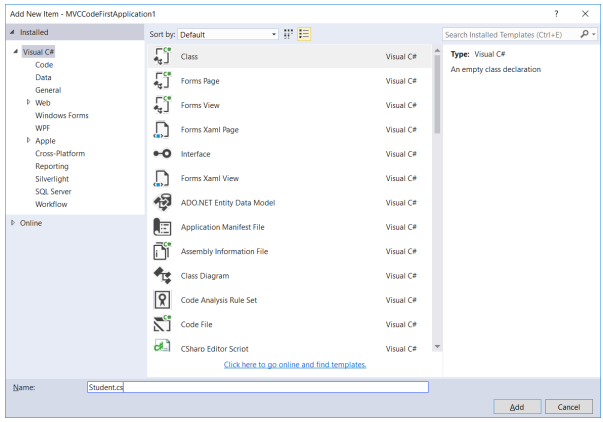
Kreiranje klase modela

Korak 1: Desni-Klik na folder `Model` i označiti `Add` zatim `Class`.



Slika 12.1. Dodavanje nove stavke u projekat i izbor klase

Korak 2: Unesite ime klase. Ovo je ujedno ime fajla sa ekstenzijom cs.



Slika 12.2. Dodavanje klase Student projektu

Kao rezultat dodavanja klase, formiran je novi fajl i programeru se automatski otvara editor za uređivanje koda klase. U našem primeru dobijen je kod:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace MVCCodeFirstApplication1.Models
{
    public class Student
    {

    }
}
```

Klasa `Student` predstavljaće stavke u bazi. Tačnije, svaka instanca klase `Student` tj. svaki objekat `Student` predstavlja takođe i jedan red u tabeli, a svako svojstvo klase predstavlja kolonu u tabeli. Dodavanjem svojstva ovoj klasi formira se opis kolona buduće tabele.

Više objekata `Student` predstavlja se kao novi objekat. Slično kao što više redova zajedno čine tabelu. Pošto je ovaj objekat analogan objektu koji odgovara skupu entiteta u EDM modelu, koristi se generički tip `DbSet<Student>`. A klasa koja opisuje ceo model koji sadrži jedan ili više različitih kolekcija drugih entiteta izvedena je iz klase **DbContext** iz prostora imena **System.Data.Entity**. Klase ovog prostora imena pripadaju **EntityFramework.dll** biblioteci. Ukoliko slučajno nemate tu biblioteku u okviru projekta možete je lako uključiti koristeći opciju `Manage NuGet Packages`. Ceo kod bi bio:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Configuration;

namespace MVCCodeFirstApplication1.Models
{
    public class Student
    {
        public int ID { get; set; }
        public string index { get; set; }
        public string ime { get; set; }
        public int semestar { get; set; }
    }

    public class StudentDbContext:DbContext
    {
        public DbSet<Student> Students { get; set; }
    }
}
```

Korišćena klasa modela koja je izvedena iz `DbContext` može da uključi više modela i tako omogući rad sa podacima tj. čuvanje podataka u jednom kontekstu rada:

```
public class StudijeDbContext:DbContext
{
    public DbSet<Student> Students { get; set; }
    public DbSet<Student> Osobe { get; set; }
    public DbSet<Student> StudijskiProgrami { get; set; }
}
```

Rad sa podacima u klasi **DbContext** zasniva se na radu sa objektima kolekcije, a pri tome se koriste Linq upiti. Na primer:

```
var db = new StudijeDb();
var stud = db.Students.Select(s => s).OrderBy(s => s.ime);
```

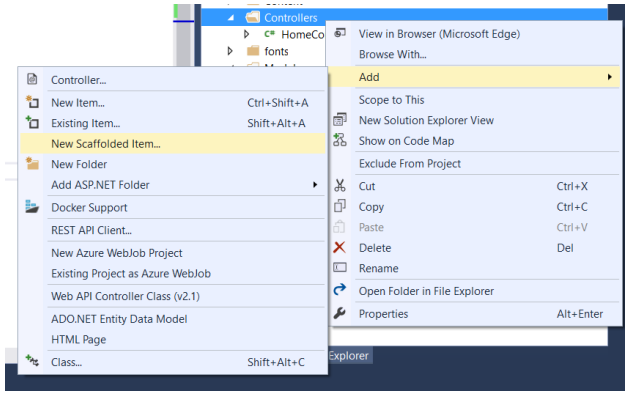
ili

```
var stud = from s in db.Students
                orderby s.ime ascending
                select s;
```

Generisanje koda - *Scaffold*

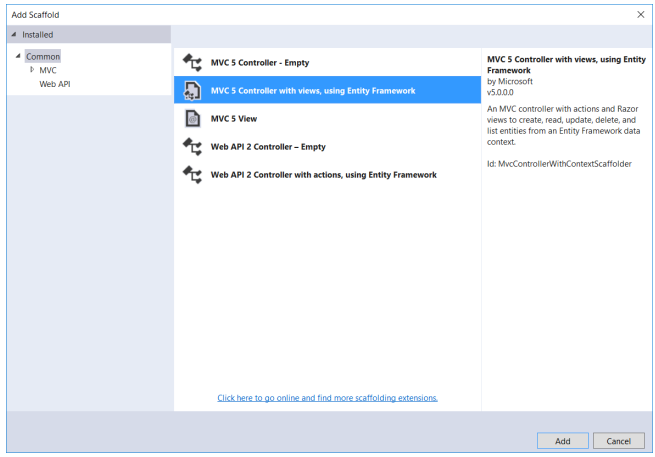
Ako se dva entiteta modela naprave metode za ažuriranje i pregled zapazićemo da postoji velika sličnost u kodu za ta dva slučaja. IDE okruženje nudi opciju da se na osnovu nekog modela šablonski kreiraju odgovarajući kontroleri i pogledi. Ovaj alat u okviru Visual Studio IDE okruženja naziva se *Scaffold,* a njegovom primenom se automatizuje postupak početnog generisanja kontrolera i pogleda. Pogledajmo kako izgleda kreiranje kontrolera i odgovarajućih pogleda.

Korak 1. Desnim klikom na stavku **Controllers** u prozoru **Solution Explorer** izaberemo opciju **Add**, a zatim i **New Scafolded Item**.



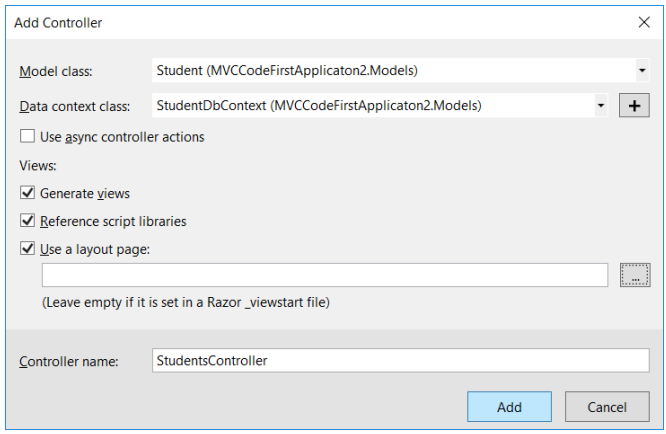
Slika 12.3. Dodavanje stavke generisane `Scaffold` alatom

Korak 2. Na pomoćnom ekranu biramo opciju automatizovanog kreiranja. U našem slučaju kreiramo kontroler sa svim pratećim operacijama za ažuriranje (CRUD operacije).



Slika 12.4. Izbor generisanja kontrolera i pogleda

Korak 3. Bira se klasa modela od onih koje već postoje u projektu. Pri tome se definišu i neke ranije opisane opcije koje se tiču pogleda.



Slika 12.5. Definisanje kontrolera

Kreiranje nove konekcije

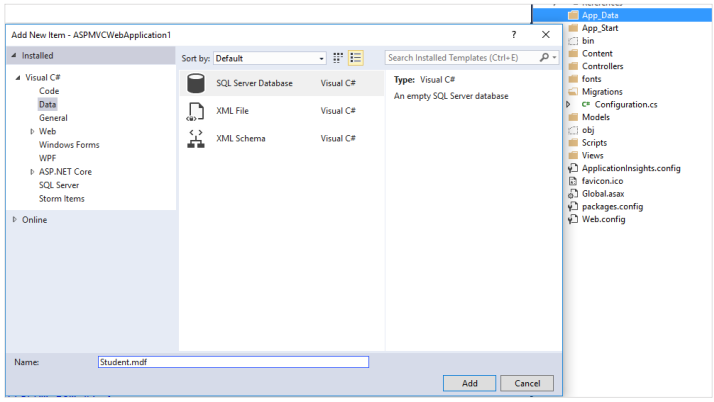
Entity Framework automatski obezbeđuje konekciju do baze i koristi je. Podrazumevana baza je LocalDB. *Entity Framework* i ASP MVC zajedno omogućavaju da rad sa bazom bude što više nezavisan od same baze. Na početku recimo da se podešavanja u vezi baze, tačnije parametri konekcije do baze nalaze u konfiguracionom fajlu `Web.Config`.

SQL Server Express i LocalDB

SQL Server Express sadrži LocalDB bazu. To je veoma mala, lagana baza koja se automatski startuje na zahtev u korisničkom modu. SQL Server Express poseduje poseban izvršni mod u kome se LocalDB izvršava i olakšava rad sa podacima kao što su *.mdf* fajlovi. Fajlovi ekstenzije ***.mdf*** čuvaju se u folderu `App_data` u aplikaciji.

Kasnije, po potrebi, lako je prebaciti tj. migrirati vašu LocalDB bazu na SQL Server ili SQL Azure. Ova baza je podrazumevano instalirana u Visual Studio.

Dakle, prvo dodajte mdf fajl `Student.mdf` u `App_Data` folder:



Slika 12.6. Dodavanje mdf fajla projektu za rad sa LocalDB

Zatim, kreirajte **konekcijski string**. Konekcijski string sadrži sve parametre za pristup do podataka u vidu stringa. Kreiranje konekcijskog stringa može da se uradi:

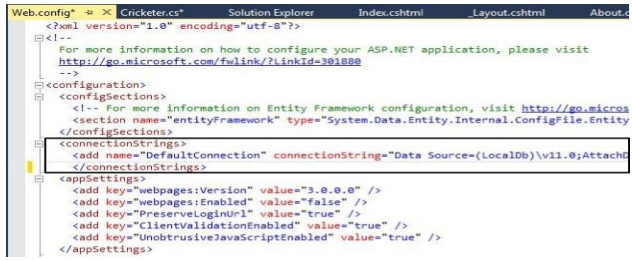
1) direktno u konfiguracioni fajl ili
2) preko pomoćnih formi za kreiranje ovog stringa.

Direktno dodavanje konekcijskog stringa

1) Dodavanje konekcije izvodite u sledećoj proceduri.

Korak 1: U `Solution Explorer` prozoru otvorite **Web.Config** fajl.

Korak 2: Nađite ili dodajte novi `<connectionStrings>` element.



Slika 12.7. Izdvojena sekcija connectionStrings u Web.Config fajlu

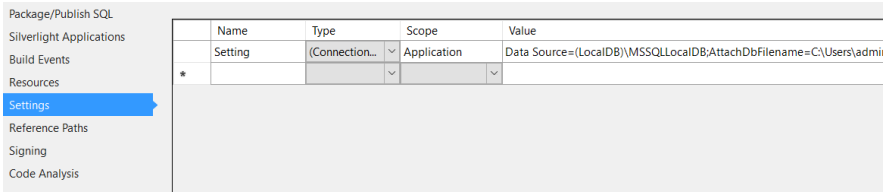
Korak 3: Dodajte sledeću vrednost za konekcijski string elementu <connectionStrings> nakon prethodnog konekcijskog stringa, ukoliko postoji:

```
<add name="StudentDBContext" connectionString="Data
Source=(LocalDb)\MSSQLLocalDB;AttachDbFileName=|DataDirectory
|\Student.mdf;Integrated Security=True"
providerName="System.Data.SqlClient" />
```

Važno: Naziv konekcijskog stringa mora odgovarati nazivu klase **DbContext** (klasa `StudentDbContext`).

Dodavanje konekcijskog stringa koristeći IDE okruženje

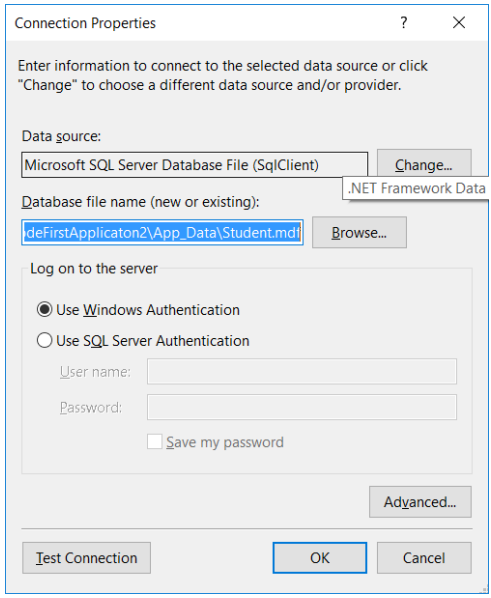
Dodavanje konekcijskog stringa možete izvesti i koristeći pomoćni prozor za svojstva projekta. Konekcijski string se definiše u delu **Settings**.



Slika 12.8. Kartica Settings i deo za podešavanje konekcijskog stringa

Izbor se vrši u više koraka. Najpre, klikom na malo dugme u ćeliji Value aktivirate pomoćni dijalog za definisanje konekcije.

Kao na slici ispod birate izvor podataka tj. `DataSource` tj. kao **Microsoft SQL Server Database File (SqlClient)**:



Slika 12.9. Podešavanje svojstava konekcije

Zatim je jako važno da uradite lociranje fajla koji ste kreirali u prethodnom koraku, a nalazi se u App_Data folderu. Ovo se postiže klikom na dugme `Browse`...

Kada je konekcijski string spreman onda je sve spremno da se testiraju odgovarajući kontroleri i pogledi odnosno sve CRUD operacije.

Pitanja i zadaci za proveru znanja

1. Objasnite osnovni koncept *Code first* aplikacija.
2. Napravite sopstveni model u folderu `Models` pod nazivom `Radnik`.
   Napravite i konteksni klasu koja obuhvata prethodni model `Radnik`.
3. Generišite kontroler i poglede za model `Radnik`.
4. Kako se obezbeđuje čuvanje podataka modela?

5. Objasnite promenu konekcijskog stringa preko konfiguracionog fajla i preko prozora za podešavanje.
6. Testirajte rad svih generisanih pogleda.
7. Dodajte novi model za `Odeljenje`, kao i vezni entitet između `Radnika` i `Odeljenja`, `RadnikOdeljenje`.
8. Generišite sve poglede po automatizmu i povežite ih sa menijem u aplikaciji.
9. Proverite da li se podaci čuvaju u bazi. Pokušajte snimanje sa lokalnom i spoljnom bazom.

.

13. Održavanje

Razvoj jedne aplikacije praćen je promenama na svim komponentama kao što su klase ili resursi. Promene u modelu aplikacija u *Code First* tehnologiji znače promene u klasi modela, kao što su: dodavanje, izmenu ili brisanje svojstava klase odnosno odgovarajućih polja u tabelama koje prate modele u bazi. Pri ovim promenama neophodno je obezbediti ne samo izmene u modelu odnosno bazi, već istovremeno treba da postoji mogućnost vraćanja na prethodno stanje, dakle praćenje promena.

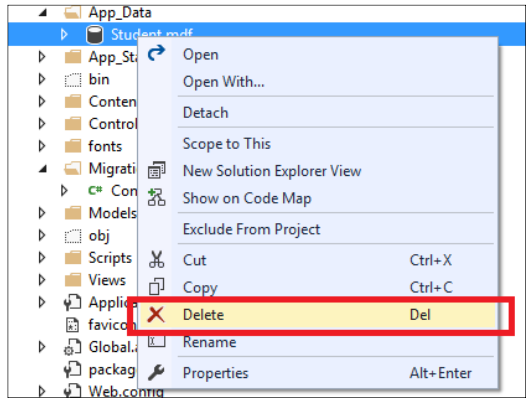
Ovo poglavlje je posvećeno upoznavanju sa tehnikama održavanja koje obuhvataju i promene na modelima, podacima ali i funkcionalnostima. Ove tehnike su poznate kao migracije.

Migracija

Dakle, zbog toga što promena na modelu predstavlja važan korak u razvoju aplikacije koji zahteva više izmena uz mogućnost čuvanja prethodnih podataka, prelaz na novo stanje modela predstavlja zaseban postupak koji nazivamo **migracija**.

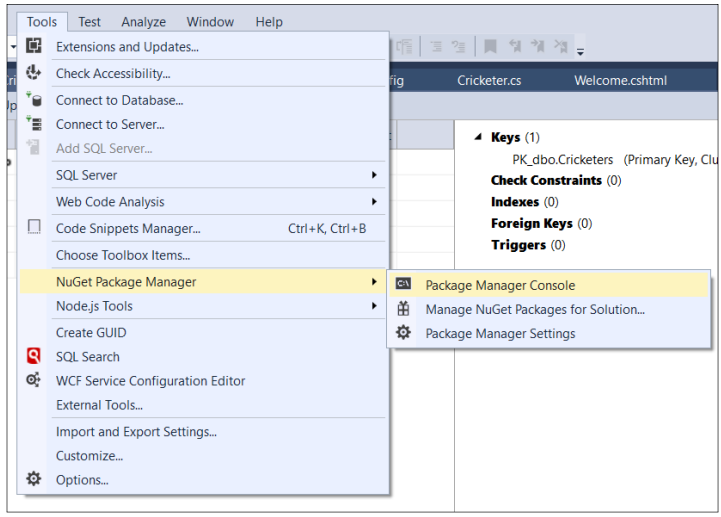
Postupak migracije prikazujemo kroz nekoliko koraka.

Korak 1: Ukoliko koristite prethodni primer u kome smo koristili postojeću bazu, sada je potrebno da je obrišete. Postojeći `mdf` fajl lociran je u `App_Data` folderu što se vidi kroz prozor `Solution Explorer`.



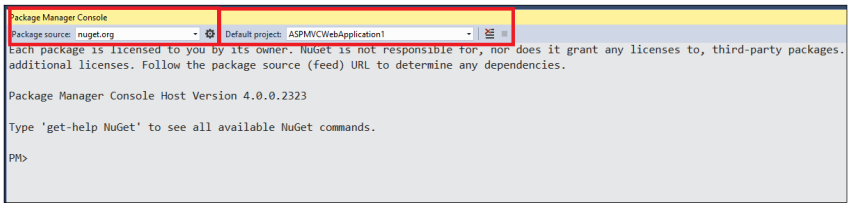
Slika 13.1. Označavanje lokalne baze za brisanje

Korak 2: Otvorite konzolu `Package Manager Console`, kao na slici:



Slika 13.2. Otvaranje konzole za rad sa migracijama

Otvaranje ove konzole znači zapravo otvaranje posebnog prozora u VS okruženju. Taj prozor se podrazumevano nalazi na dnu okruženja, a neki inicijalni prikaz dat je na slici ispod.



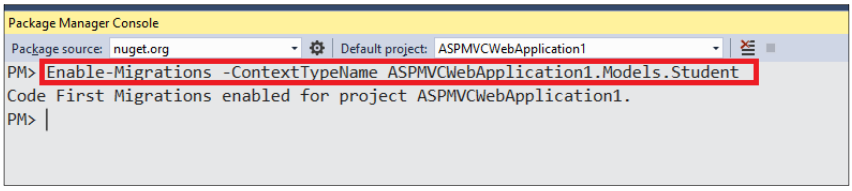
Slika 13.3. `Package Manager Console`

Korak 3: Unesite sledeću komandu, a zatim Enter:

```
PM> Enable-Migrations -ContextTypeName
ASPMVCWebApplication1.Models.Student
```

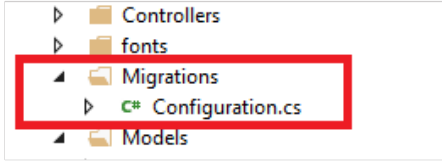
U opštem slučaju komanda je oblika:

```
Enable-Migrations -ContextTypeName punoImeKlaseDBContext
```



Slika 13.4. Odgovor na komandu `Enable-Migration`

Nakon ovog kreira se folder `Migrations` i u njemu fajl `Configurations.cs`, Pogledajte sliku:



Slika 13.5. Novi folder i fajl nakon omogućavanja migracija

Korak 4: Kao što znate, lokalna baza tj. **mdf** fajl je obrisan. Početni podaci u bazu se snimaju prvi put pri pokretanju aplikacije. Tačnije, prvi put se kreiraju tabele, ograničenja i veze između tabela i dodaju se početni podaci. Ovo je ujedno i prva migracija koja se definiše. Za prvu migraciju već je pripremljena klasa.

Otvorite sada fajl `Configuration.cs`. U ovoj klasi se metoda `Seed` koja sadrži kod kojim se formira početni sadržaj baze. Sada ćemo videti kako da ga prilagodimo sopstvenoj potrebi.

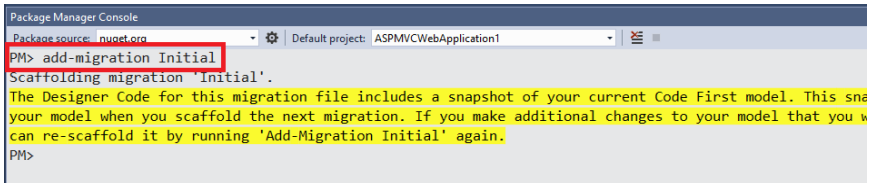
Osnovni argument ove metode je ...DbContext objekat osnovnog objekta, u našem slučaju Student objekata. U kodu se vrši ručno dodavanja početnih vrednosti u bazi.

```
protected override void
Seed(MVCCodeFirstApplicaton2.Models.StudentDbContext
context){
    context.Students.AddOrUpdate(x => x.ID,
        new Models.Student {
            ID = 1,
            ime = "Jova",
            index = "NRT-55/55",
            semestar = 1
        },
        new Models.Student
        {
            ID = 2,
            ime = "Mira",
            index = "NRT-44/44",
            semestar = 1
        },
        new Models.Student
        {
            ID = 3,
            ime = "Aco",
            index = "NRT-33/33",
            semestar = 2
        }
    );
```

Korak 6: Sada prevedite vašu aplikaciju (Shift+F6).

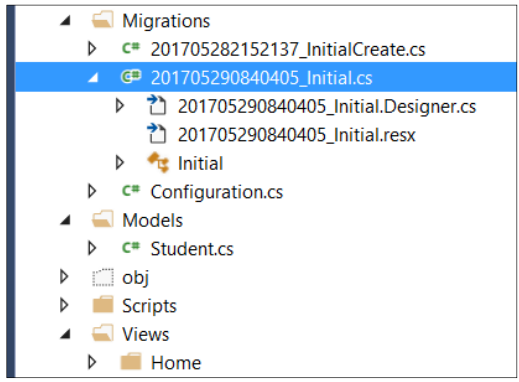
Korak 7: U ovom koraku vrši se pokretanje inicijalne migracije. U otvorenom prozoru `Package Manager Console` unesite sledeću komandu:

add-migration Initial



Slika 13.6. Dodavanje inicijalne migracije

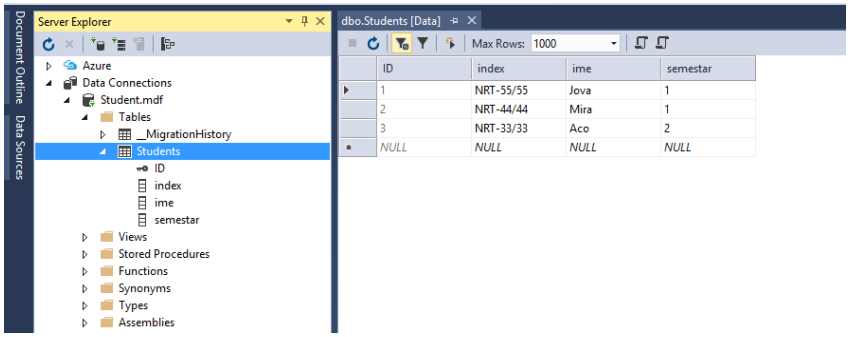
Ova komanda kreira novu klasu, u posebnom fajlu, takođe u folderu `Migration`. Nova klasa dobija ime po šablonu "(DataStamp)_initial.cs code". Ona omogućava kreiranju šeme u bazi podataka i početne podatke. Konkretno u našem slučaju, fajl sa kodom implementira tabelu Student u bazi Student.



Slika 13.7. Formirani fajlovi za inicijalnu migraciju

Korak 8: Konačno postavljanje početnih podataka vrši se takođe preko konzole. Unesite sledeću komandu u vaš Package Manager Console:

update-database



Slika 13.8. Prikaz tabele nakon postavljanja početnih podataka

Dodavanje svojstva

U klasi Osoba postoje 4 svojstva uključujući i svojstvo koje je ID. Kako se dodaje novo svojstvo, na primer `datumUpisa`? Procedura je sledeća

Korak 1: Otvoriti fajl Student.cs i dodati svojstvo: `DateTime datumUpisa`:

```
public class Student
{
    // * * *
     // postojeća svojstva
    // * * *

     public DateTime datumUpisa { get; set; }
}
```

Korak 2: Kompajlirajte projekat: "Ctrl+Shift+B".

Ažuriranje stranica

Mada je kompajler uspešno preveo aplikaciju, novo svojstvo nije dodato u ranije kreirane fajlove. Na početku uradimo izmene na fajlovima: Index.cshtml i Create.cshtml.

Index.cshtml

Ubacite kod Index.cshtml:

```
@model IEnumerable<ASPMVCWebApplication1.Models.Student>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.index)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ime)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.semestar)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.datumUpisa)
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.index)
        </td>
```

```
        <td>
            @Html.DisplayFor(modelItem => item.ime)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.semestar)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.datumUpisa)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID
}) |
            @Html.ActionLink("Details", "Details", new {
id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new {
id=item.ID })
        </td>
    </tr>
}

</table>
```

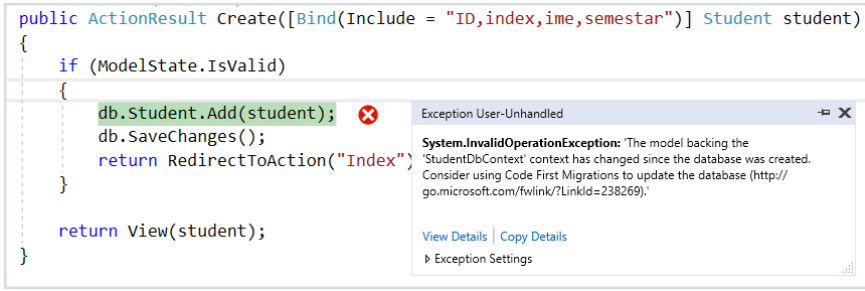
Create.cshtml

Na sličan način dopunite i fajl Create.cshtml sledećim kodom na odgovarajućem mestu:

```
<div class="form-group">
     @Html.LabelFor(model => model.datumUpisa,
htmlAttributes: new { @class = "control-label col-md-2" })
     <div class="col-md-10">
          @Html.EditorFor(model => model.datumUpisa, new {
htmlAttributes = new { @class = "form-control" } })
          @Html.ValidationMessageFor(model =>
model.datumUpisa, "", new { @class = "text-danger" })
     </div>
 </div>
```

Korak 4: Pošto ste uradili izmene pokrenite vašu veb aplikaciju.

Note: Nakon prikaza liste dobija se greška:



Slika 13.9. Prikaz greške nakon promene klase

Rešenje je u modifikaciji baze odnosno promeni Entity Data Modela. Zato je potrebno uraditi **Code First Migration**. Ažurirajte vaš **Seed** method u **Configurtion.cs** fajlu i dodajte polje **datumUpisa** za svaki Student objekat. Na primer za prvi bi bilo:

```
new Models.Student
{
    ID = 1,
    ime = "Jova",
    index = "NRT-55/55",
    semestar = 1,
    datumUpisa = new DateTime(2000,10,1)
},
```

Korak 5: Kompajlirate vaš projekat pritiskom na "Ctrl+Shift+B".

Korak 6: Zatim ponovo uraditi migraciju: otvoriti Package Manager Console i unesite komandu:

```
add-migration Student
```

Pošto je komanda završena, nova klasa **DbMigration** otvara se automatski. U ovom kodu se vidi nova kolona koja je dodata:

```
namespace ASPMVCWebApplication1.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class Student : DbMigration
    {
        public override void Up()
```

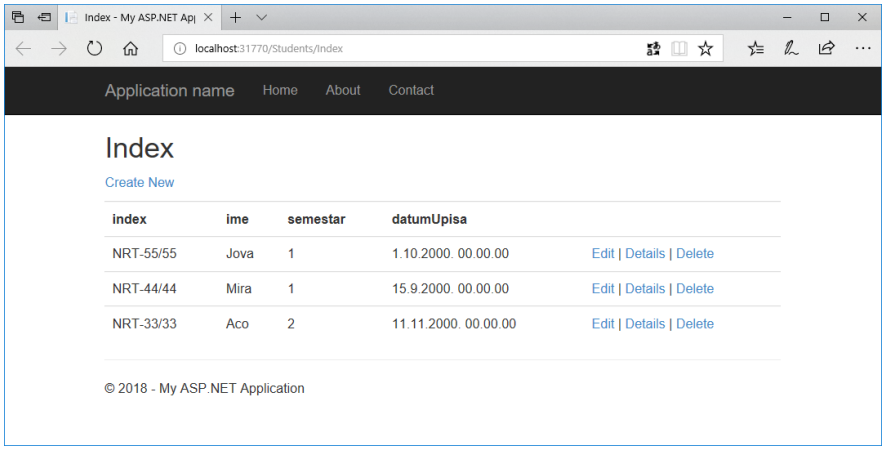
```
        {
            AddColumn("dbo.Students", "datumUpisa", c =>
c.DateTime(nullable: false));
        }

        public override void Down()
        {
            DropColumn("dbo.Students", "datumUpisa");
        }
    }
}
```

Korak 7: Zatim unesite sledeću komandu u prozor Package Manager Console:

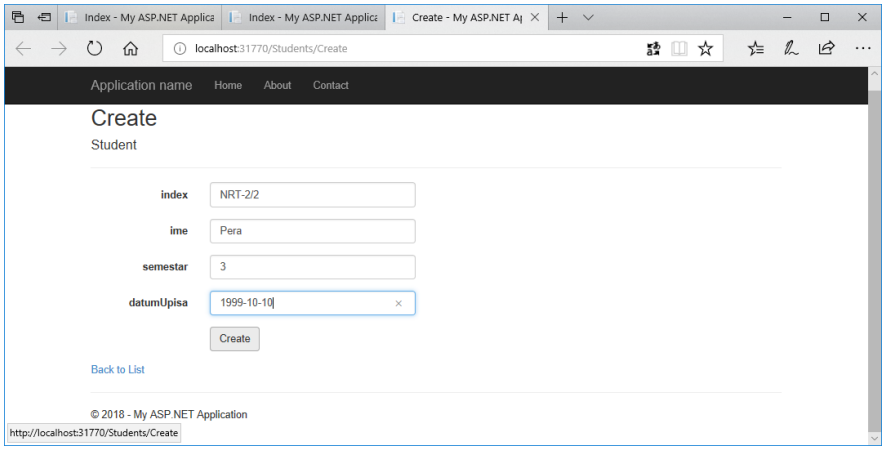
```
update-database
```

Korak 8: Ukoliko se kod metoda `Create` odnosno `Edit` koristi `[Bind(Include = "ID,index,ime,semestar ")]` potrebno je dodati polje `datumUpisa`.

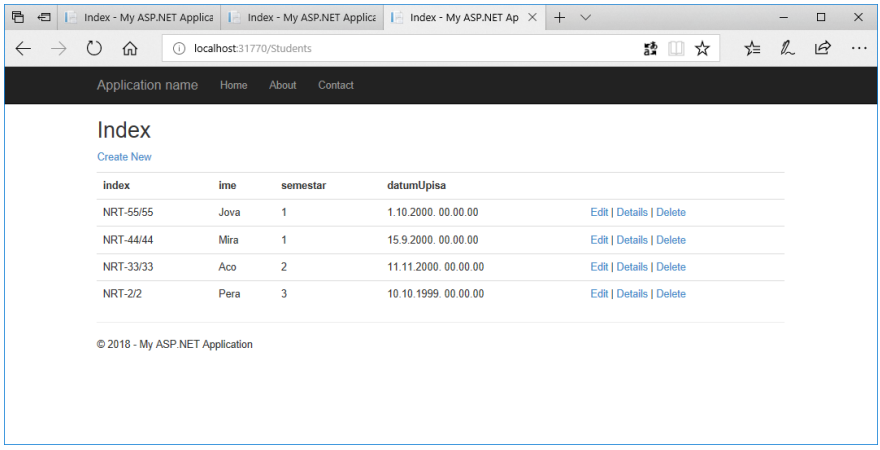
Korak 9: Pokrenite sada aplikaciju i testirajte `Index` i `Create`:



Slika 13.10. Pogled na `Index` stranicu kontrolera `Student`



Slika 13.11. Pogled na `Create` stranicu kontrolera `Student`



Slika 13.12. Pogled na `Index` stranicu nakon kreiranja novog objekta `Student`

Details.cshtml i Edit.cshtml

Korak 1: Označiti `Details.cshtml` fajl u prozoru `Solution Explorer`.

Korak 2: Promenite kod na sledeći po uzoru na postojeći:

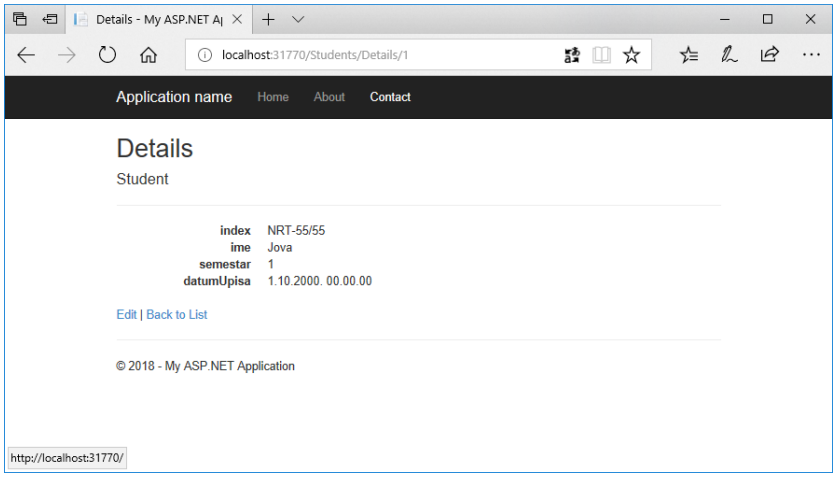
```
@model ASPMVCWebApplication1.Models.Student

@{
    ViewBag.Title = "Details";
}
```

```
<h2>Details</h2>

<div>
    <h4>Student</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.index)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.index)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.ime)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.ime)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.semestar)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.semestar)
        </dd>

        <dt>
            @Html.DisplayNameFor(model =>
model.datumUpisa)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.datumUpisa)
        </dd>
    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID
}) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

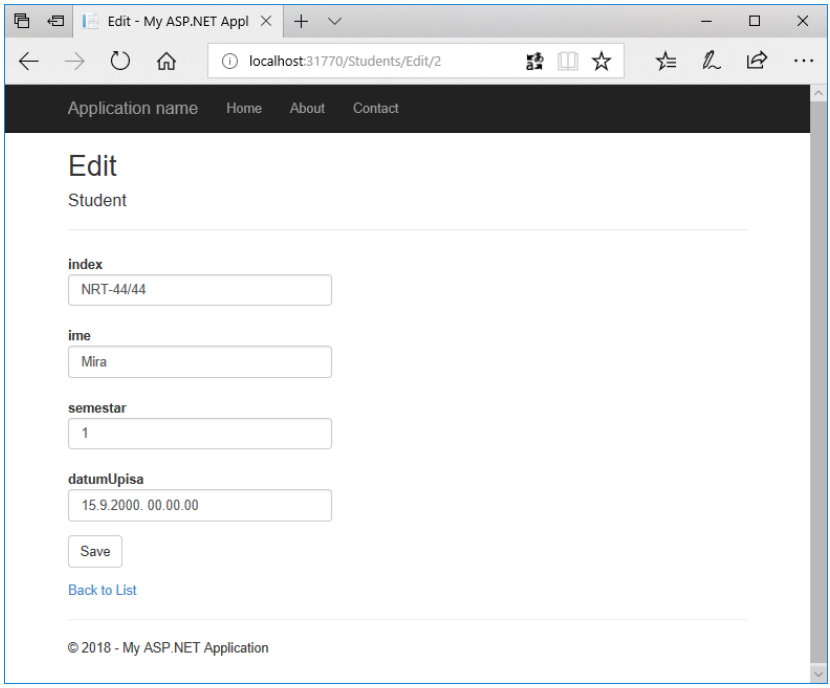
U fajlu **Edit.cshtml**, na sličan način, dakle po analogiji sa već postojećim kodom za postojeća svojstva vršimo dodavanje dela koda za novo svojstvo.

```
<div class="form-group">
    @Html.LabelFor(model => model.datumUpisa,
htmlAttributes: new { @class = "control-label col-md-2" })
    <div class="col-md-10">
        @Html.EditorFor(model => model.datumUpisa, new {
htmlAttributes = new { @class = "form-control" } })
        @Html.ValidationMessageFor(model =>
model.datumUpisa, "", new { @class = "text-danger" })
    </div>
</div>
```

Korak 3: Pokrenuti aplikaciju i pogledajte stranicu **Details**.



Slika 13.13. Pogled na novu stranicu Details

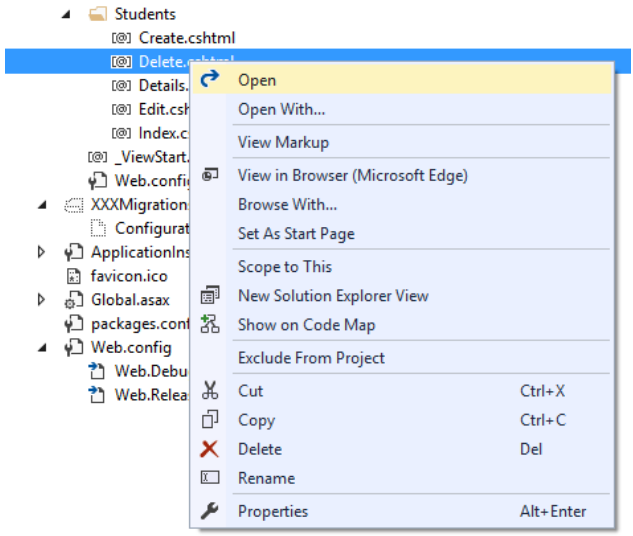


Slika 13.14. Pogled na novu stranicu Edit

Delete.cshtml

Pre brisanja, a nakon klika na `Delete` link, otvara se stranica sa svim detaljima o objektu `Student` i jednim dugmetom za brisanje. Brisanje je osetljiva operacija i zato se traži provera tj. potvrda od korisnika. Zbog prikaza detalja potrebno je da ažuriramo i fajl `Delete.cshtml`.

Korak 1: Označiti `Delete.cshtml` fajl u prozoru `Solution Explorer`



Slika 13.15. Otvaranje stranice pogleda Delete.cshtml preko konteksnog menija

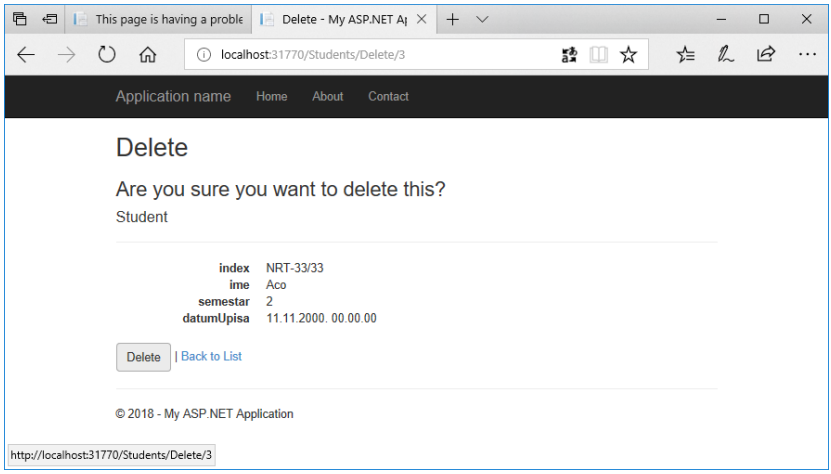
Korak 2: Promenimo kod na sledeći način:

```
@model ASPMVCWebApplication1.Models.Student

@{
    ViewBag.Title = "Delete";
}

<h2>Delete</h2>

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

        <dt>
            @Html.DisplayNameFor(model => model.ime)
        </dt>
        <dd>
```

```
            @Html.DisplayFor(model => model.ime)
        </dd>

        <dt>
            @Html.DisplayNameFor(model => model.semestar)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.semestar)
        </dd>

        <dt>
            @Html.DisplayNameFor(model =>
model.datumUpisa)
        </dt>
        <dd>
             @Html.DisplayFor(model =>
model.datumUpisa)
         </dd>
    </dl>

    @using (Html.BeginForm()) {
        @Html.AntiForgeryToken()

        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn
btn-default" /> |
            @Html.ActionLink("Back to List", "Index")
        </div>
    }
</div>
```

Korak 3: Pokrenite aplikaciju.

Korak 4: Klik na link Delete i proverite izgled.



Slika 13.16. Pogled na stranicu Delete

Pretraga

Sada ćemo pogledati kako se može na sajtu obezbediti pretraga podataka. U konkretnom slučaju uradićemo pretragu na dva načina:

1. po polju *ime*
2. po polju *semestar*

Pretraga po stringu

U ovoj pretrazi formira se lista objekata Student za koje važi da polje ime sadrži zadati string. Pretragu ćemo realizovati modifikacijom metode Index i odgovarajućeg pogleda.

Korak 1: Otvoriti kontroler `Students` i modifikujte kod dodajući argument metodi.

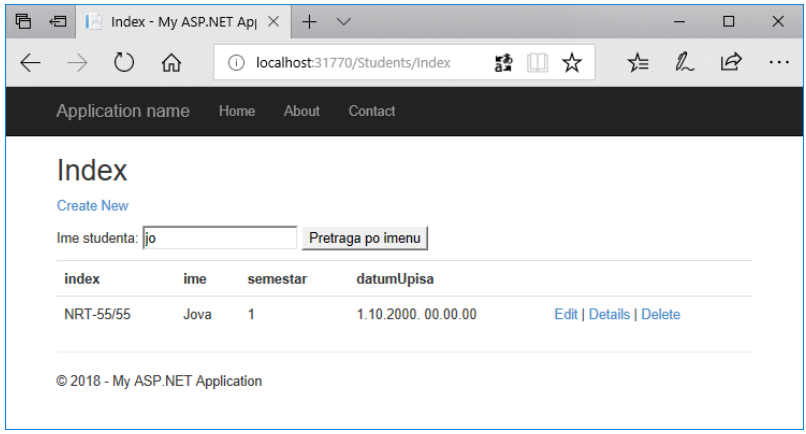
```
public ActionResult Index(string searchString)
{
    var students = from st in db.Student select st;
```

```
    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(c =>
c.ime.Contains(searchString));
    }

    return View(students);
}
```

Korak 2: Modifikujte pogled: `Views\Students\Index.cshtml`. Dodajte polje za pretragu. Polje za pretragu je TextBox, a vrednost se šalje uz odgovarajuće ime polja `"searchString"`. Napomena: Svako slanje podataka obvlja se ili preko forme ili preko ajax metoda. Ovde se prikazuje upotrebu formi preko Razor sintakse.

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm())
{
    Ime studenta: @Html.TextBox("searchString")
    <input type="submit" value="Pretraga po imenu" /> <br />
}
```

Korak 3: Pokrenite aplikaciju.



Slika 13.17. Pregraga po polju ime

Višestruka pretraga

Sada ćemo uraditi pretragu i po polju semestar koje je po tipu ceo broj. Pre pretrage potrebno je definisati izbor vrednosti za pretragu. Zbog toga se pre pretrage čitaju sve različite vrednosti semestara na osnovu koji se formira lista. Dobijena lista se koristi za prikaz padajuće liste za izbor semestra za pretragu.

```
public ActionResult Index(string searchString, string sem)
{
    var svisemestri = from st in db.Student orderby
st.semestar select st.semestar.ToString();

    var semestri = new List<string>();
    semestri.AddRange(svisemestri.Distinct());

    ViewBag.ListaSemestara = semestri;

    var students = from st in db.Student select st;

    if (!String.IsNullOrEmpty(searchString))
    {
        students = students.Where(s =>
s.ime.Contains(searchString));
    }
    if (!string.IsNullOrEmpty(sem))
    {
        int isem = int.Parse(sem);
        students = students.Where(s => s.semestar ==
isem);
    }

    return View(students);
}
```

Ovoj metodi prosleđuju se dva parametra. Treba imati na umu da se ova metoda kontrolera poziva pre prvog prikaza stranice, odnosno da pre prvog prikaza stranice imamo formiranje liste

`ViewBag.ListaSemestara` koja sadrži listu semestara koji postoje u listi objekata Student.

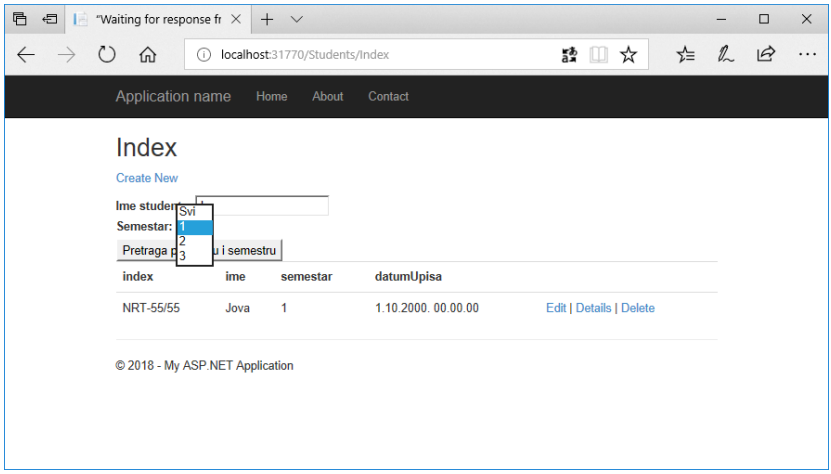
Korak 2: Da bi pretraga bila dostupna korisniku mora postojati odgovarajući dizajn odnosno kontrole na stranici za pretragu. Dakle, u prikazu treba dodati novo polje za izbor semestra. Za to smo se već dogovorili da ćemo koristiti jednu padajuću listu. Dakle, otvoriti fajl "Views\Students\Index.cshtml" i modifikovati kod:

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
@using (Html.BeginForm())
{
    <label for="searchString">Ime studenta:</label>
    @Html.TextBox("searchString")<br />

    <label for="ListaSemestara">Semestar:</label>
    @Html.DropDownList("sem", new SelectList(
ViewBag.ListaSemestara ), "Svi")<br />

    <input type="submit" value="Pretraga po imenu i
semestru" />
}
```

U gornjem kodu dodata je padajuća lista tj. DropDownList uz raniju kontrolu TextBox za pretragu. Pošto se kontroler svakako koristi pri prikazu stranice, a tada su ulazni argumenti null, onda se ista metoda koristi za popunjavanje padajuće liste. Tačnije ovo se radi svaki put pri prikazu.

Korak 3: Pokrenite aplikaciju.

Označiti određenu stavku u padajućoj listi i Klik na "Search".



Slika 13.18. Pogled na modifikovanu stranicu Index

Validacija

Validacija predstavlja proveru vrednosti koju korisnik unosi. Ta provera se obavlja pre slanja podataka serverskoj strani. Validacija se primenjuju uključujući prostor imena:

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

U ASP.NET MVC aplikacijama validacije su definisane u klasama Modela i primenjuju se u celoj aplikaciji. Entity Framework Code First pristup i MVC su osnova za validaciju.

Anotacije

Sada ćemo dodati određenu validacionu logiku našoj MVC aplikaciji, prateći korak po korak.

Korak 1: Otvoriti fajl `Student.cs`.

Korak 2: Dodajte imenski prostor `DataAnnotations`:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;
```

Korak 3: Izmenite vaš kod sa sledećim:

```
public class Student
{
    public int ID { get; set; }

    [Required]
    [StringLength(10, MinimumLength = 3, ErrorMessage
="Podatak o indeksu neispravne dužine")]
    public string index { get; set; }

    [Required]
    public string ime { get; set; }

    [Required]
    [Range(1, 6, ErrorMessage ="Semestar može biti 1-6")]
    public int semestar { get; set; }

    [Required]
    public DateTime datumUpisa { get; set; }
}
```

Anotacija `[Required]` označava da se ta vrednost mora uneti. Slično, za svojstvo index definiše se anotacija `[StringLength]` koja definiše minimalnu odnosno maksimalnu dužinu indeksa studenta, kao i poruku koja će se koristi za prikaz u slučaju greške.

Anotacije ujedno prestavljaju i dodatna ograničenja koja se mogu preneti na opis u bazu podataka, a to se vrši migracijama.

Korak 4: Otvoriti konzolu Package Manager i unesite sledeću komandu:

```
add-migration DataAnnotations
```

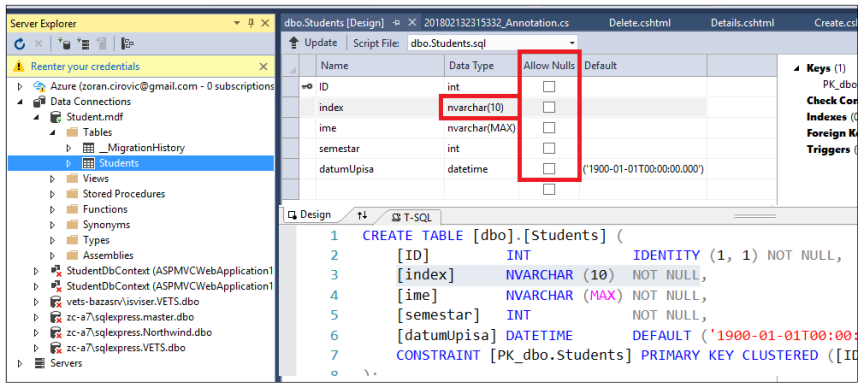
Visual Studio je ažurirao fajl `DataAnnotations.cs` čija je bazna klasa `DbMigration`. Novi kod je:

```
namespace ASPMVCWebApplication1.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class Annotation : DbMigration
    {
        public override void Up()
        {
            AlterColumn("dbo.Students", "index", c =>
c.String(nullable: false, maxLength: 10));
            AlterColumn("dbo.Students", "ime", c =>
c.String(nullable: false));
        }

        public override void Down()
        {
            AlterColumn("dbo.Students", "ime", c =>
c.String());
            AlterColumn("dbo.Students", "index", c =>
c.String());
        }
    }
}
```

Ova ograničenja još uvek nisu preneta na bazu. Dakle, definisane anotacije se prenose na bazu pokretanjem iste komande koju smo i ranije koristili:

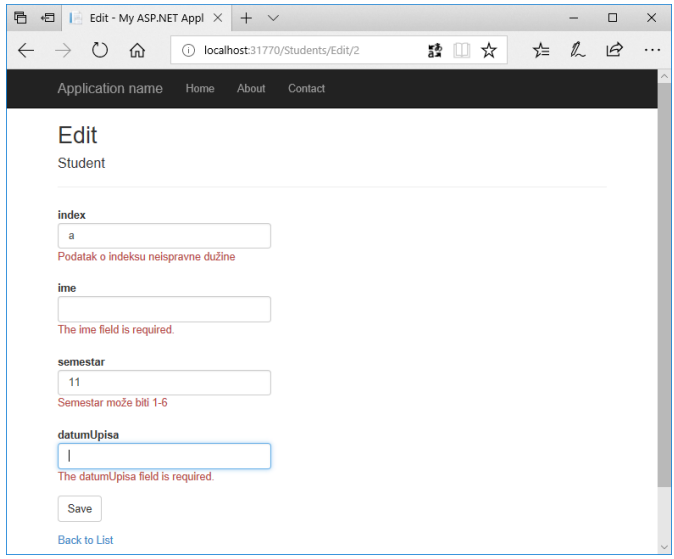
update-database

Pogledajmo sada definiciju polja u tabeli Student.



Slika 13.19. Ograničena u tabeli kreirana na osnovu primenjenih anotacija

Korak 5: Testirajte vašu aplikaciju i otvorite stranicu za Osobe i testirajte Edit.



Slika 13.20. Pogled na stranicu sa primenjenim ograničenjima u slučaju neispravnih podataka

Eksplicitna validacija

Validacija može biti i eksplicitno pozivana koristeći svojstvo `IsValid` u okviru objekta `ModelState`. Pogledajmo primer eksplicitne validacije u kontroleru Student, u metodi Edit pre snimanja izmenjenih podataka.

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include =
"ID,index,ime,semestar,datumUpisa")] Student student)
{
    if ( ModelState.IsValid )
    {
        db.Entry(student).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(student);
}
```

Validacija u pogledu

Validacija mora biti praćena porukama korisniku. Zato je važno da se validacija ugradi u poglede gde se i primenjuje. Pogledajmo najpre kod Edit.cshtml pogleda, a zatim ćemo objasniti kako funkcioniše:

```
@model ASPMVCWebApplication1.Models.Student

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>


@using (Html.BeginForm())
```

```
{
    @Html.AntiForgeryToken()

    <div class="form-horizontal">
        <h4>Student</h4>
        <hr />
        @Html.ValidationSummary(true, "", new { @class =
"text-danger" })
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.index,
htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.index, new
{ htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model =>
model.index, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.ime,
htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.ime, new {
htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model =>
model.ime, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.semestar,
htmlAttributes: new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.semestar,
new { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model =>
model.semestar, "", new { @class = "text-danger" })
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.datumUpisa,
htmlAttributes: new { @class = "control-label col-md-2" })
```

```
            <div class="col-md-10">
                @Html.EditorFor(model => model.datumUpisa,
new { htmlAttributes = new { @class = "form-control" } })
                @Html.ValidationMessageFor(model =>
model.datumUpisa, "", new { @class = "text-danger" })
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save"
class="btn btn-default" />
            </div>
        </div>
    </div>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

- `@Html.ValidationSummary` – Koristi se za prikaz svih poruka koje potiču od grešaka od svih polja. Takođe se koristi da bi se prikazala neka specifična poruka o grešci. Vraća jednu neuređenu listu (`ul` element) poruka koje su u objektu `ModelStateDictionary`.
- `@Html.ValidationMessageFor` – Vraća HTML markap kod za poruku greške validacije za svako polje podataka koje je definisano izrazom.

Pitanja i zadaci za proveru znanja

1. Objasnite pojam migracije.
2. Koji prozor u IDE okruženju koristite za pokretanje migracija?

3. Modifikujte metodu Seed. Gde se ova metoda nalazi i čemu služi?
4. Kako se kreira migracija? Za tekući primer pokrenite početnu migraciju.
5. Šta je rezultat migracije? Objasnite naziv i funkciju novog cs fajla koji je dodat u vaš projekat.
6. Sada promenite model, ažurirajte sve stranice, a zatim uradite novu migraciju.
7. Uradite sopstvenu pretragu podataka po jednom ili više polja.
8. Uradite validaciju podataka koristeći poznate anotacije. Za ove promene pogledajte migracije, ali i izmenjene tabele u bazi.
9. Šta je eksplicitna validacija?
10. Kako se obavlja validacija u pogledu?

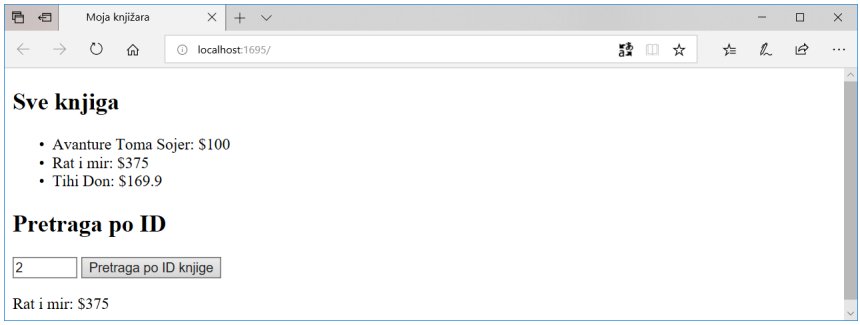
14. Web Api

HTTP je protokol namenjen ne samo za dostavljanje veb sadržaja u vidu stranica. HTTP je moćna platforma i za izgradnju API-ja koji nude usluge i podatke. HTTP je jednostavan, fleksibilan i sveprisutni. Gotovo svaka platforma ima biblioteke za HTTP. HTTP servisi mogu da ostvare širok spektar klijenata, uključujući pregledače, mobilne uređaje kao i tradicionalne desktop aplikacije.

ASP.NET Web API je radni okvir za izgradnju veb API-ja u .NET Framework. U nastavku pokazaćemo način kreiranja ASP.NET Web API koji formira listu knjiga.

Kreiranje Web API projekta

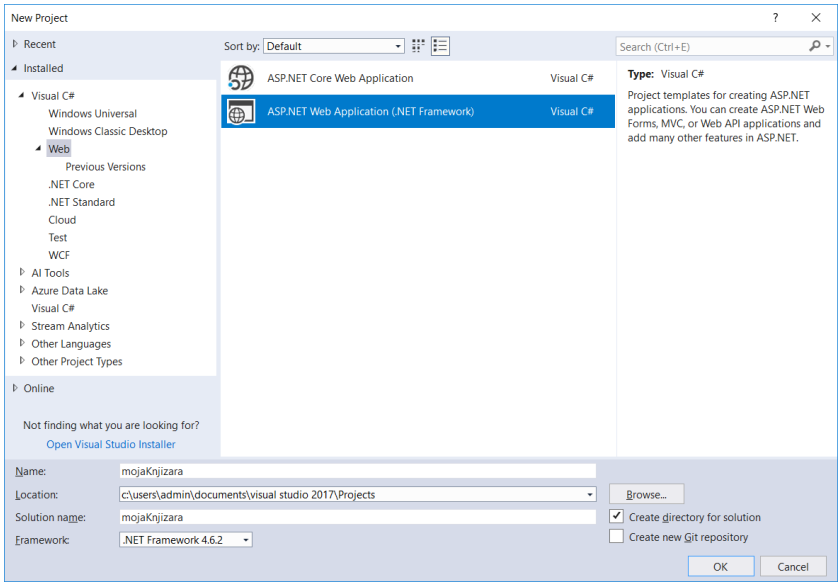
U ovom predavanju, koristićete **`ASP.NET Web API`** za kreiranje veb API koji daje listu knjiga. Veb stranica koristi jQuery pristupanje api funkcijama i vraćanje rezultata.



Slika 14.1. Prikaz stranice koju kreiramo primenom Web API

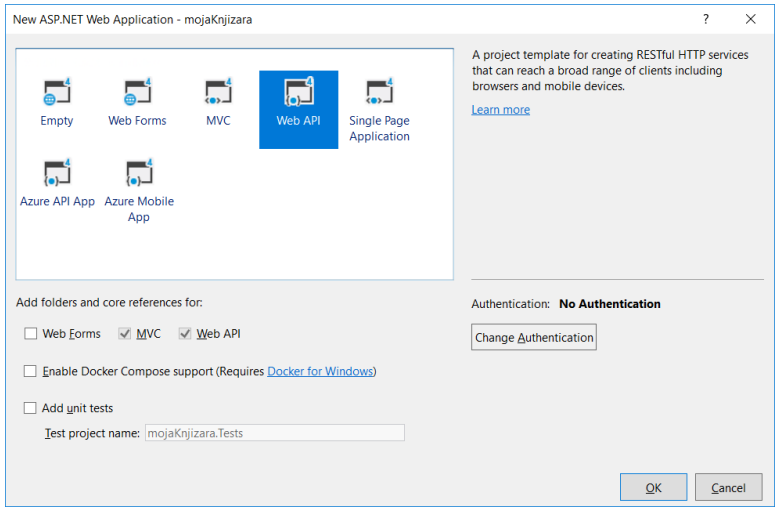
Pokrenimo Visual Studio i odaberimo **New Project**.

U delu **`Templates`** odaberimo **`Installed Templates`** zatim proširimo čvor **Visual C#**. Pod opcijom **Visual C#**, odaberimo **Web**. U listi šablona projekta odaberimo **ASP.NET Web Application**. Imenujmo projekat "mojaKnjizara" a zatim **OK**.



Slika 14.2. Formiranje novog projekta

U dijalogu **New ASP.NET Project** odaberite šablon **Empty**. Odaberite **Web API**. Klik na **OK**.



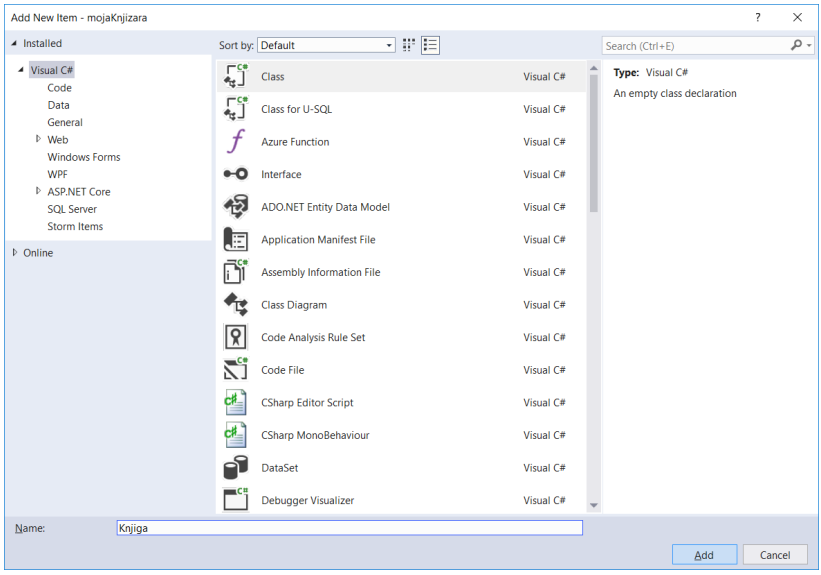
Slika 14.3. Izbor šablona za novi projekta

Dodavanje Modela

Model je objekat koji predstavlja podatke u aplikaciji. ASP.NET Web API može automatski da serijalizuje model u formate kao što je JSON, XML. A zatim se takvi podaci mogu ubaciti u telo poruke koja je HTTP odgovor. Većina klijenata može da analizira ili XML ili JSON. Štaviše, možete naznačiti format koji želite da se prihvati kroz zaglavlje HTTP zahteva za poruke.

Kreirajmo sada jednostavan model.

U prozoru **`Solution Explorer`**, desni-klik na folder Models, a zatim iz kontekstnog menija birati opciju **Add** a zatim odabrati **Class**.



Slika 14.4. Dodavanje nove klase za model

Imenujmo klasu "Knjiga". Zatim dodajmo sledeća svojstva ovoj klasi.

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Web;

namespace mojaKnjizara.Models
{
    public class Knjiga
    {
        public int Id { get; set; }
        public string Naziv { get; set; }
        public string Kategorija { get; set; }
        public decimal Cena { get; set; }
    }

}
```

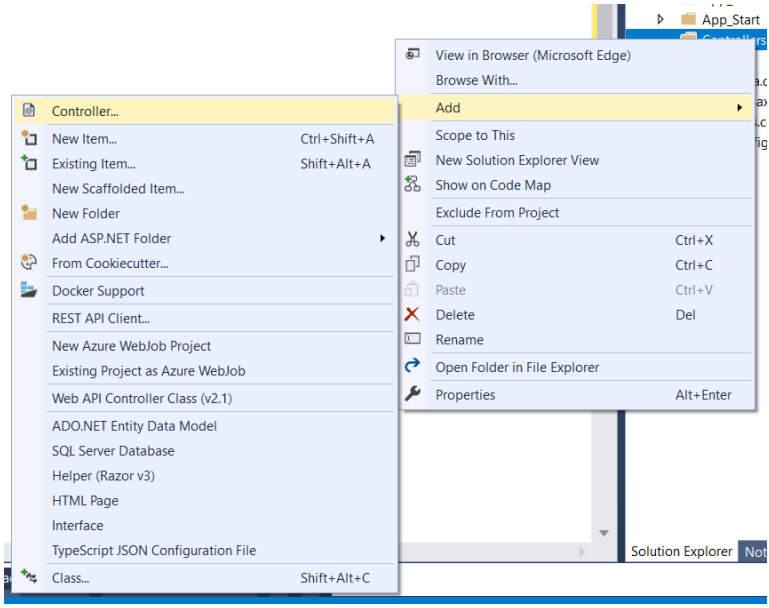
Dodavanje kontrolera

U Web API, kontroler je objekat koji rukuje HTTP zahtevom. Dodaćemo kontroler koji može da vrati listu proizvoda ili jedan proizvod a koji je zahtevan na osnovu prosleđenog Id podatka.

Napomena

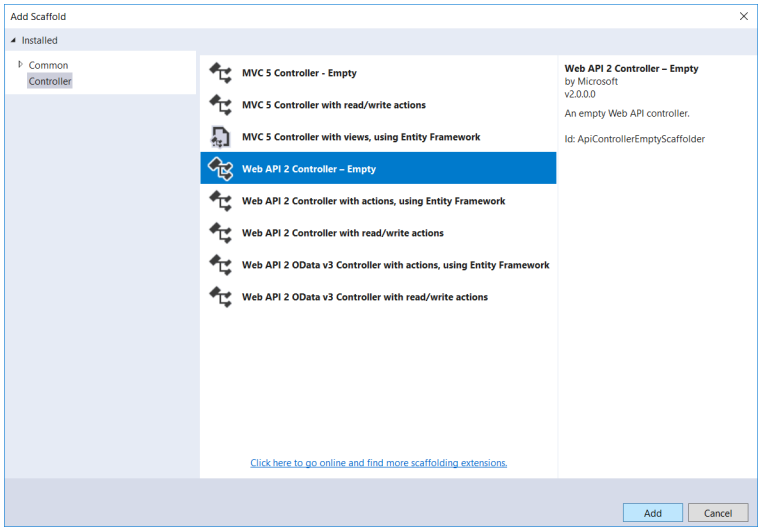
Ako ste već koristili ASP.NET MVC, onda ste već bliski i koristili ste kontrolere. Web API kontroleri su slični MVC kontrolerima, s tim što nasleđuju klasu `ApiController` umesto klase `Controller`.

Korak 1. U prozoru `Solution Explorer`, desni-klik na Controllers folder, zatim odaberite Add a zatim odaberite Controller.



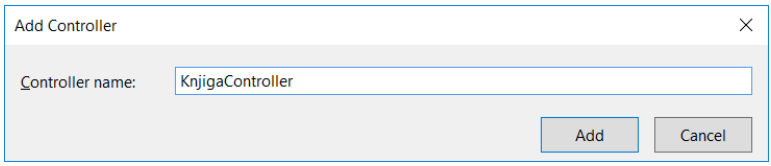
Slika 15.5. Dodavanje kontrolera

Korak 2. Odaberite **Web API Controller - Empty**. a zatim **Add**.



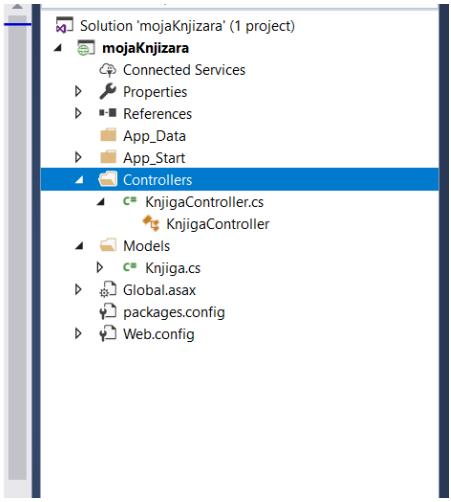
Slika 14.6. Izbor kontrolera

Korak 3. U dijalogu **`Add Controller`** imenujte kontroler **`KnjigaController`**. Odaberite **`Add`**.



Slika 14.7. Definisanje naziva kontrolera

Mehanizam **scaffolding** kreira jedan fajl naziva **KnjigaController.cs** u folderu **Controllers**.



Slika 14.8. Pogled na prozor **Solution Explorer**

Napomena: Nije neophodno da se kontroler postavi u folder koji se naziva Controllers. Izbor naziva foldera je pitanje konvencije i organizacije fajlova u projektu

Ako ovaj fajl nije već automatski otvoren, dvostrukim-klikom na fajl on će se otvoriti za editovanje. Uredite kod na način kako je to prikazano u nastavku:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using mojaKnjizara.Models;
```

```
namespace mojaKnjizara.Controllers
{
  public class KnjigaController : ApiController
  {
    // Kreiranje podataka
    // Kasnije ovi podaci se preuzimaju iz neke baze
    Knjiga[] knjige = new Knjiga[]
    {
      new Knjiga { Id = 1, Naziv = "Avanture Toma Sojer",
Kategorija = "Dečiji", Cena = 100 },
      new Knjiga { Id = 2, Naziv = "Rat i mir", Kategorija
= "Klasika", Cena = 375 },
      new Knjiga { Id = 3, Naziv = "Tihi Don", Kategorija =
"Klasika", Cena = 169.9M }
    };

    [System.Web.Http.HttpGet]
    public IEnumerable<Knjiga> SveKnjige()
    {
      return knjige;
    }

    [System.Web.Http.HttpGet]
    public IHttpActionResult Knjiga(int id)
    {
      var knjiga = knjige.FirstOrDefault((p) => p.Id == id);
      if (knjiga == null)
      {
          return NotFound();
      }
      return Ok(knjiga);
    }

  }
}
```

Kontroler definiše dve metode koje vraćaju knjige:

- Metoda `SveKnjige` vraća celu listu knjiga kao neki `IEnumerable<Knjiga>` tip.
- Metod Knjiga pronalazi jednu knjigu iz liste knjiga koje postoje.

Svaki metod na kontroleru odgovara jednom ili više URL-ova:

U primeru se koristi pojednostavljen primer u kome se knjige formiraju u kontroleru tako što ih kreiramo neposredno u kodu. Umesto toga u realnom slučaju, to bi bilo preko pristupa nekoj bazi podataka ili drugom spoljnjem izvoru podataka odnosno dobavljanjem podataka iz nje.

Tabela 14.1. Primer povezivanja metode i URL-a

Metod kontrolera	URI
SveKnjige	/api/SveKnjige
Knjiga	/api/Knjiga/*id*

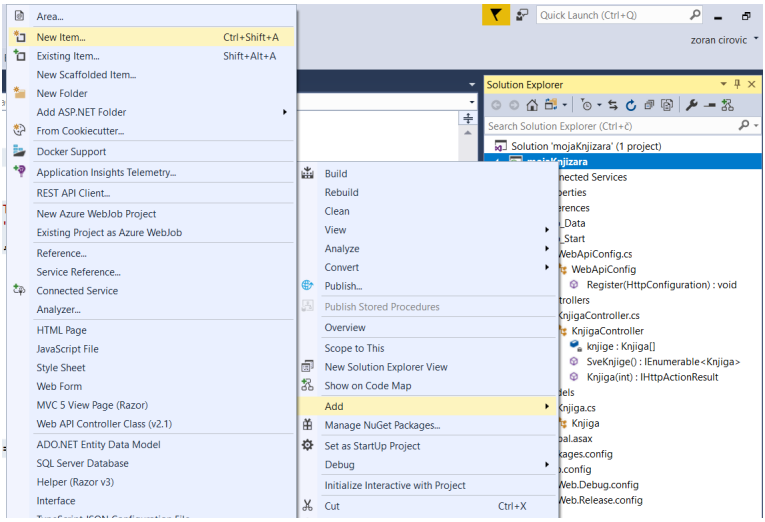
Za metodu Knjiga argument id je u URI. Na primer, da bi se dobila knjiga čiji je `ID = 5`, odgovarajući URI je: `api/knjiga/5`.

Veb API ima u kreiranom projektu podrazumevano definisan način rutiranja URI ka metodama, za više informacija pogledati: `https://docs.microsoft.com/en-us/aspnet/veb-api/overview/veb-api-routing-and-actions/routing-in-aspnet-veb-api`.

JavaScript / jQuery pozivi

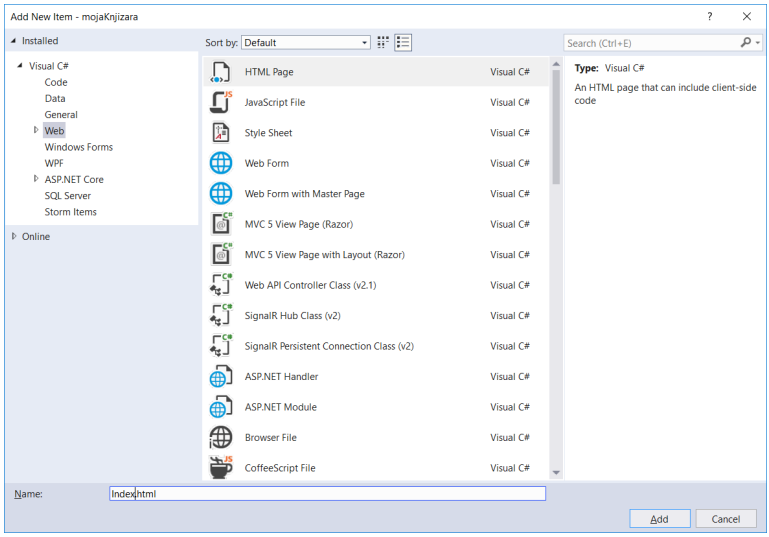
U ovom poglavlju videćemo kako da dodamo neku HTML stranicu koja koristi AJAX da bi pozvala veb API. Koristićemo jQuery kako bi napravili AJAX poziv a zatim ažurirali stranicu sa rezultatom tj knjigama.

Korak 1. U prozoru `Solution Explorer`, desni-klik na projekat a zatim odaberite Add, zatim odaberite New Item.



Slika 14.9. Dodavanje nove stavke

Korak 2. U dijalogu **Add New Item** odaberite čvor **Web** pod opcijom **Visual C#**, a zatim odaberite **HTML Page**. Nazovite stranicu `Index.html`.

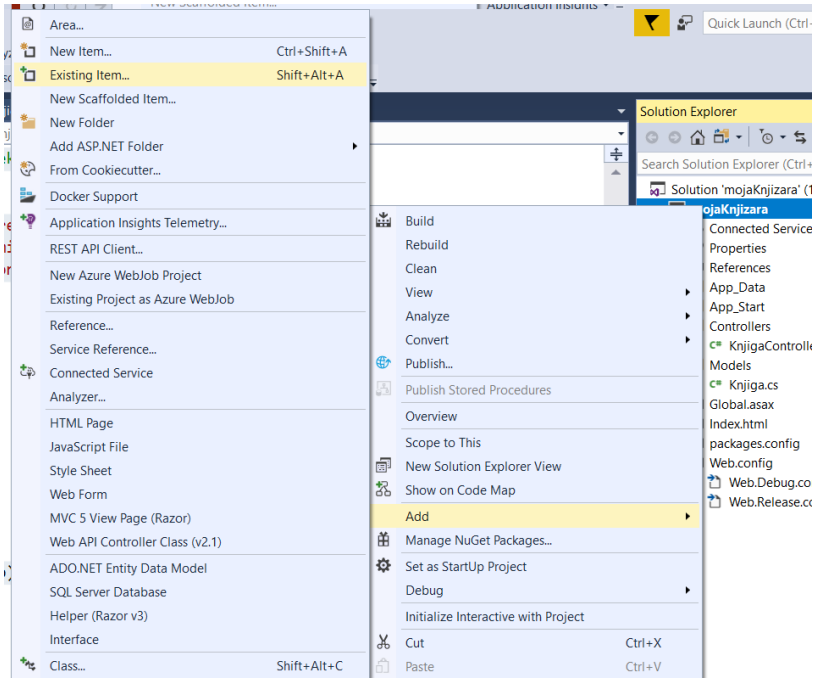


Slika 14.10. Izbor HTML stranice

Zatim dodajte kod:

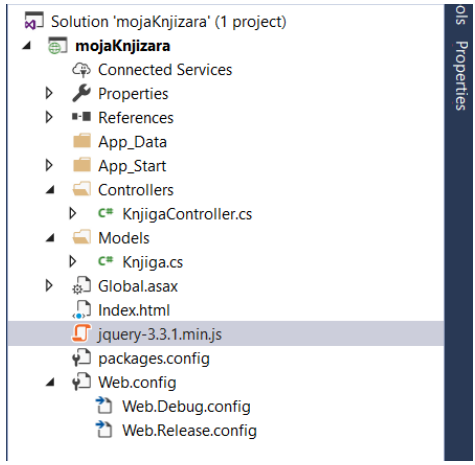
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using mojaKnjizara.Models;

namespace mojaKnjizara.Controllers
{
    public class KnjigaController : ApiController
    {
        // Kreiranje podataka
        // Kasnije ovi podaci se preuzimaju iz neke baze
        Knjiga[] knjige = new Knjiga[]
        {
            new Knjiga { Id = 1, Naziv = "Avanture Toma
Sojer", Kategorija = "Dečiji", Cena = 100 },
            new Knjiga { Id = 2, Naziv = "Rat i mir",
Kategorija = "Klasika", Cena = 375 },
            new Knjiga { Id = 3, Naziv = "Tihi Don",
Kategorija = "Klasika", Cena = 169.9M }
        };

        [System.Web.Http.HttpGet]
        public IEnumerable<Knjiga> SveKnjige()
        {
            return knjige;
        }

        [System.Web.Http.HttpGet]
        public IHttpActionResult Knjiga(int id)
        {
            var knjiga = knjige.FirstOrDefault((p) => p.Id
== id);
            if (knjiga == null)
            {
                return NotFound();
            }
            return Ok(knjiga);
        }

    }
}
```

Postoji više načina kako da se uključi biblioteka jQuery u veb stranicu. U ovom primeru korišćen je Microsoft Ajax CDN. Moguće je preuzeti celu biblioteku sa lokacije http://jquery.com/, a zatim je uključiti u projekat, kao na slici.



Slika 14.11. Uključivanje postojeće biblioteke projektu

Nakon dodavanja, u Project folderu pojavljuje se ubačeni `js` document, kao na slici:



Slika 14.12. `Solution Explorer` prikaz dodate biblioteke

A zatim isti možete koristiti i u aplikaciji:

```
<!--<script
src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-
3.3.1.min.js"></script>-->

<script src="jquery-3.3.1.min.js"></script>
```

Lista

Da bi se dobila lista knjiga potrebno je da se pošalje HTTP GET zahtev na URI: "/api/knjige".

jQuery funkcija `getJSON` šalje AJAX zahtev. Za odgovor očekuje se niz JSON objekata. Funkcija *done* definiše povratnu funkciju tzv. `callback` koji se poziva kada je zahtev obavljen uspešno. U povratnoj metodi menjamo DOM elemente uključujući povratne informacije:

```
$(document).ready(function () {
    $.getJSON(uri)
        .done(function (data) {
            $.each(data, function (key, item) {
                $('<li>',    {    text:    formatItem(item)
}).appendTo($('#sveknjigeUL'));
```

```
            });
        });
});
```

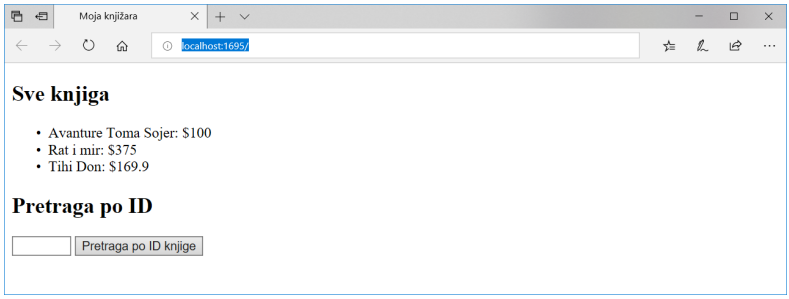
Jedan podatak

Da bi se dobila knjiga po ID, potrebno je da korisnik učita ID I klikom na dugme pošalje zahtev tipa HTTP GET na adresu "/api/knjige/*id*", gde je *id* ID od knjige koja se traži:

```
function find() {
    var id = $('#knjigaId').val();
    $.getJSON(uri + '/' + id)
        .done(function (data) {
            $('#knjigaP').text(formatItem(data));
        })
        .fail(function (jqXHR, textStatus, err) {
            $('#knjigaP').text('Error: ' + err);
        });

}
```

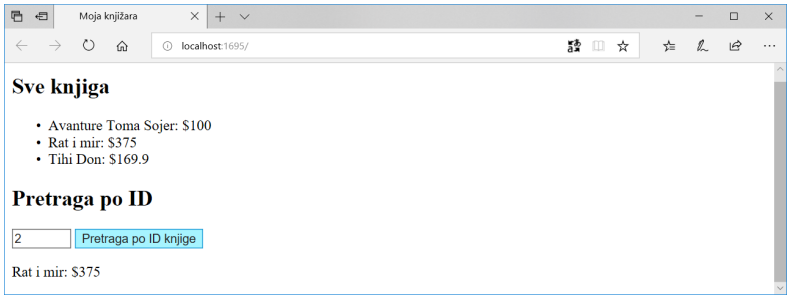
Testiranje aplikacije

Klikom na skraćenicu F5 pokreće se aplikacija sa mogućnošću debagovanja, **Debug mode**. Stranica izgleda na sledeći način:



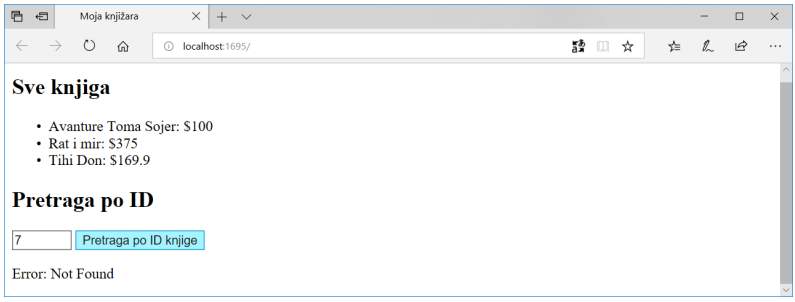
Slika 14.13. Prikaz početne stranice

Da bi se dobila knjiga po ID, unese se vrednost u polje klikne na dugme za pretragu:



Slika 14.14. Prikaz rezultata pretrage

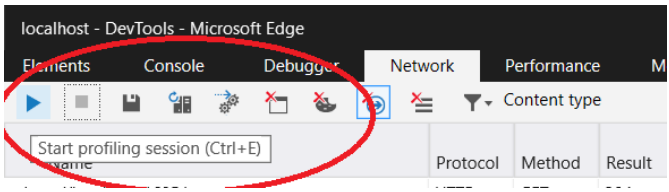
Ako se unese neispravan ID, server vraća HTTP grešku:



Slika 14.15. Prikaz pogrešnog zahteva za pretragu

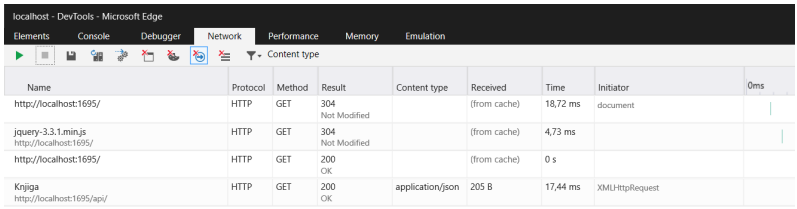
HTTP Request / Response

Kada radite sa HTTP servisima može biti veoma korisno da se vide HTTP zahtevi odnosno odgovori. To se može izvesti pomoću alatke za programere koja se pokreće tasterom **F12**. Odaberite karticu **`Network`**.



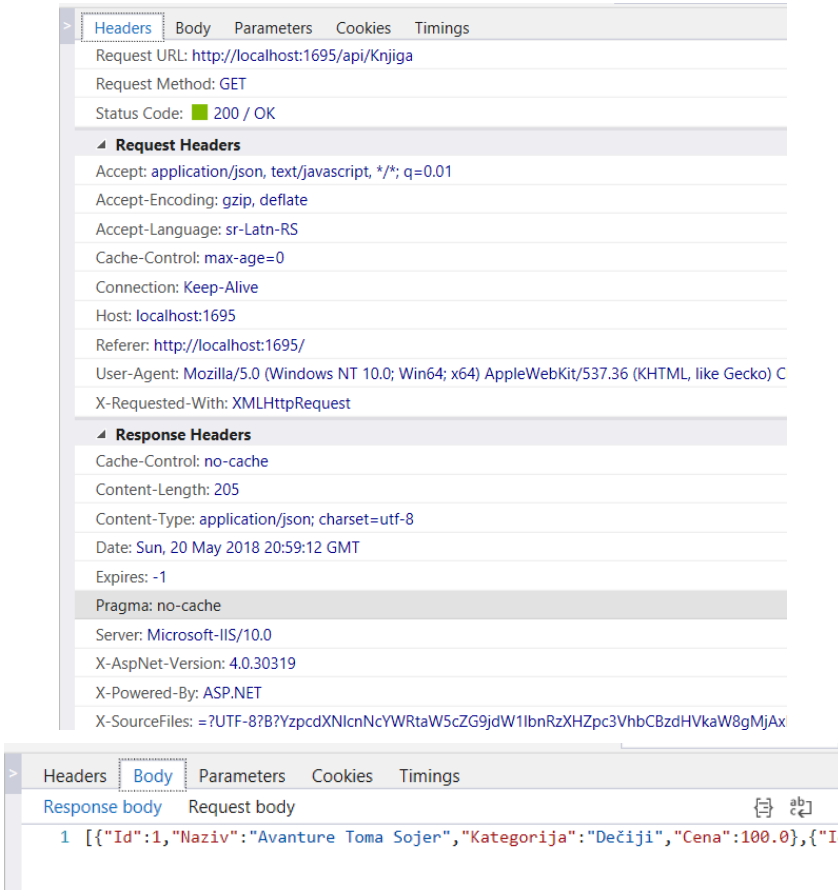
Slika 14.16. Kartice u veb čitaču korišćene za razvoj

zatim pritisnite dugme za „hvatanje“ poruka. Sada se vratite se na veb stranicu i pritisnite taster F5 da ponovo učitati veb stranicu. Veb čitač će izvršiti hvatanje HTTP saobraćaja između veb čitača i veb servera. Slika pokazuje mrežni saobraćaj za stranicu:



Slika 14.17. Network kartica

Nađite poziv za URI **`api/Knjiga/`**. Odaberite tu stavku, a zatim pogledajte detalje u desnom delu prozora. Među detaljima postoje tabovi koji prikazuju **`request`**/**`response`** zaglavlja i telo poruke.



Slika 14.18. Praćenje sadržaja poruka

Pitanja i zadaci za proveru znanja

1. Objasnite pojam Web Api aplikacija. Da li poznajete neki api?
2. Kreirajte novi projekat tipa Web Api. Da li možete birati MVC opciju pri kreiranju projekta?
3. Ubacite vaš model Radnik u ovaj projekat. Obratite pažnju na nazive kolona. Zašto?

4. Kreirati kontroler za podatke o radnicima.
5. Iz koje klase je izvedena klasa kontrolera?
6. Kreirajte metode koje vraćaju sve radnike odnosno jednog radnika.
7. Definišite URL za pozive kreiranih metoda.
8. Dodajte projektu HTML stranicu a zatim primenom JavaScript-a i jQuery jezika napišite forme za dobavljanje podataka preko vaših metoda.
9. Napraviti novi projekat koji je tipa ASP MVC aplikacije pa tom projektu dodajte kontroler i HTML stranicu kao u prethodnom slučaju.

Literatura

[1] J., Allwork. C# Programiranje za Windows i Android, InfoElektronika 2016.

[2] M., Price. C# 7.1 i .NET Core 2.0 moderno međuplatformsko program., Kompjuter biblioteka 2018.

[3] J. Albahari, B. Albahari, „C# 5.0 za programere sveobuhvatan ref. priručnik“, Mikro knjiga 2015.

[4] B. Watson, „C# 4.0: Kako do rešenja. Rešeni zadaci iz prog. na jeziku C#“, Mikro knjiga. 2011.

[5] Zoran, Ćirović. *Tehnike vizuelnog programiranja*. VETS 2005.

[6] Jon Galloway, Brad Wilson, K. Scott Allen and David Matson. Professional Asp.Net MVC 5. Wrox 2014.

[7] Freeman, Adam. *Pro ASP.NET MVC 5*. Apress 2013.

[8] Nimit, Joshi. James, Henry. Programming *ASP.NET MVC 5 A Problem Solution Approach. The Ambassadors*. C# Corner 2013.

[9] Jess Chadwick, Todd Snyder, and Hrusikesh Panda. *Programming ASP.NET MVC 4*. O'Reilly Media 2012.

[10] Morem, Susan. 101 Tips for Graduates. http://www.infobasepublishing.com

[11] Stephen Walther, ASP.NET MVC Views Overview (C#), https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/views/asp-net-mvc-views-overview-cs

[12] https://getbootstrap.com

[13] Tom FitzMacken, Introduction to ASP.NET Web Programming Using the Razor Syntax

[14] Main Razor Syntax Rules for C#, www.w3schools.com/asp/razor_syntax.asp

[15] Steve Smith, Build beautiful, responsive sites with Bootstrap and ASP.NET Core

[16] Freeman, Adam. *Pro ASP.NET MVC 5*. Apress 2013.

[17] Tom FitzMacken, EF Database First with ASP.NET MVC: Creating the Web Application and Data Models, 01.10.2014

[18] Rick Anderson, Using the DropDownList Helper with ASP.NET MVC

[19] Rick Anderson, Examining how ASP.NET MVC scaffolds the DropDownList Helper

[20] Joseph Albahari, Ben Albahari, C# 7.0 in a Nutshell, 2017.

[21] Bill Wagner, Maira Wenzel, Mike B, Luke Latham, Getting Started with LINQ in C#

[22] LINQ Tutorial for Beginners, https://www.codeproject.com/Tips/590978/LINQ-Tutorial-for-Beginners

[23] 101 LINQ Samples, https://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b

[24] Tom Dykstra, Rick Anderson, Getting Started with Entity Framework 6 Code First using MVC5, Apress 016

[25] Lee Naylor, ASP.NET MVC with Entity Framework and CSS, Apress 2016.

[26] Mike Wasson, Use Code First Migrations to Seed the Database, Microsoft doc.

[27] Mike Wasson, Get Started with ASP.NET Web API 2 (C#), Microsof doc.

[28] https://www.asp.net/veb-api

[29] https://developer.spotify.com/documentation/veb-api/quick-start/

Indeks pojmova

P

R

S

T

U

V

X

Indeks slika