

## Laboratorijska Vježba 6. ASP.NET Core MVC Kontroleri i Rutiranje

### Cilj vježbe:

- Upoznavanje sa načinom automatskog kreiranja kontrolera i pogleda, kao i pripadajućih metoda za obradu podataka;
- Definisanje rutiranja u kontroleru bez perzistencije podataka.

### 1. Kreiranje modela bez korištenja baze podataka

U prethodnoj laboratorijskoj vježbi kreirana je jednostavna ASP.NET Core MVC aplikacija koja se može uspješno pokrenuti. Ova aplikacija posjeduje *Home* kontroler i dvije forme *Index* i *Privacy*, što nije dovoljno za demonstriranje rada sa kontrolerima, njihovim metodama i rutiranjem. Iz tog razloga prvo je potrebno kreirati neki model podataka na osnovu kojeg će se generisati kontroleri i pogledi sa predefinisanim programskim kodom koji će se zatim nadograđivati. U ovoj laboratorijskoj vježbi neće se koristiti baza podataka, već će se koristiti model klasa bez perzistencije (svaki put kada se rad aplikacije zaustavi, svi podaci koji su uneseni u toku njenog rada bivaju obrisani).

Neka je potrebno omogućiti da aplikacija vrši upis studenata na predmete, kao simulacija studentske službe fakulteta. Svaki student ima svoje ime, prezime i broj indeksa, kao i pripadajuću godinu studija i predmete na koje je upisan. Svaki predmet ima svoj ID broj, naziv, broj ECTS bodova i studente koji su na njega upisani. Za upis na predmet važno je zapamtiti na koju akademsku godinu se upis odnosi, koji je datum upisa i koji je broj protokola. Za ove zahtjeve moguće je napraviti jednostavni sistem koji se sastoji od tri klase: *Student*, *Predmet* i *UpisNaPredmet*, koje je potrebno definisati **isključivo u Models folderu**.

U nastavku je prikazan programski kod klase *Predmet*. Postoji nekoliko ključnih razlika u odnosu na način definisanja klasa u prethodnim laboratorijskim vježbama:

- **Atributi klase su implicitno definisani**, na način da je definisana samo *property* metoda koja omogućava dobavljanje i izmjenu tih atributa. Sve *property* metode označene su javnim.
- Iznad svih *property* metoda nalaze se **stereotipi**. Ovi stereotipi koriste se za dodavanje oznaka atributima koji su ključni za ispravan rad sa modelom podataka. Stereotip *Key* označava da je atribut primarni ključ tabele podataka, odnosno da se pretraživanje prema identitetu vrši na osnovu tog atributa. Za automatsko generisanje koda za neku klasu (koje će biti demonstrirano u nastavku) ta klasa mora imati definisan primarni ključ, inače automatsko generisanje neće biti dozvoljeno. Stereotip *Required* označava da nije moguće izvršiti dodavanje novog reda u tabelu podataka ukoliko željeni atribut nema dodijeljenu vrijednost. Stereotip *NotMapped* koristi se za oznaku da klasa sadrži atribut koji se ne može preslikati u jednu ćeliju tabele baze podataka (npr. lista koja sadrži više elemenata).

Na ovaj način definišu se sve tri klase modela podataka koje se žele koristiti u okviru aplikacije i za koje će biti generisani pripadajući kontroleri i pogledi.

```
public class Predmet
{
    #region Properties

    [Key]
    [Required]
    public int ID { get; set; }

    [Required]
    public string Naziv { get; set; }

    [Required]
    public double ECTS { get; set; }

    [NotMapped]
    public List<Student> UpisaniStudenti { get; set; }

    #endregion

    #region Konstruktor

    public Predmet() { }

    public Predmet(string name, double points)
    {
        ID = GenerišiID();
        Naziv = name;
        ECTS = points;
        UpisaniStudenti = new List<Student>();
    }

    #endregion

    #region Metode

    public int GenerišiID()
    {
        int id = 0;
        Random generator = new Random();
        for (int i = 0; i < 10; i++)
        {
            id += (int)Math.Pow(10, i) * generator.Next(0, 9);
        }
        return id;
    }

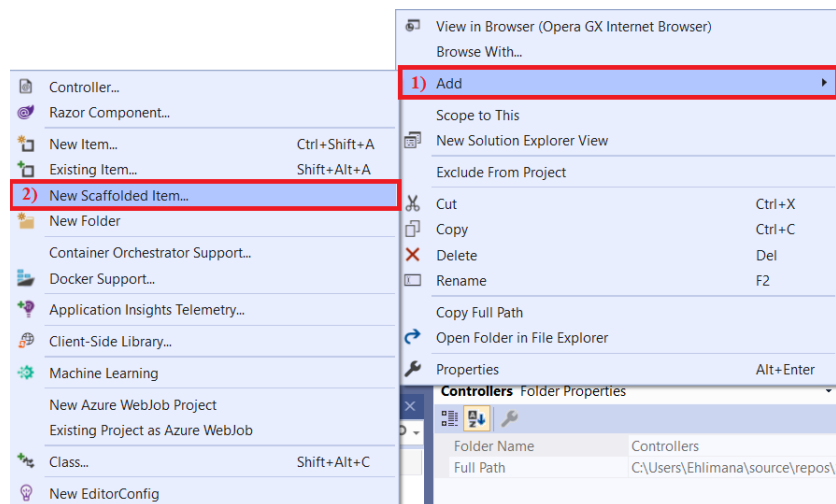
    #endregion
}
```

## 2. Scaffolding kontrolera i pogleda

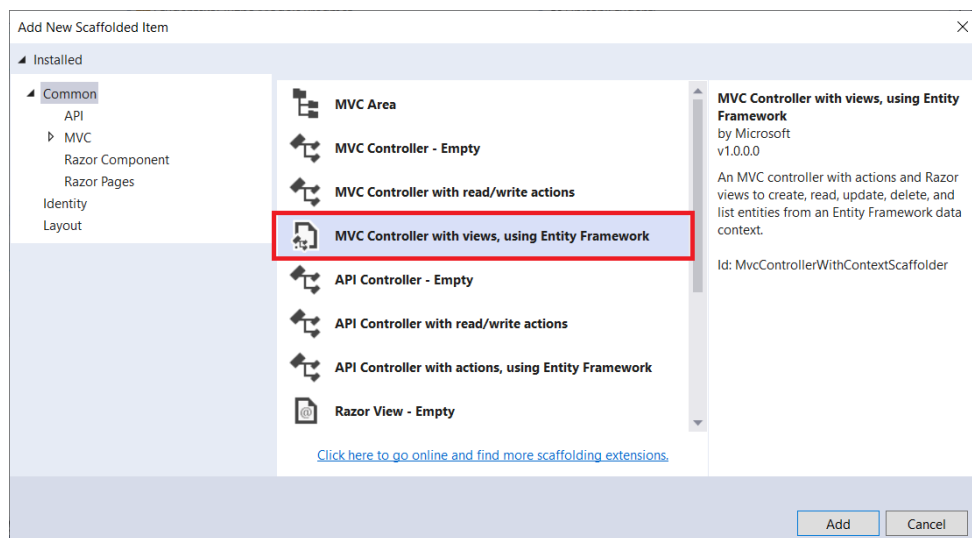
Iako je moguće manuelno dodavati kontrolere i povezivati ih sa željenim klasama modela podataka i pogleda, to nije preporučeno jer je potrebno manuelno registrovati nove kontrolere i njihove rute. Osim toga, u slučaju kada model podataka ima mnogo atributa sa kompleksnom logikom za validaciju podataka, pravljenje pogleda za dodavanje i izmjenu tog modela može

biti mukotrpan posao. Iz tog razloga koristi se **scaffolding**, odnosno automatsko generisanje komponenti kontrolera i pogleda na osnovu željene klase modela podataka.

Da bi se izvršio **scaffolding**, potrebno je izvršiti desni klik na folder **Controllers**, a zatim odabrati opciju **Add → New Scaffolded Item...**, na način prikazan na Slici 1. Nakon toga dobiva se opcija odabira kakav automatski generisani kod se želi kreirati. Postoji mnogo opcija, od praznog kontrolera bez akcija, kontrolera bez pogleda (**API Controller**) ili pogleda bez kontrolera (**Razor View**). Potrebno je odabrati kontroler sa pogledima koji je definisan kao **MVC Controller With Views, using Entity Framework** (prikazano na Slici 2).



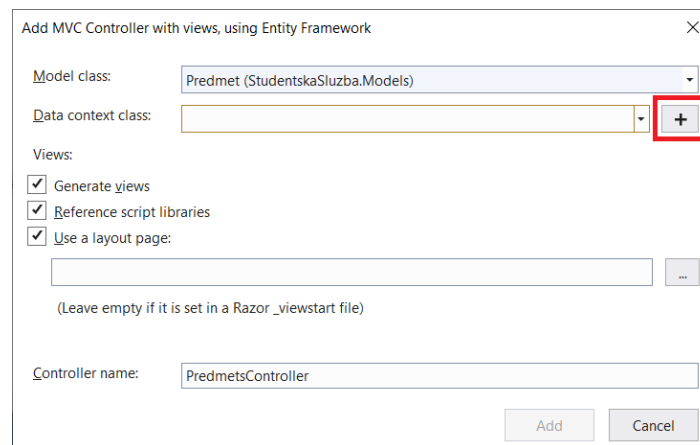
Slika 1. Odabir opcije za scaffolding kontrolera i pogleda



Slika 2. Odabir ispravne vrste objekta koji se želi scaffold-ati

Sada je potrebno specificirati izvor na osnovu kojeg se želi vršiti automatsko generisanje koda. Potrebno je odabrati željenu **model klasu** (u ovom primjeru biti će izvršen **scaffolding** za klasu *Predmet*, a za potpunu funkcionalnost programskog rješenja potrebno je izvršiti **scaffolding** za sve model klase) i dati **željeno ime kontroleru**. Svaki kontroler mora imati **Controller** u svom imenu, a prvi dio je moguće modificirati po želji. Odabrano ime koristiti će se za rutiranje, odnosno unosom željenog imena kao rute u web-adresi aplikacije pristupiti će se formama za dati kontroler (npr. pristup formama kontrolera *PredmetsController* vrši se na ruti **localhost:44325/Predmets**).

Još je neophodno definisati **kontekst podataka** (*data context*) na osnovu kojeg će se vršiti automatsko generisanje koda. Kako se u aplikaciji ne koristi nikakva baza podataka, ovaj kontekst nije prethodno definisan i može se automatski generisati klikom na oznaku +, na način prikazan na Slici 3. Nakon toga omogućava se davanje imena kontekstu podataka (moguće je odabrati proizvoljno ime), čime se dodaje novi kontekst podataka i može se započeti *scaffolding* kontrolera i pogleda. Da bi *scaffolding* bio uspješan, klasa za koju se *scaffolding* vrši mora imati definisan primarni ključ (Key) atribut i prazan konstruktor!



Slika 3. Dodavanje novog konteksta podataka za scaffolding

Nakon specifikacije svih podataka, počinje *scaffolding* kontrolera i pogleda. Ovaj proces traje nekoliko minuta i u okviru njega definiše se specificirani kontroler, sve njegove metode rutiranja (*Index*, *Create*, *Update*, *Delete* i *FindById*), kao i pogledi koji omogućavaju pristup svim metodama rutiranja kontrolera. Na ovaj način dobiva se potpuno funkcionalan model sa formama i metodama kontrolera pomoću kojih se može vršiti manipulacija podacima. Ovako definisan programski kod predstavlja dobru osnovu za rad sa podacima koristeći MVC arhitekturni patern.

### 3. Rutiranje u okviru automatski generisanog kontrolera

Najčešće korištene metode kod HTTP zahtjeva su GET i POST metode. U skladu s tim, za svaku funkcionalnost koja se odnosi na CRUD (*Create-Read-Update-Delete*) generisani su pogledi i pripadajuće akcije u kontroleru. Automatski generisani kontroler posjeduje potpuno funkcionalne metode koje omogućavaju sljedeće funkcionalnosti:

- **Index**, metoda koja prikazuje listu sa svim objektima koji su dosad napravljeni, tj. koji se nalaze u bazi podataka. Ova – i svaka druga - metoda je GET, osim ako se ne naglasi drugačije anotacijom.

```
// GET: Predmets
public async Task<IActionResult> Index()
{
    return View(await _context.Predmet.ToListAsync());
}
```

- **Details**, metoda koja dobavlja objekat iz baze podataka na osnovu primarnog ključa i prikazuje sve informacije o njemu.

```
// GET: Predmets/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var predmet = await _context.Predmet
        .FirstOrDefaultAsync(m => m.ID == id);
    if (predmet == null)
    {
        return NotFound();
    }

    return View(predmet);
}
```

- **Create**, set od dvije metode od kojih jedna prikazuje pripadajući pogled za dodavanje objekta, a druga kao parametar prima objekat i zatim vrši njegovo dodavanje u bazu podataka. Druga metoda se poziva kad se izvrši *submit* forme, odnosno kad se klikne na *Create* dugme na formi za unos koja se nalazi na *Create* pogledu. Ona posjeduje **[HttpPost]** anotaciju, čime se metoda proglašava POST metodom. Ova (i sve ostale metode sa ključnom riječi *async*) metoda je asinhrona, što omogućava brže izvršavanje, jer se na taj način oslobađa prostor za rad aplikacije dok se čekaju povratni podaci.

Kako je prikazano, metoda kao parametar prima model, odnosno objekat klase *Predmet*. Ovaj objekat prime se uz stereotip **[Bind("ID,Naziv,ETCS")]**, što služi da se spriječi slanje nepotrebnih podataka u obliku više parametara i da se izvrši sakrivanje pojedinačnih atributa. Svaka metoda prvo vrši provjeru ispravnosti modela **ModelState.IsValid**, kako bi se spriječila manipulacija objektima ukoliko model nije dostupan. Nakon što se izvrši spašavanje podataka, vrši se poziv metode **Index**, koja vrši prikaz liste unesenih predmeta. Ukoliko spašavanje podataka nije moguće izvršiti, vraća se **View**, tj. korisnik se vraća na isti pogled koji je imao i prije klika na dugme *Create*.

```
// GET: Predmets/Create
public IActionResult Create()
{
    return View();
}

// POST: Predmets/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create([Bind("ID,Naziv,ECTS")] Predmet
predmet)
{
    if (ModelState.IsValid)
    {
        _context.Add(predmet);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(predmet);
}
```

- **Edit**, set od dvije metode od kojih jedna prikazuje pripadajući pogled za izmjenu postojećeg objekta, a druga kao parametar prima ID postojećeg objekta i nove vrijednosti njegovih atributa, na osnovu čega se vrši promjena u bazi podataka. U ovoj metodi koriste se isti principi kao i u prethodno detaljno objašnjennoj *Create* metodi.

```
// GET: Predmets/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var predmet = await _context.Predmet.FindAsync(id);
    if (predmet == null)
    {
        return NotFound();
    }
    return View(predmet);
}

// POST: Predmets/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id, [Bind("ID,Naziv,ECTS")]
Predmet predmet)
{
    if (id != predmet.ID)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(predmet);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!PredmetExists(predmet.ID))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(predmet);
}
```

- **Delete**, set od dvije metode od kojih jedna prikazuje pripadajući pogled za brisanje postojećeg objekta, a druga kao parametar prima ID postojećeg objekta, na osnovu čega ga briše iz baze podataka. I za ovu metodu koristi se ista programska logika kao i u prethodno prikazanim metodama.

```
// GET: Predmets/Delete/5
public async Task<IActionResult> Delete(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var predmet = await _context.Predmet
        .FirstOrDefaultAsync(m => m.ID == id);
    if (predmet == null)
    {
        return NotFound();
    }

    return View(predmet);
}

// POST: Predmets/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var predmet = await _context.Predmet.FindAsync(id);
    _context.Predmet.Remove(predmet);
    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

- **PredmetExists**, metoda koja kao rezultat vraća informaciju o tome da li objekat čiji je ID poslan kao parametar postoji u bazi podataka. Ova metoda označena je kao *private* što znači da nije direktno dostupna korisniku, ali se koristi u okviru drugih metoda kontrolera.

```
private bool PredmetExists(int id)
{
    return _context.Predmet.Any(e => e.ID == id);
}
```

Automatski generisani pogledi *Index*, *Details*, *Create*, *Edit* i *Delete* omogućavaju programsku logiku za vršenje željenih akcija nad modelom podataka. O programskom kodu ovih metoda biti će riječi u okviru sljedeće laboratorijske vježbe, a u ovoj vježbi biti će korištene kako bi se demonstrirao ispravan rad automatski generisanog kontrolera.

Kao što je vidljivo u isječcima koda iznad, za rad sa podacima koristi se atribut **\_context**. Ovaj atribut predstavlja automatski generisani kontekst podataka koji nema nikakvu funkciju jer aplikacija ne posjeduje konekciju na bazu podataka. Kako se u ovoj laboratorijskoj vježbi

neće izvršiti konfiguracija na izvor podataka, ovaj kontekst potrebno je zamijeniti statičkim atributom u okviru kojeg će se definisati izvor podataka – lista objekata.

Inicijalna definicija kontrolerske klase izgleda kao u programskom kodu prikazanom ispod:

```
public class PredmetsController : Controller
{
    private readonly StudentskaSluzbaContext _context;

    public PredmetsController(StudentskaSluzbaContext context)
    {
        _context = context;
    }
}
```

Nakon izmjene tako da se kao izvor podataka ne koristi kontekst podataka, već predefinisana lista podataka (sa nekoliko inicijaliziranih elemenata), programski kod kontrolerske klase izgleda na sljedeći način:

```
public class PredmetsController : Controller
{
    static List<Predmet> predmeti = new List<Predmet>()
    {
        new Predmet(name: "OOAD", points: 5.0),
        new Predmet(name: "AFJ", points: 5.0),
        new Predmet(name: "ORM", points: 5.0),
        new Predmet(name: "RA", points: 5.0)
    };

    public PredmetsController(StudentskaSluzbaContext context)
    {
    }
}
```

Sada je potrebno izvršiti izmjene u svim predefinisanim metodama tako da se ne koristi kontekst podataka, već prethodno kreirana lista predmeta. U predefinisanim metodama koristi se asinhroni pristup dobavljanja podataka, pa je potrebno ukloniti sve oznake korištenja asinhronosti tako da se dobiva sljedeći programski kod:

```
// GET: Predmets
public async Task<IActionResult> Index()
{
    return View(await predmeti);
}
```

Najznačajnija promjena koju je potrebno izvršiti nalazi se u metodi za editovanje postojećih predmeta. U ovoj metodi poziva se funkcija **`_context.Update(predmet);`**, a lista kao kolekcija podataka ne posjeduje metodu *Update*. Ovu metodu potrebno je zamijeniti programskim kodom gdje se manuelno vrši pronalazak postojećeg predmeta, a zatim izjednačavanje svih njegovih atributa sa atributima novog predmeta<sup>1</sup>, čime se dobiva sljedeći programski kod:

---

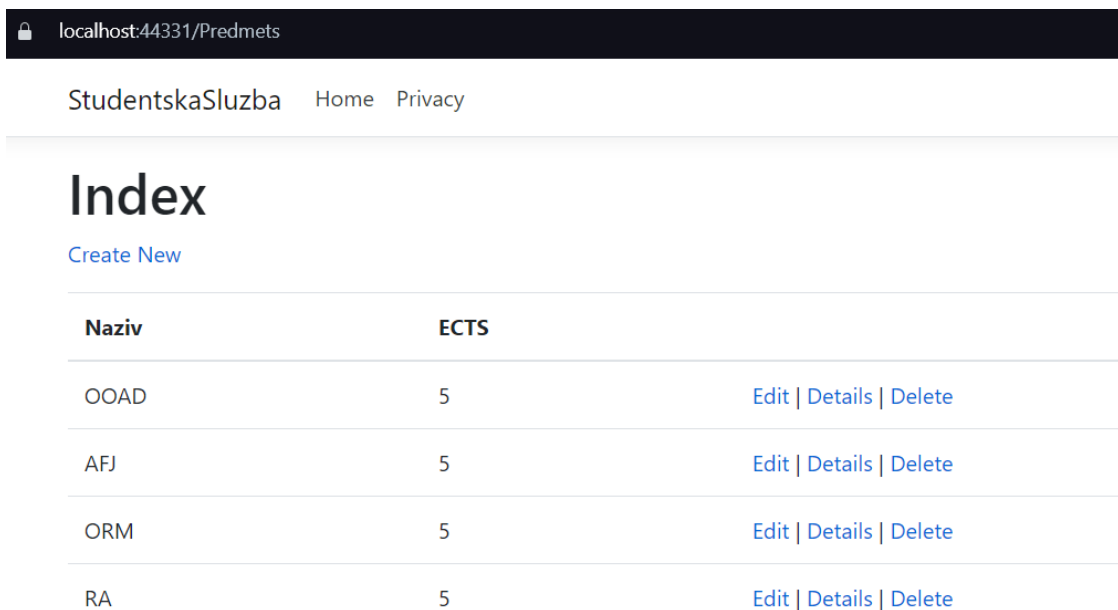
<sup>1</sup> Ovu funkcionalnost moguće je i definisati u okviru klase *Predmet*, tako da se umjesto postavljanja pojedinačnih atributa cijeli objekat šalje kao parametar metodi za izmjenu atributa postojećeg objekta.



```
// POST: Predmets/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, [Bind("ID,Naziv,ECTS")] Predmet predmet)
{
    ...

    if (ModelState.IsValid)
    {
        try
        {
            var stariPredmet = predmeti.Find(p => p.ID == id);
            stariPredmet.ID = predmet.ID;
            stariPredmet.Naziv = predmet.Naziv;
            stariPredmet.ECTS = predmet.ECTS;
            stariPredmet.UpisaniStudenti = predmet.UpisaniStudenti;
        }
        ...
    }
}
```

Na ovaj način dobiva se potpuno definisan kontroler bez perzistencije podataka. Sada je moguće pokrenuti aplikaciju i manuelno se pozicionirati na rutu **<adresa aplikacije>/<naziv kontrolera>/<naziv akcije>** (u konkretnom slučaju web-adresa je: <https://localhost:44331/Predmets/Index>), čime se dobiva pristup *Index* stranici ovog kontrolera. Na Slici 4 prikazan je izgled ovog pogleda, koji učitava inicijalizirane elemente liste koja je kreirana u kontrolerskoj klasi za predmete. Ovaj pogled omogućava pristup formama *Create New*, *Edit*, *Details* i *Delete* koje omogućavaju rad sa postojećim predmetima i dodavanje novih predmeta. Ove poglede sada je moguće mijenjati i prilagođavati korisničkim potrebama, što će biti objašnjeno u okviru naredne laboratorijske vježbe.



Naziv	ECTS	
OOAD	5	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
AFJ	5	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
ORM	5	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
RA	5	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

Slika 4. Izgled Index forme za kontroler predmeta

#### 4. Zadaci za samostalni rad

Programski kod aplikacije koja je kreirana u vježbi dostupan je u repozitoriju na sljedećem linku: <https://github.com/ehlymana/OOADVjezbe>, na *branchu* **lv6**.

1. U laboratorijskoj vježbi izvršen je *scaffolding* kontrolera i pogleda samo za klasu *Predmet*. Potrebno je izvršiti *scaffolding* kontrolera i pogleda za klase *Student* i *UpisNaPredmet*. Voditi računa da je moguće ponovo iskoristiti postojeći kontekst podataka pri specifikaciji parametara za automatsko generisanje koda.
2. Pokrenuti aplikaciju i testirati njene funkcionalnosti. Ukoliko se doda novi predmet, zatim se rad aplikacije zaustavi i ponovo pokrene, da li će novododani predmet biti prisutan u listi predmeta? Šta to znači za rad aplikacije?