**Rukovanje promenama u modelu**

**00:02:55**

Jedinica: 16 od 19

**Rezime**

* Postoje tri osnovne strategije za inicijalizaciju baze podataka, koje su ugrađene u EF.
* CreateDatabaseIfNotExists – baza podataka će biti kreirana samo ukoliko ne postoji.
* DropCreateDatabaseIfModelChanges – postojeća baza se odbacuje i nova se automatski kreira kada dođe do promene na modelu.
* DropCreateDatabaseAlways – postojeća baza podataka se uvek odbacuje, a nova generiše, bez obzira na to da li je došlo do promene na klasama modela.
* Sve strategije za inicijalizaciju baze podataka koriste se kao instance istoimenih klasa, koje se prosleđuju metodi SetInitializer.
* EF Code First API omogućava potpuno samostalno kreiranje strategije inicijalizacije, koja odgovara potrebama aplikacije koja se razvija.
* Proces ubacivanja podataka u bazu  prilikom njene inicijaliazcije naziva se *Seed.*
* Funkcionalnost koja omogućava automatsko ažuriranje konceptulanog modela, ali bez gubitka bilo kakvih podataka ili objekata baze podataka naziva se migracija podataka.
* Postoje dva tipa migracija: automatska migracija i migracija koja se zasnivana na kodu.

U prethodnim lekcijama ovog modula prikazane su osnove korišćenja EF Code First API-a, ali i tehnike kojima se može izvršiti konfigurisanje procesa pretvaranja objektnog u konceptualni model. Tokom razvijanja objektnog modela često će se javiti potreba za modifikacijama na kreiranim klasama. U ovoj lekciji će upravo o tome biti reč, odnosno o tome kako rukovati eventualnim promenama na objektnom modelu.

U prethodnim lekcijama demonstrirano je i kako nakon prvog pokretanja aplikacije Code First API obavlja generisanje tabela baze podataka. Ali, šta se događa prilikom narednih pokretanja aplikacije? Da bi se odgovorilo na ovo pitanje, potrebno je razumeti različite strategije za inicijalizaciju baze podataka.

**Strategije za inicijalizaciju baze podataka**

Postoje tri osnovne strategije za inicijalizaciju baze podataka koje su ugrađene u EF:

* + **CreateDatabaseIfNotExists.**Kao što samo ime sugeriše, prema ovoj strategiji baza podataka će biti kreirana samo ukoliko ne postoji. Ukoliko se izvrši neka promena na modelu i aplikacija ponovo pokrene, okruženje će izbaciti izuzetak. Ovo je strategija koja se podrazumevano koristi.
  + **DropCreateDatabaseIfModelChanges.**Korišćenjem ove strategije, kada dođe do promene na modelu, postojeća baza se odbacuje i nova se automatski kreira.
  + **DropCreateDatabaseAlways.**Strategija čijom primenom se postojeća baza podataka uvek odbacuje, a nova generiše, bez obzira na to da li je došlo do promene na klasama modela. Ova strategija je naročito korisna tokom razvija, ukoliko programer želi čistu bazu podataka svaki put kada pokrene aplikaciju.

Sve navedene strategije za inicijalizaciju baze podataka koriste se kao instance istoimenih klasa, koje se prosleđuju metodi SetInitializer. Sledeći blok koda ilustruje način na koji se aktivira odgovarajuća strategija inicijalizacije:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | protected override void OnModelCreating(DbModelBuilder modelBuilder)          {              Database.SetInitializer(new CreateDatabaseIfNotExists<PersonContext>());  //Database.SetInitializer(new DropCreateDatabaseIfModelChanges<PersonContext>());              //Database.SetInitializer(new DropCreateDatabaseAlways<PersonContext>());              //Database.SetInitializer(new PersonDBInitializer());          } |

U primeru je aktivirana CreateDatabaseIfNotExists strategija inicijalizacije, a zakomentarisane linije prikazuju kako se aktiviraju ostale strategije.

Strategija inicijalizacije se može podesiti i korišćenjem konfiguracionog fajla:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <configuration>    <configSections>    </configSections>  ...    <appSettings>      <add key="DatabaseInitializerForType ConsoleApplication.PersonContext, ConsoleApplication"  value="System.Data.Entity.DropCreateDatabaseAlways`1[[ConsoleApplication.PersonContext, ConsoleApplication]], EntityFramework" />    </appSettings>  ...  </configuration> |

Podatak ConsoleApplication.PersonContext odnosi se na pun naziv kontekstne klase, dok je ConsoleApplication naziv *assemblyja* u kome se takva klasa nalazi.

**Koja se strategija inicijalizacije baze podataka podrazumevano koristi od strane EF Code FirstAPI-a?**

 CreateDatabaseIfNotExists

 DropCreateDatabaseIfModelChanges

 DropCreateDatabaseAlways

**Custom strategija inicijalizacije**

EF Code First API omogućava potpuno samostalno kreiranje strategije inicijalizacije koja odgovara potrebama aplikacije koja se razvija. Za aktiviranje custom strategije inicijalizacije može se napisati nešto ovako:

|  |  |
| --- | --- |
| 1  2  3  4 | protected override void OnModelCreating(DbModelBuilder modelBuilder)  {      Database.SetInitializer(new PersonDBInitializer());  } |

Linija unutar OnModelCreating metode dočarava na koji način je moguće aktivirati custom strategiju inicijalizacije baze podataka. Da bi ta linija funkcionisala, neophodno je postojanje custom klase sa nazivom PersonDBInitializer. Takva klasa se može kreirati na dva načina. Prvi način podrazumeva nasleđivanje neke od klasa koje predstavljaju strategiju za inicijalizaciju. Drugi način podrazumeva implementiranje interfejsa IDatabaseInitializer.

Custom strategija inicijalizacije se može kreirati ukoliko postoji potreba za kreiranjem nekih osnovnih podataka prilikom kreiranja baze. Takvi podaci se mogu doživeti kao test podaci, dok custom klasa inicijalizacije uglavnom nasleđuje neku od postojećih klasa. Sledeći primer to ilustruje:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class PersonDBInitializer : CreateDatabaseIfNotExists<PersonContext>  {      protected override void Seed(PersonContext context)      {          IList<Person> people = new List<Person>();          Country russia = new Country() { Name = "Russia", Capital = "Moscow" };          Country usa = new Country() { Name = "USA", Capital = "Washington" };          people.Add(new Person() { FirstName = "John", LastName = "Lord", DateOfBirth = new DateTime(1960, 03, 02), Country = usa });          people.Add(new Person() { FirstName = "Ivan", LastName = "Peters", DateOfBirth = new DateTime(1993, 05, 07), Country = russia });          foreach (Person person in people)              context.People.Add(person);          context.SaveChanges();      }  } |

Primerom je prikazana klasa koja nasleđuje CreateDatabaseIfNotExists klasu, koja predstavlja istoimenu strategiju inicijalizacije. Za ubacivanje podataka u bazu, prilikom njene inicijaliazcije koristi se metoda Seed. Stoga se i takav proces zove *Seed*.

U primeru je unutar klase custom inicijalizacije pregažena metoda Seed, a unutar metode je smeštana logika za unos inicijalnih podataka u bazu. Prvo je napravljena jedna kolekcija, sa nazivom people, a zatim je takva kolekcija napunjena podacima. Podatke predstavljaju dve osobe i dve države. Na kraju je unutar jedne foreach petlje izvršeno dodavanje kreiranih objekata tipa Person u kontekst i pozivanje metode SaveChanges.

***Napomena:****ne zaboravite da je potrebno ovako kreiranu strategiju inicijalizacije i aktivirati na sledeći način:*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class PersonContext : DbContext  {      public PersonContext() : base("name = TestConnString")      {        }      public DbSet<Person> People { get; set; }      public DbSet<Country> Countries { get; set; }      protected override void OnModelCreating(DbModelBuilder modelBuilder)      {          Database.SetInitializer(new PersonDBInitializer());      }  } |

Drugi značajan razlog zbog koga se kreira custom strategija inicijalizacije jeste samostalno definisanje načina na koji će baza podataka biti inicijalizovana. U takvim situacijama najčešće se custom klasa inicijalizacije dobija implementiranjem interfejsa IDatabaseInitializer. Sledeći primer ilustruje način na koji je moguće kreirati custom klasu inicijalizacije sa specifičnom strategijom kreiranja baze.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class PersonDBInitializer : IDatabaseInitializer<PersonContext>      {          public void InitializeDatabase(PersonContext context)          {              if (context.Database.Exists())              {                  if (!context.Database.CompatibleWithModel(true))                  {                      context.Database.Delete();  context.Database.Create();                  }              }              else              {                  context.Database.Create();              }              context.Database.ExecuteSqlCommand("CREATE TABLE GLOBAL\_DATA([KEY] VARCHAR(50), [VALUE] VARCHAR(250))");          }      } |

U prikazanom primeru custom klasa implementira interfejs IDatabaseInitializer i redefiniše metodu InitializeDatabase. Ova metoda prihvata instancu kontekstne klase. Unutar ove metode prvo se proverava da li baza podataka postoji metodom Exists. Ukoliko baza podataka postoji, proverava se da li je šema baze podataka kompatibilna sa objektnim modelom. Ukoliko baza podataka nije kompatibilna sa objektnim modelom, postojeća baza podataka se briše i kreira nova. Takođe, ukoliko baza podataka ne postoji, opet se vrši kreiranje nove baze. Tako se baza podataka ne inicijalizuje samo u situacijama kada postoji i kada je kompatibilna sa objektnim modelom.

Na kraju metode InitializeDatabase dodata je i jedna linija koja izvršava custom SQL komandu, pozivanjem metode ExecuteSqlCommand. Ova linija kreira jednu tabelu sa nazivom *GLOBAL\_DATA*. Takva tabela na učestvuje u objektnom modelu, a njena uloga u ovim primeru je da prikaže kako je moguće proces inicijalizacije prilagoditi sopstvenim potrebama, dodavanjem custom logike.

**Onemogućavanje inicijalizacije**

Proces inicijalizacije baze podataka je u potpunosti moguće isključiti. Za takvim nečim se može javiti potreba u produkcionim uslovima, kada postoji rizik od gubitka podataka, ponovnim generisanjem baze podataka.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class PersonContext : DbContext  {      public PersonContext() : base("name = TestConnString")      {          Database.SetInitializer<PersonContext>(null);      }      public DbSet<Person> People { get; set; }      public DbSet<Country> Countries { get; set; }  } |

Identičan efekat je moguće postići i korišćenjem konfiguracionog fajla, i to na sledeći način:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <configuration>    <configSections>    </configSections>  ...    <appSettings>      <add key="DatabaseInitializerForType ConsoleApplication.PersonContext, ConsoleApplication" value="Disabled" />    </appSettings>  ...  </configuration> |

Inicijalizacija se unutar konfiguracionog fajla isključuje unutar sekcije appSettings, dodavanjem add elementa. Ključ koji se koristi je DatabaseInitializerForType. Prvi podatak se odnosi na puno ime klase koja predstavlja kontekst, a drugi na naziv sklopa u kome se klasa nalazi.

**Migracija podataka**

U dosadašnjem toku lekcije bilo je reči o različitim strategijama inicijalizacije baze podataka. Jedna od značajnijih tema ove lekcije je kako reagovati na promene na objektnom modelu. Već je prikazano kako se korišćenjem različitih strategija može rukovati ovakvim situacijama. Tako je moguće odabrati strategiju koja će u slučaju nekompatibilnosti baze i objektnog modela vršiti ponovno kreiranje baze podataka. Ipak, ovakva metoda je prilično invanzivna po podatke, s obzirom na to da svi podaci, uskladištene procedure, okidači i ostali objekti u bazi bivaju obrisani.

Od EF 4.3 verzije uvedena je funkcionalnost koja omogućava automatsko ažuriranje konceptulanog modela, ali bez gubitka bilo kakvih podataka ili objekata baze podataka. Ta nova strategija se zove *MigrateDatabaseToLatestVersion*.

Generalno se može reći da postoje dva tipa migracija:

* + automatska migracija,
  + migracija zasnivana na kodu.

**Automatska migracija**

Korišćenjem automatske migracije, EF Code First API će voditi računa o sinhronizaciji objektnog i konceptulanog modela automatski, a pritom neće doći do gubitka podataka ili objekata baze podataka. Automatsku migraciju je potrebno aktivirati pre prvog pokretanja aplikacije koja se zasniva na Code First pristupu, i to tako što se unutar [Package Manager Console](https://www.link-elearning.com/linkdl/opisPojma.php?id=142457) prozora unese sledeća komanda:

enable-migrations –EnableAutomaticMigration:$true

Izvršavanjem prikazane komande unutar aktivnog projekta kreira se Migrations folder sa Configuration klasom. Configuration klasa ima sledeći sadržaj:

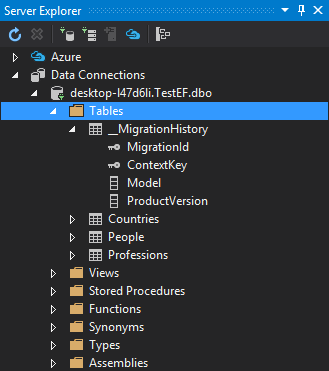
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | internal sealed class Configuration :   DbMigrationsConfiguration<ConsoleApplication.PersonContext>      {          public Configuration()          {              AutomaticMigrationsEnabled = true;          }          protected override void Seed(ConsoleApplication.PersonContext context)          {              //  This method will be called after migrating to the latest version.          }      } |

Interesantno je primetiti da se unutar konstruktora klase Configuration vrednost svojstva AutomaticMigrationsEnabled postavlja na true i na taj način aktivira automatska migracija.

S obzirom na to da ja automatska migracija samo još jedna od strategija za inicijalizaciju baze podataka, i da bi funkcionisala, neophodno je da se aktivira unutar kontekstne klase. To se postiže na sledeći način:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class PersonContext : DbContext  {      public PersonContext() : base("name = TestConnString")      {       Database.SetInitializer(new MigrateDatabaseToLatestVersion<PersonContext,  Migrations.Configuration>("TestConnString"));      }      public DbSet<Person> People { get; set; }      public DbSet<Country> Countries { get; set; }  } |

Uvidom u strukturu kreirane baze podataka može da se primeti postojanje tabele *\_\_MigrationHistory*, kao na slici 16.1.



*Slika 16.1 – Struktura baze kreirane Code First pristupom*

Ovo je tabela koju EF koristi za rukovanje automatskom migracijom, tako što u njoj beleži istoriju promena unutar baze podataka.

Automatska migracija podrazumevano funkcioniše za dodavanje kompletnih entiteta ili nekih novih svojstava klasa modela. Ukoliko se pokuša uklanjanje postojećeg svojstva nekog od entiteta, EF Code First API će emitovati izuzetak tipa AutomaticDataLossException. Razlog je i više nego očigledan. U situacijama kada se uklanja neko od postojećih svojstava za koje postoje podaci u bazi, doći će do gubitka podataka. Automatska migracija je konfigurisana tako da inicijalno ne dozvoljava gubitak bilo kakvih podataka. Stoga je tako nešto neophodno eksplicitno omogućiti, a to se postiže postavljanjem svojstva  AutomaticMigrationDataLossAllowed na true, unutar konstruktora Configuration klase.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | internal sealed class Configuration :  DbMigrationsConfiguration<ConsoleApplication5.PersonContext>      {          public Configuration()          {              AutomaticMigrationsEnabled = true;              AutomaticMigrationDataLossAllowed = true;         }  ...  } |