**Korišćenje EF-a u nepovezanom okruženju**

**00:19:57**

Jedinica: 13 od 19

**Rezime**

* EF dozvoljava dva scenarija prilikom rada sa podacima: povezani scenario i nepovezani scenario.
* Povezani scenario podrazumeva korišćenje jednog kontekstnog objekta za čitanje i upisivanje podataka.
* Povezani scenario ima smisla koristiti posebno kod onih aplikacija koje sve objekte, kojima rukuju tokom čitavog životnog veka, čuvaju u memoriji.
* Kada se koristi nepovezani scenario, entiteti se čitaju iz baze podataka korišćenjem jednog kontekstnog objekta, a izmenjeni entiteti se bazi prosleđuju korišćenjem potpuno novog kontekstnog objekta.
* Nepovezani scenario savršeno odgovara web aplikacijama kod kojih se kontekstni objekat uništava kada se stranica isporuči klijentu.
* Stanje entiteta se definiše korišćenjem tipa EntityState.
* EF API poseduje nekoliko vrlo značajnih metoda, koje se koriste za uključivanje entiteta u kontekst, od kojih su najznačajnije DbSet.Add(), DbSet.Attach() i DbContext.Entry().

U svim lekcijama do sada, za rad sa podacima pomoću EF-a korišćen je takozvani povezani pristup. Rad sa podacima započinjao je kreiranjem kontekstne klase, zatim je sledilo učitavanje entiteta i eventualna obrada, a na kraju su promene perzistirane korišćenjem identične kontekstne klase, koja je iskorišćena i za dobavljanje entiteta.

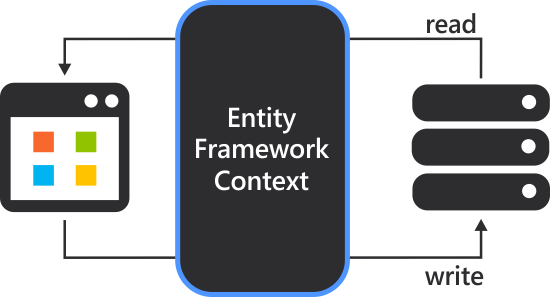
EF je moguće koristiti na još jedan način, koji podrazumeva korišćenje različitih kontekstnih objekata za čitanje i za kasnije čuvanje objekata.

Iz svega navedenog može se zaključiti da EF dozvoljava dva scenarija prilikom rada sa podacima:

* + povezani scenario,
  + nepovezani scenario.

**Povezani scenario**

Kao što je već rečeno, svi primeri do sada prikazani realizovani su korišćenjem povezanog scenarija. Izvršavanje osnovnih CRUD operacija korišćenjem povezanog modela je prilično lak posao, s obzirom na to da kontekst automatski prati stanja entiteta tokom njihovog čitavog životnog veka. Tako se operacije čitanja i pisanja podataka korišćenjem povezanog scenarija mogu prikazati slikom 13.1.

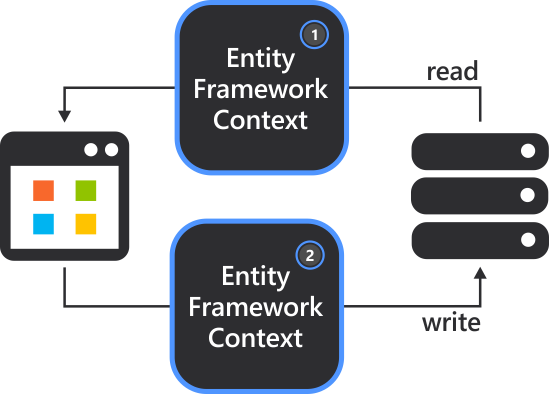


*Slika 13.1 – Čitanje i pisanje podataka korišćenjem jednog objekta konteksta*

Slika 13.1 prikazuje kako se jedan kontekstni objekat koristi za čitanje i upisivanje podataka. Povezani scenario ima smisla koristiti posebno kod aplikacija koje sve objekte kojima rukuju, tokom čitavog životnog veka čuvaju u memoriji. Tipičan primer takvih aplikacija su desktop aplikacije, koje su tokom svog celog životnog veka učitane u radnu memoriju računara. Ipak, situacija je značajno drugačija kod web aplikacija. Objekti ovakve aplikacije tipično žive samo onoliko koliko je potrebno da bi se obradio zahtev i stranica isporučila klijentu. U takvim situacijama mnogo je prikladnije koristiti nepovezan scenario za rad sa podacima.

**Nepovezani scenario**

Kada se koristi nepovezani scenario, entiteti se čitaju iz baze podataka korišćenjem jednog kontekstnog objekta, a izmenjeni entiteti se bazi prosleđuju korišćenjem potpuno novog kontekstnog objekta. Ovakav scenario je prikazan slikom 13.2.



*Slika 13.2 – Korišćenje različitih kontekstnih objekata za čitanje i pisanje*

Sa slike 13.2 se može videti da je prvi kontekstni objekat iskorišćen za operaciju čitanja. Nakon završene operacije čitanja, ovakav kontekstni objekat se uništava. Posle obrade entiteta od strane aplikacije, oni se bazi prosleđuju korišćenjem potpuno novog kontekstnog objekta koji je obeležen brojem 2.

Ovakav scenario savršeno odgovara web aplikacijama kod kojih se kontekstni objekat uništava kada se stranica isporuči klijentu. U takvim situacijama, kada se od klijenta dobiju podaci, koji se pretvore u objektne entitete, potrebno je napraviti potpuno novi kontekstni objekat kako bi se obavila perzistencija takvih objekata. Ipak, kontekstni objekat, koji se kreira na taj način, nema nikakvo znanje o entitetima koji mu se priključuju. Situacija je bila potpuno drugačija kod povezanog scenarija, jer je u tom slučaju kontekstni objekat pratio stanja svih objekata upisanih u kontekst. Kod nepovezanog scenarija, tako nešto nije slučaj, pa programer mora samostalno voditi računa o praćenju stanja entiteta.

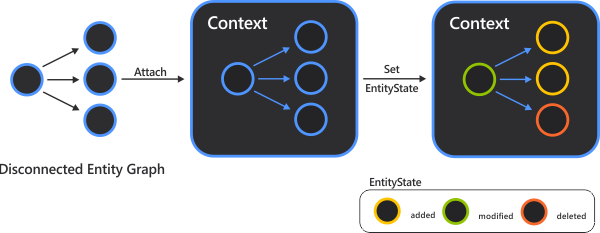
**Ukoliko se za manipulaciju podacima koristi više kontekstnih objekata, vrlo je verovatno da je obrada podataka obavljena u:**

 povezanom okruženju

 nepovezanom okruženju

**Rad sa nepovezanim entitetima**

Pre nego što se steknu uslovi za rad sa entitetima korišćenjem EF-a u nepovezanom okruženju, neophodno je uraditi dve stvari. Prvo je potrebno povezati sve nepovezane entitete sa novim kontekstnim objektom. Na taj način će novi kontekst postati svestan postojanja entiteta. Nakon dodavanja entiteta kontekstu potrebno je definisati odgovarajuće stanje entiteta korišćenjem tipa EntityState zato što novi kontekstni objekat nema nikakvo znanje o operacijama koje su prethodno izvršene nad nepovezanim entitetima. Slika 13.3 prikazuje ilustrativno dva upravo opisana koraka.



*Slika 13.3 – Uključivanje entiteta u novi kontekst i definisanje stanja*

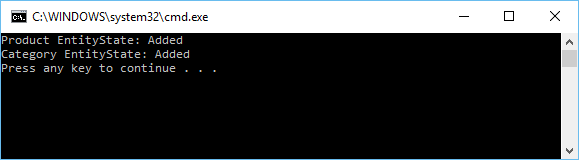
EF API poseduje nekoliko vrlo značajnih metoda koje se koriste za sprovođenje opisanih akcija. U nastavku će biti prikazane ove metode.

**DbSet.Add()**

Metoda Add, DbSet klase, koristi se za priključivanje entiteta ili [stabla entiteta](https://www.link-elearning.com/linkdl/opisPojma.php?id=142491) novom kontekstu. Tako priključeni entiteti automatski dobijaju Added stanje. Sledeći kod ilustruje priključenje entiteta kontekstu.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | //disconnected entity graph   Product disconnectedProduct = new Product() { ProductName = "DisProduct", Discontinued = false };   disconnectedProduct.Category = new Category() { CategoryName = "DemoCat", Description = "DemoCat Description" };   using (var ctx = new NORTHWNDEntities())   {       //add disconnected Product entity graph to new context instance - ctx       ctx.Products.Add(disconnectedProduct);  // get DbEntityEntry instance to check the EntityState of specified entity       var productEntry = ctx.Entry(disconnectedProduct);       var categoryEntry = ctx.Entry(disconnectedProduct.Category);       Console.WriteLine("Product EntityState: {0}", productEntry.State);       Console.WriteLine("Category EntityState: {0}", categoryEntry.State);   } |

U prikazanom primeru prvo se vrši instanciranje Product entiteta, kome se priključuje i Category entitet. Na taj način se formira stablo entiteta. Nakon kreiranja stabla entiteta, instancira se i kontekstna klasa, a za dodavanje entiteta kreiranom kontekstu, koristi se metoda Add. Metoda Add obavlja dodavanje kompletnog stabla entiteta u kontekst. Nakon dodavanja entiteta kontekstu dobavljaju se instance DbEntityEntry klase kako bi se pročitalo stanje upravo dodatih entiteta. Tako će nakon izvršenja prikazanog koda izgled konzolnog prozora biti kao na slici 13.4.



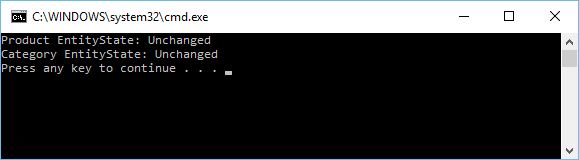
*Slika 13.4 – Stanja entiteta nakon uključivanja u kontekst*

**DbSet.Attach()**

Još jedna metoda koja se može koristiti za priključenje entiteta kontekstu je metoda Attach, klase DbSet. Sledeći primer ilustruje njeno korišćenje.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | //disconnected entity graph   Product disconnectedProduct = new Product() { ProductName = "DisProduct", Discontinued = false, CategoryID = 999 };   disconnectedProduct.Category = new Category() { CategoryName = "DemoCat", Description = "DemoCat Description", CategoryID = 999 };   using (var ctx = new NORTHWNDEntities())   {       //add disconnected Product entity graph to new context instance - ctx       ctx.Products.Attach(disconnectedProduct);  // get DbEntityEntry instance to check the EntityState of specified entity       var productEntry = ctx.Entry(disconnectedProduct);       var categoryEntry = ctx.Entry(disconnectedProduct.Category);       Console.WriteLine("Product EntityState: {0}", productEntry.State);       Console.WriteLine("Category EntityState: {0}", categoryEntry.State);   } |

Primer je praktično identičan prethodnom, u kome se upotrebljava metoda Add. Ovoga puta su prilikom definisanja entiteta Product i Category definisane vrednosti primarnog ključa Category i stranog ključa Product entiteta. Jednostavno, metoda Attach se koristi u situacijama kada entiteti već postoje upisani u bazu podataka, pa se stoga očekuje da poseduju vrednosti primarnih i stranih ključeva. Iz toga se može zaključiti da se metoda Attach koristi na entitetima koji su već učitani iz baze podataka korišćenjem nekog drugog kontekstnog objekta. U prilog svemu ovome govori i činjenica da entiteti dodati u kontekst metodom Attach, imaju stanje Unchanged, što se može videti i na slici 13.5.



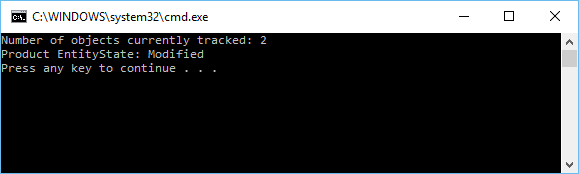
*Slika 13.5 – Stanja entiteta učitanih korišćenjem metode Attach*

**DbContext.Entry()**

U situacijama kada je potrebno eksplicitno postaviti stanje entiteta, moguće je koristiti objekat klase DbEntityEntry. Da bi se došlo do instance ove klase, koristi se metoda Entry, klase DbContext. Ova metoda vraća objekat tipa DbEntityEntry, koji je u prethodnim primerima korišćen za čitanje stanja entiteta. Pored čitanja, on ima i mogućnost upisivanja stanja entiteta. Sledeći primer to  ilustruje:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | //disconnected entity graph  Product disconnectedProduct = new Product() { ProductName = "DisProduct", Discontinued = false, CategoryID = 999 };  disconnectedProduct.Category = new Category() { CategoryName = "DemoCat", Description = "DemoCat Description", CategoryID = 999 };  using (var ctx = new NORTHWNDEntities())  {      var productEntry = ctx.Entry(disconnectedProduct);      productEntry.State = EntityState.Modified;      Console.WriteLine("Number of objects currently tracked: " + ctx.ChangeTracker.Entries().Count());      Console.WriteLine("Product EntityState: {0}", productEntry.State);  } |

Prilikom eksplicitnog definisanja stanja entiteta, on automatski postaje član konteksta, zajedno sa svim pripadajućim objektima objektnog stabla. To se može videti i na slici 13.6, koja prikazuje rezultat izvršavanja prikazanog koda. U datom kodu prvo je kreirana instanca DbEntityEntryklase korišćenjem metode Entry, a zatim je promenjeno stanje objekta. Na kraju su navedene dva linije za ispis poruka – prva za ispis ukupnog broja objekata koji se prate od strane konteksta, a druga koja ispisuje stanje entiteta, tipa Product.



*Slika 13.6 – Prikaz poruka o objektima unutar konteksta*

Sa slike 13.6 se jasno vidi da su oba entiteta, koja čine jedno stablo entiteta, dodata kontekstu, a takođe i to da je stanje objekta Modified.

**Dodavanje novog entiteta**

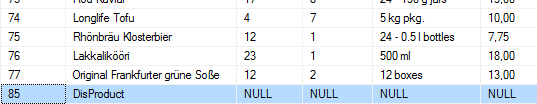
Iz dosadašnjeg izlaganja jasno je da se u situacijama, kada je entitet nov, ne upisan u bazu, koristi metoda Add. Stoga će u narednom primeru biti prikazano kako se entitet može kreirati, priključiti kontekstu i upisati u bazu.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //disconnected entity graph  Product disconnectedProduct = new Product() { ProductName = "DisProduct", Discontinued = false };             using (var ctx = new NORTHWNDEntities())             {                 ctx.Products.Add(disconnectedProduct);                 ctx.SaveChanges();             } |

Identičan efekat je mogao biti postignut i eksplicitnim definisanjem stanja entiteta na Added, kao u sledećem primeru:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //disconnected entity graph  Product disconnectedProduct = new Product() { ProductName = "DisProduct", Discontinued = false };  using (var ctx = new NORTHWNDEntities())  {      ctx.Entry(disconnectedProduct).State = System.Data.Entity.EntityState.Added;      ctx.SaveChanges();  } |

Nakon izvršavanja bilo kojeg od prikazanih primera, u bazu će biti upisan kreiran entitet, kao na slici 13.7.



*Slika 13.7 – Nov red u bazi podataka*

**Ažuriranje postojećeg entiteta**

Nekada će se javiti potreba za ažuriranjem entiteta koji je prethodno već učitan u aplikaciju. Sledeći primer ilustruje takvu situaciju.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | Product product;  Get product from DB  using (var ctx = new NORTHWNDEntities())  {      product = ctx.Products.Where(p => p.ProductID == 1).FirstOrDefault();  }  Change product name in disconnected mode (out of ctx scope)  if (product != null)  {      product.ProductName = "UpdatedName";  }    Save modified entity using new Context  using (var ctx = new NORTHWNDEntities())  {      Mark entity as modified      ctx.Entry(product).State = System.Data.Entity.EntityState.Modified;      Call SaveChanges      ctx.SaveChanges();  } |

U prikazanom primeru upotrebljene su dve instance kontekstne klase. Prvom instancom kontekstne klase iz baze podataka učitan je proizvod sa vrednošću ProductID kolone 1. Tako je jedan entitet smešten u prvi kontekstni objekat. Napuštanjem using bloka, unutar koga je kontekst instanciran, završava se i životni vek prvog konteksta.

Kada se život prvog konteksta završi, vrši se izmene na učitanom entitetu, i to ovoga puta u nepovezanom okruženju. Izmene koje se vrše, odnose se na naziv proizvoda. Da bi se sada ovako izmenjen entitet opet perzistirao, kreira se novi kontekst koji nema nikakvo saznanje o entitetu koji je prethodno učitan i izmenjen. Stoga mu je potrebno staviti do znanja šta se dogodilo sa entitetom. To se postiže prilikom njegovog priključivanja novom kontekstu, tako što se njegovo stanje postavlja na Modified. Pozivanjem metode SaveChanges, sve promene nastale u nepovezanom okruženju čuvaju se u bazi podataka.

**Brisanje entiteta**

Ukoliko je potrebno obrisati entitet koji je učitan korišćenjem nekog drugog konteksta, odnosno objekat koji ne pripada tekućem kontekstu, dovoljno je postaviti njegovo stanje na Deleted, kao u sledećem primeru:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Product product;  using (var ctx = new NORTHWNDEntities())  {      product = ctx.Products.Where(p => p.ProductID == 85).FirstOrDefault();  }  using (var ctx = new NORTHWNDEntities())  {      ctx.Entry(product).State = System.Data.Entity.EntityState.Deleted;      ctx.SaveChanges();  } |

**Tehnike za utvrđivanje stanja**

Iz dosadašnjeg toka lekcije moglo je da se nasluti da je utvrđivanje stanja objekta zahtevan posao. Možda to na prvi pogled tako ne izgleda, s obzirom na to da su navedeni primeri trivijalni. Ipak, u kompleksnim sistemima neophodno je postojanje određenog mehanizma koji bi se koristio za utvrđivanje stanja objekta kojim je rukovano u nepovezanom okruženju.

Postoji veliki broj načina na koji je moguće realizovati praćenje stanja na nivou entiteta, ali će u nastavku ove lekcije biti prikazan samo jedan, koji podrazumeva postojanje svojstva za praćenje stanja u svakom entitetu.

Da bi se tako nešto realizovalo, najbolje je odmah na početku napraviti interfejs, koji bi svi entiteti implementirali. Interfejs bi mogao da izgleda ovako:

|  |  |
| --- | --- |
| 1  2  3  4 | interface IEntityObjectState  {      EntityState ObjectState { get; set; }  } |

Sledeći korak u omogućavanju entitetima da prate svoje stanje, jeste implementacija interfejsa od strane svih klasa entiteta, kao u sledećem bloku koda.

Product klasa entiteta:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public partial class Product : IEntityObjectState  {      public int ProductID { get; set; }      public string ProductName { get; set; }      public Nullable<int> SupplierID { get; set; }      public Nullable<int> CategoryID { get; set; }      public string QuantityPerUnit { get; set; }      public Nullable<decimal> UnitPrice { get; set; }      public Nullable<short> UnitsInStock { get; set; }      public Nullable<short> UnitsOnOrder { get; set; }      public Nullable<short> ReorderLevel { get; set; }      public bool Discontinued { get; set; }      public byte[] RowVersion { get; set; }      public virtual Category Category { get; set; }      [NotMapped]      public EntityState ObjectState { get; set; }  } |

Category klasa entiteta:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public partial class Category : IEntityObjectState  {      public Category()      {          this.Products = new HashSet<Product>();      }      public int CategoryID { get; set; }      public string CategoryName { get; set; }      public string Description { get; set; }      public byte[] Picture { get; set; }      public virtual ICollection<Product> Products { get; set; }      [NotMapped]      public EntityState ObjectState { get; set; }  } |

Prikazane klase entiteta implementiraju ranije kreiran interfejs pod nazivom IEntityObjectState. To ih obavezuje na definisanje svojstva sa nazivom ObjectState tipa EntityState, što je enumeracija kojom se definišu stanja entiteta.

Sada je prilikom rukovanja entitetima u nepovezanom okruženju potrebno voditi računa o njihovom stanju, postavljanjem vrednosti za upravo kreirano svojstvo. Naredni primer će ilustrovati ažuriranje entiteta u nepovezanom okruženju, čije stanje će biti čuvano korišćenjem upravo kreiranog mehanizma.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Product product;  using (var ctx = new NORTHWNDEntities())  {      product = ctx.Products.Where(p => p.ProductID == 1).FirstOrDefault();  }  if (product != null)  {      product.ProductName = "UpdatedName";      product.ObjectState = EntityState.Modified;  }  using (var ctx = new NORTHWNDEntities())  {      ctx.Entry(product).State = product.ObjectState;      ctx.SaveChanges();  } |

Naveden primer je identičan onom koji je prikazan nešto ranije, prilikom demonstracije ažuriranja entiteta u nepovezanom okruženju. Entitet se čita korišćenjem jednog kontekstnog objekta, a upisuje u bazu korišćenjem drugog. Pritom se, dok je objekat nepovezan, obavlja ažuriranje jednog njegovog svojstva. Prilikom ažuriranja svojstva ProductName, ovoga puta se postavlja i vrednost svojstva ObjectState, kojim se prati stanje objekta. Na kraju se, neposredno pre pozivanja metode SaveChanges, obavlja uključivanje objekta u novi kontekst postavljanjem njegovog stanja, koje se čita iz svojstva  ObjectState.