**Transakcije**

**00:13:03**

Jedinica: 18 od 19

**Rezime**

* Transakcija je jedinica posla koja se sastoji od jedne ili više komandi baze podataka koje moraju da se izvrše kao jedinica.
* Implicitna transakcija je ona koju ne definiše korisnik i u kojoj ne postoji mogućnost kontrole od strane korisnika.
* Eksplicitne transakcije su one koje se eksplicitno definišu u kodu aplikacije.
* Pozivanjem metode SaveChanges, EF automatski izvršava operaciju kao jednu transakciju.
* Database.BeginTransaction() je metoda koja omogućava započinjanje i završavanje transakcije unutar postojećeg kontekstnog objekta.
* Database.UseTransaction() je metoda koja omogućava postojećem kontekstu da koristi transakciju započetu izvan Entity Frameworka.
* Pre EF6, preporučen način za kreiranje transakcija podrazumevao je upotrebu TransactionScope objekta.

U svim verzijama EF-a prilikom pozivanja metode SaveShanges, bilo da se na taj način sprovodi unos, ažuriranje ili brisanje podataka, operacija nad podacima biva izvršena atomično.

|  |
| --- |
| ***Napomena:***  *Šta znači atomično?*  *U atomičnoj transakciji serija operacija nad bazom podataka će se ili u potpunosti izvršiti ili se uopšte neće izvršiti, ukoliko bar jedna od njih ne uspe. Dakle, da bi promene u bazi bile zapamćene, sve operacije iz jedne transakcije moraju biti uspešno izvršene.* |

**Šta je transakcija?**

Transakcija je jedinica posla koja se sastoji od jedne ili više komandi baze podataka koje moraju da se izvrše kao jedinica.

Transakcija predstavlja operaciju nad podacima i ima sledeće osobine:

* + atomičnost (Atomicity),
  + konzistentnost (Consistency),
  + izolacija (Isolation),
  + trajanje (Duration).

Navedene osobine su poznate i pod pojmom **ACID**.

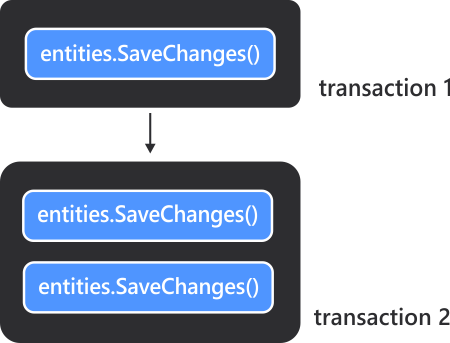
* + *Atomičnost* – transakcija je nedeljiva i može da se izvrši u potpunosti ili da se uopšte ne izvrši.
  + *Konzistentnost* – transakcija mora da radi na konzistentan način i mora da ostavi podatke u konzistentnom stanju.
  + *Izolacija* – transakcija bi trebalo da bude izolovana, u smislu da efekti te transakcije koja se trenutno izvršava nisu vidljivi drugim transakcijama. Bilo koji posao koji se trenutno izvršava ne sme biti vidljiv drugim transakcijama, sve dok se ta transakcija ne potvrdi.
  + *Trajanje* – kada se transakcija izvrši, rezultati ove transakcije će ostati trajni i preživeće pad sistema.

**Tipovi transakcija**

Transakcije se mogu podeliti na sledeće kategorije:

* 1. *Implicitne transakcije* –  transakcije koje ne definiše korisnik i u kojima ne postoji mogućnost kontrole od strane korisnika. SQL Server automatski startuje novu implicitnu transakciju, pošto se tekuća transakcija potvrdi ili poništi.
  2. *Eksplicitne transakcije* – one koje se eksplicitno definišu u kodu aplikacije. One omogućavaju da grupiše više SQL naredbi i da se izvrše kao jedna celina ili da se izvrši opoziv svih operacija. Koriste se kako bi se implementirala specifična pravila integriteta podataka aplikacije.
  3. *Distribuirane transakcije* – slične eksplicitnim transakcijama, s tim što se odnose na dva ili više izvora podataka. Distribuirana transakcija osigurava ACID svojstva nad više sistema ili izvora podataka. U Windows operativnim sistemima postoji DTC (Distributed Transaction Coordinator), transakcioni menadžer koji upravlja distribuiranim transakcijama.

Slika 18.1 ilustruje dve transakcije. Prva transakcija je implicitna zato što se postiže pozivanjem jedne SaveChanges metode. Druga transakcija na slici 18.1 je eksplicitna, jer grupiše dva poziva SaveChanges metoda. Da ova dva poziva nisu grupisana u jednu transakciju, bila bi izvršena kao dve zasebne transakcije.



*Slika 18.1 – Implicitna i Eksplicitna transakcija*

**Kako se nazivaju transakcije koje okruženje samostalno kreira?**

 Implicitne

 Eksplicitne

 Distribuirane

**Transakcije i Entity Framework**

Već je rečeno da pozivanjem metode SaveChanges EF automatski operaciju izvršava kao jednu transakciju. Na taj način se može govoriti o implicitnim transakcijama.

Pitanje je na koji način objediniti više operacija, tako da se one izvrše atomično, kao jedna transakcija. Pre EF6, jedini način za grupisanje više operacija u jednu transakciju bio je korišćenje TransactionScope klase ili ObjectContext.Connection svojstva. EF6 donosi nove opcije koje omogućavaju rad sa transakcijama, i to:

* + Database.BeginTransaction() – metoda koja omogućava započinjanje i završavanje transakcije unutar postojećeg kontekstnog objekta. Na taj način je moguće kombinovanje više operacija u jednu transakciju, što omogućava njihovo atomično izvršavanje.
  + Database.UseTransaction() – omogućava postojećem kontekstu da koristi transakciju započetu izvan Entity Frameworka.

**Implicitne transakcije**

Već više puta je rečno da se pozivanjem metode SaveChanges operacije izvršavaju unutar jedne transakcije. Kada se pozove SaveChanges metoda, Entity Framework kreira novu transakciju za sve zadate operacije. Ukoliko npr. instanca DbContext tipa sadrži dva zahteva za dodavanje novih kontakata, tri zahteva za update i jedan za brisanje, svih šest operacija će biti deo jedne implicitne transakcije nad bazom podataka. Ukoliko su sve naredbe u ovoj fazi izvršene uspešno, transakcija će biti primenjena na bazu. Ali, ukoliko nastane neka greška tokom izvršavanja SQL komandi, transakcija će biti opozvana, a podaci vraćeni u početno stanje.

Potpis metode SaveChanges je sledeći:

|  |  |
| --- | --- |
| 1 | public virtual int SaveChanges() |

Kao što se može videti, metoda vraća vrednost tipa int, koja ukazuje na broj objekata upisanih u bazu podataka.

**Kombinovanje više operacija u jednu transakciju**

Korišćenje Database.BeginTransaction metode omogućava objedinjavanje više operacija unutar jedne transakcije, ali i podešavanje nivoa izolacije. Tako ova metoda poseduje dva oblika – jedan koji prihvata nivo izolacije izražen enumeracijom IsolationLevel i drugi koji ne prihvata parametre i postavlja podrazumevani nivo izolacije.

Metoda Database.BeginTransaction vraća DbContextTransaction objekat koji poseduje metode Commit i Rollback, koje omogućavaju izvršavanje ili poništavanje transakcije.

Objekat tipa DbContextTransaction je nakon upotrebe potrebno osloboditi, tako da se instanciranje ovog objekta obavlja uglavnom unutar using bloka, koji automatski poziva Disposemetodu.

Sledeći primer ilustruje kako se korišćenjem tipa DbContextTransaction dve operacije mogu objediniti u jednu transakciju.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using (var ctx = new NorthwindEntities())          {              using (var dbContextTransaction = ctx.Database.BeginTransaction())              {                  try                  {  Employee emp = ctx.Employees.Where(e => e.EmployeeID == 14).First();                      emp.FirstName = "NewName2";                      ctx.SaveChanges();                      ctx.Database.ExecuteSqlCommand(                          @"UPDATE Employees SET FirstName = NewName2" +                              " WHERE EmployeeID = 13"                          );                      ctx.SaveChanges();                      dbContextTransaction.Commit();                  }                  catch (Exception exc)                  {                      dbContextTransaction.Rollback();                  }              }          } |

U prikazanom kodu, unutar već dobro poznatog using bloka, koji definiše objekat konteksta, napravljen je još jedan using blok, ali ovoga puta za instanciranje objekta tipa DbContextTransaction koji metoda BeginTransaction vraća. Na ovaj način je započeta transakcija.

Unutar using bloka transakcije definisane su dve naredbe za manipulaciju podacima. Prva naredba definisana je korišćenjem LINQ to Entities jezika, a njom se vrši izmena zapisa sa rednim brojem 14 iz tabele Employees.

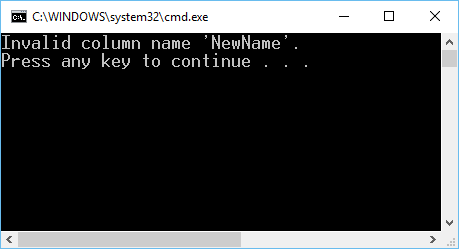
Druga naredba je definisana korišćenjem izvornog SQL jezika i njen efekat jeste ažuriranje zapisa sa identifikacionim brojem 13, u tabeli Employees.

Nakon definisanja naredbi, poziva se metoda SaveChanges za perzistenciju promena, a tek nakon nje i metoda Commit, kojom se transakcija završava. Ukoliko tokom opisanih koraka dođe do bilo kakvog izuzetka, on se hvata definisanim catch blokom, i transakcija se poništava, a svi podaci se vraćaju na stanje pre početka transakcije. Na taj način se osigurava da će definisane operacije biti atomične.

Ukoliko izvršite prikazani kod, primetićete da se ništa ne dešava. Uvidom u bazu podataka može se videti da podaci nisu promenjeni. Kako bi se stekao bolji uvid u efekat koda, može se dodati jedna linija za ispis eventualnog izuzetka.

|  |  |
| --- | --- |
| 1  2  3  4  5 | catch (Exception exc)  {      Console.WriteLine(exc.Message);      dbContextTransaction.Rollback();  } |

Nakon ponovnog izvršenja prikazanog koda, stvari postaju mnogo jasnije, što se vidi na slici 18.2.

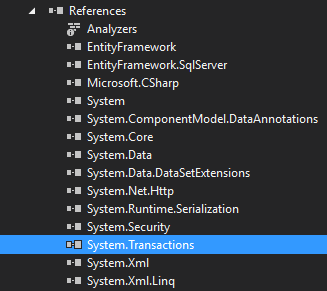


*Slika 18.2 – Poruka o grešci do koje je došlo prilikom izvršavanja transakcije*

Jednostavno, prilikom definisanja SQL naredbe, napravljena je greška, ali ne slučajno. Ovim je ilustrovano da se, i pored toga što je prva naredba potpuno legitimna, njen efekat nije primenio na podacima nakon završetka transakcije. Da je kojim slučajem identičan kod bio prikazan izvan transakcije, efekat prve naredbe bi bio vidljiv u bazi, odnosno dve definisane operacije ne bi bile atomične.

**Transkacije korišćenjem TransactionScope klase**

Pre EF6, preporučen način za kreiranje transakcija podrazumevao je upotrebu TransactionScopeobjekta. Da bi se TransactionScope klasa koristila, neophodno je prethodno dodati referencu na prostor imena u kojem se ona nalazi. Prostor imena je System.Transactions, a nalazi se u istoimenom [DLL](https://www.link-elearning.com/linkdl/opisPojma.php?id=142446) fajlu.



*Slika 18.3 – System.Transactions DLL*

Dakle, prvo je potrebno referencirati DLL prikazan na slici 18.3, a onda unutar dokumenta dodati using direktivu.

Nakon uključivanja prostora imena potrebnog za korišćenje TransactionScope klase, ona se može upotrebiti u kodu. Najčešći oblik TransactionScope bloka je sledeći:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using (var scope = new TransactionScope())  {    try    {  ...      //do something      scope.Complete();    }      catch (Exception exc)    {  ...    }  } |

Uzimajući u obzir blok koda koji je naveden, kod ekvivalentan malopre prikazanom može izgledati ovako:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using (var ctx = new NorthwindEntities())  {      using (var scope = new TransactionScope())      {            try          {       Employee emp = ctx.Employees.Where(e => e.EmployeeID == 14).First();              emp.FirstName = "NewName2";              ctx.SaveChanges();              ctx.Database.ExecuteSqlCommand(                  @"UPDATE Employees SET FirstName = NewName2" +                      " WHERE EmployeeID = 13"                  );              ctx.SaveChanges();              scope.Complete();          }          catch (Exception exc)          {              Console.WriteLine(exc.Message);          }      }  } |

U prikazanom kodu, na sličan način kao u prethodnom primeru, kreirana je jedna transakcija, ali ovoga puta korišćenjem TransactionScope klase. Objekat ove klase instanciran je unutar jednog using bloka, koji direktno sadrži try i catch blokove. Unutar try bloka nalazi se logika transakcije. Transakcija objedinjuje dve operacije. Na kraju try bloka nalazi se linija kojom se transakcija završava. To je linija u kojoj se poziva metoda Complete. Tek u ovom trenutku efekat svih operacija unutar transakcije biće vidljiv i u bazi podataka. Ukoliko izvršavanje koda ne dođe do ove linije, transakcija neće biti izvršena.