**Performanse**

**00:12:31**

Jedinica: 19 od 19

**Rezime**

* Pod pojmom performansi podrazumeva se – koliko brzo, sigurno i pouzdano neka aplikacija obavlja postavljene zadatke.
* Sa povećanjem apstrakcije aplikacije i broja slojeva koji se koriste za njeno realizovanje, generalno se usporava izvršavanje aplikacije.
* Jedna od osnovnih stvari o kojoj je potrebno voditi računa jeste dobavljanje samo podataka koji su nam zaista i potrebni. Cena čekanja na server da pronađe i  isporuči podatke preko mreže je vrlo visoka.
* EF automatski obavlja keširanje sirovih upita koji se upućuju serveru baze podataka.
* Podrazumevani tip učitavanja EF-a je Lazy Loading, na šta se može uticati.
* Veoma efikasna metoda po pitanju performansi, koja se može koristiti za čitanje jednog entiteta jeste metode Find.
* Korišćenjem AsNoTracking metode moguće je dobaviti Read-Only entitet koji neće biti praćen od strane konteksta.

Kada se pomenu performanse, misli se na sledeće – koliko dobro neka aplikacija obavlja postavljene zadatke. Pritom se pod dobrim obavljanjem podrazumeva brzina, sigurnost i pouzdanost. U ovoj lekciji biće reči o tehnikama koje mogu da pomognu u razvijanju upravo takvih aplikacija.

Veoma je lako prenebregnuti pitanje performansi prilikom korišćenja EF-a, s obzirom na to da većinu stvari on obavlja potpuno automatski. Stoga je jedan od značajnijih zadataka prilikom upoznavanja sa EF-om razumevanje situacija koje mogu da pogoršaju performanse. U prvom delu ove lekcije biće izneti faktori koji utiču na performanse.

**Faktori koji utiču na performanse**

Postoje mnogi izvori potencijalnih narušilaca performansi. Nedostatak radne memorije, spora internet veza, loši vremenski uslovi, samo su neki od faktora koji mogu da naruše performanse sistema. Navedeni faktori se ipak mogu nazvati spoljnim faktorima, i zato će u ovoj lekciji pažnja biti posvećena onim faktorima koji se tiču korišćenja EF-a.

**Aplikativni slojevi**

Sa povećanjem apstrakcije aplikacije i broja slojeva koji se koriste za njeno realizovanje, generalno se usporava izvršavanje aplikacije. Ono što se možda na prvi pogled ne vidi jeste to da EF obavlja značajnu količinu posla u pozadini kako bi sve funkcionalnosti radile besprekorno. Činjenica je da EF aplikaciji dodaje minimalno tri sloja:

* + upit se mora transformisati iz LINQ to Entities ili Entity SQL oblika u izvornu naredbu;
  + prevedeni upit se propušta kroz EntityClient komponentu;
  + odgovor se korišćenjem EntityClienta prevodi u oblik pogodan za korišćenje u objektnom modelu.

Sva tri navedena sloja obavljaju značajan posao u smanjenju logike koju je potrebo da napiše programer. Ipak, ništa ne dolazi besplatno. Cena koja se plaća za jednostavnost koju EF nudi ogleda se u smanjenju performansi aplikacije. Ipak, uticaj na performanse je u većini slučajeva zanemarljiv u poređenju sa onim što se dobija zauzvrat, tako da se upotreba EF-a u takvim situacijama smatra razumnom. I pored toga, potrebno je imati u vidu rečeno, i eventualno, na kritičnim mestima koristiti direktno [ADO.NET](https://www.link-elearning.com/linkdl/opisPojma.php?id=142492).

**Čitanje velike količine podataka**

Jedna od osnovnih stvari o kojoj je potrebno voditi računa jeste dobavljanje samo podataka koji su nam zaista i potrebni. Cena čekanja na server da pronađe i isporuči podatke preko mreže je vrlo visoka. Ukoliko bi svaki klijent pokušavao da pročita više podataka nego što mu je zaista potrebno, takvo ponašanje bi vrlo verovatno dovelo server u ozbiljan problem.

Zbog svega navedenog, gotovo uvek je potrebno izbegavati čitanje svih kolona neke tabele, korišćenjem karaktera \*, kao u primeru:

|  |  |
| --- | --- |
| 1 | SELECT \* FROM Products |

Mnogo je prikladnije, ukoliko nam nisu potrebne sve kolone, napisati jedan ovakav upit:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT      ProductName      ,QuantityPerUnit      ,UnitPrice      ,UnitsInStock      ,UnitsOnOrder      ,Discontinued    FROM Products |

Iako na prvi pogled drugi upit izgleda znatno složenije od prvog, on je mnogo efikasniji, zato što čita samo neophodne podatke.

Takođe, podatke je poželjno maksimalno filtrirati na serveru, što će naravno smanjiti potrebu za njihovim filtriranjem na klijentu. Primeri jednog filtriranog i nefiltriranog upita dati su u nastavku:

Nefiltriran upit:

|  |  |
| --- | --- |
| 1 | SELECT ProductName, UnitPrice FROM Products |

Filtriran upit:

|  |  |
| --- | --- |
| 1 | SELECT ProductName, UnitPrice FROM Products WHERE UnitPrice > 20 |

**Keširanje plana upita**

Svaki put kada EF izvršava upit, takav posao prolazi kroz niz faza. Jedna od najznačajnijih je konvertovanje upita višeg nivoa u sirove SQL naredbe. Stoga, pre nego što se upusti u takav posao, EF proverava da je identičan upit u prošlosti već kompajliran. Ukoliko jeste, kompajliranje se neće ponavljati, već će se iskoristiti keširana verzija sirovog upita. Ukoliko se keširana verzija ne pronađe, upit se kompajlira i kešira.

EF podudaranje upita detektuje na osnovu teksta i kolekcije parametara, pri čemu se misli na naziv i tip parametara. Takođe, prilikom provere uzimaju se obzir velika i mala slova.

*EF plan caching* je podrazumevano omogućen za sve Entity SQL i LINQ to Entities upite. U nekim situacijama je ipak korisno ovakvo keširanje onemogućiti. Kada se tako nešto kaže, prevashodno se misli na situacije kada se izvršava upit za koji se zna da neće biti izvršavan često. Sledeći primer ilustruje način na koji je moguće isključiti keširanja plana upita:

|  |  |
| --- | --- |
| 1  2  3 | var result = from t in ctx.Products\_by\_Categories select t;         ObjectQuery oQuery = result as ObjectQuery;         oQuery.EnablePlanCaching = false; |

**Tipovi učitavanja – Eager Loading VS Lazy Loading**

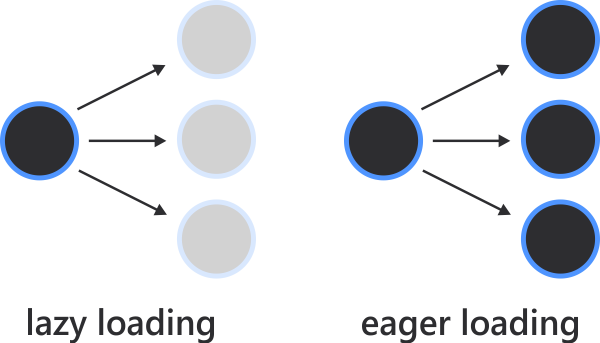
Kada se govori o učitavanju podataka, mogu se razlikovati dva tipa:

* + Lazy Loading,
  + Eager Loading.

Podrazumevani tip učitavanja EF-a je Lazy Loading. Po njemu, entiteti se učitavaju samo kada se eksplicitno i zatraže. Dakle, dobitak u performansama značajan zato što se učitavaju samo podaci koji su potrebni, i to onda kada su potrebni.

Eager Loading je drugačiji tip učitavanja, po kome se dobavljaju ne samo podaci koji su zatraženi već i oni koji su u relaciji sa zatraženim podacima korišćenjem navigacionih svojstava.

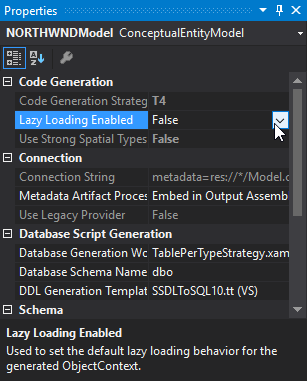
Slika 19.1 dočarava razliku između ova dva tipa učitavanja.



*Slika 19.1 – Lazy VS Eager Loading*

U većini slučajeva napredak na polju performansi donosi Lazy učitavanje, te je stoga ono i podrazumevano aktivirano. U nekim situacijama, ipak, Eager učitavanje može doneti poboljšanje performansi. Stoga je o aktiviranju Eager Loading učitavanja moguće razmišljati kada je unapred poznato da će unutar upita biti korišćena navigaciona svojstva.

Kako bi se isključilo Lazy Loading učitavanje, dovoljno je unutar Properties prozora kompletnog modela postaviti svojstvo Lazy Loading Enabled na false, kao na slici 19.2.



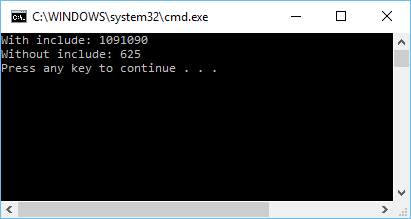
*Slika 19.2 – Onemogućavanje Lazy Loadinga*

Takođe, Eager Loading se može aktivirati i na nivou upita, i to korišćenjem Include metode IQueryable interfejsa.

U narednom primeru biće prikazan uporedni test identičnog upita izvršenog dva puta –  prvi put sa dodatkom Include metode, a drugi put bez nje. Pritom će biti merena brzina izvršavanja obe naredbe:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using (NORTHWNDEntities ctx = new NORTHWNDEntities())    {      var watch = new Stopwatch();      long totalTicks = 0;      watch.Restart();      var res = (from p in ctx.Products.Include("Category")          where p.UnitPrice > 20          select p);      watch.Stop();      totalTicks += watch.ElapsedTicks;      Console.WriteLine("With include: {0}", watch.ElapsedTicks);      watch.Restart();      var res2 = (from p in ctx.Products          where p.UnitPrice > 20          select p);      watch.Stop();      totalTicks += watch.ElapsedTicks;      Console.WriteLine("Without include: {0}", watch.ElapsedTicks);    } |

Rezultat koji proizvodi prikazan kod može se videti na slici 19.3.



*Slika 19.3 – Vremena izvršavanja upita sa i bez korišćenja Includea*

**Koji tip učitavanja podataka EF podrazumevano koristi?**

 Lazy loading

 Eager loading

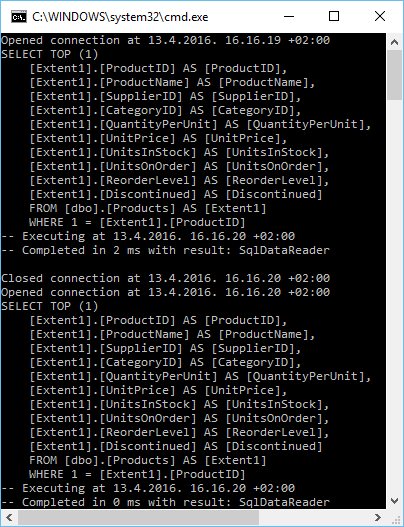
**Čitanje jednog entiteta korišćenjem ključa**

Veoma efikasna metoda po pitanju performansi, koja se može koristiti za čitanje jednog entiteta, jeste metode Find. Ova metoda očekuje jedan argument koji predstavlja primarni ključ entiteta. Kada se aktivira, ona će pretragu prvo obaviti unutar konteksta. Ukoliko pronađe traženi entitet, čita ga i isporučuje direktno iz konteksta. Ukoliko entitet ne bude pronađen unutar konteksta, pretraga se nastavlja unutar skladišta. Ukoliko entitet ne bude pronađen ni u okviru skladišta, metoda Find vraća vrednost NULL. Na kraju, metoda Find prilikom pretraga uzima u obzir i entitete koji nisu još upisani u skladište, a dodati su u kontekst.

U nastavku će biti prikazan primer čitanja jednog entiteta korišćenjem dve različite tehnike. Prvo će entitet biti pročitan korišćenjem klasičnog LINQ to Entities upita formiranog korišćenjem lambda izraza.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | using (NORTHWNDEntities ctx = new NORTHWNDEntities())  {      ctx.Database.Log = Console.Write;      var queryOne = ctx.Products.First(p => p.ProductID == 1);      var queryTwo = ctx.Products.First(p => p.ProductID == 1);  } |

U primeru su definisana dva izraza za čitanje jednog istog entiteta korišćenjem LINQ to Entities tehnologije. Na početku using bloka definisana je linija kojom se aktivira logovanje, tako da će se u konzoli ispisivati sirove SQL komande koje okruženje upućuje serveru. Kada se primer pokrene, efekat je kao na slici 19.4.



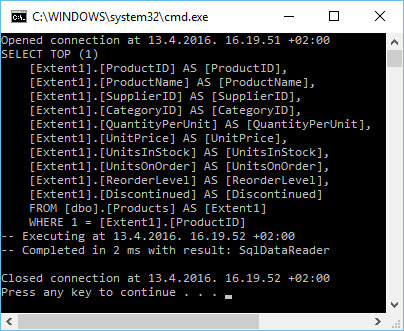
*Slika 19.4 – Log SQL komandi koje EF upućuje serveru*

Sa slike 19.4 je više nego jasno da je EF uputio dva [SELECT upita](https://www.link-elearning.com/linkdl/opisPojma.php?id=142496) SQL serveru. Iz ovoga se može zaključiti da iako je oba puta zatražen identičan entitet, EF je čitanje obavio iz skladišta.

Identičan efekat je moguće postići korišćenjem nešto ranije pomenute metode Find. Sledi primer:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | using (NORTHWNDEntities ctx = new NORTHWNDEntities())  {      ctx.Database.Log = Console.Write;      var queryOne = ctx.Products.First(p => p.ProductID == 1);      var queryTwo = ctx.Products.Find(1);  } |

Efekat prikazanog koda može se videti na slici 19.5.



*Slika 19.5 – Log SQL komande koju je EF uputio serveru*

Sa slike 19.5 se može videti da je u ovom primeru EF uputio samo jedan zahtev SQL serveru, što znači da je metoda Find entitet pročitala iz konteksta.

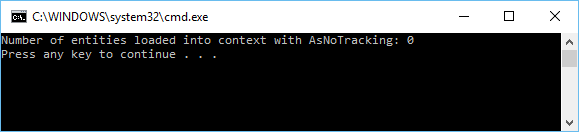
**Kreiranje Read-Only entiteta**

U nekim situacijama, pogotovu kada se razvijaju web aplikacije, biće dovoljno korisnicima omogućiti samo pregled nekih podataka. U takvim situacijama je znatno efikasnije izbeći korišćenje keširanja i praćenja promena, koji se primenjuju na regularnim entitetima. Tako nešto se jednostavno može postići korišćenjem AsNoTracking metode.

Primer učitavanja Read-Only entiteta dat je u nastavku:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using (NORTHWNDEntities ctx = new NORTHWNDEntities())              {                  var productsNoTracking = (from p in ctx.Products                                            where p.UnitPrice > 20                                            select p).AsNoTracking();      Console.WriteLine("Number of entities loaded into context with AsNoTracking: {0}",  ctx.ChangeTracker.Entries().Count());              } |

Efekat prikazanog koda je kao na slici 19.6.



*Slika 19.6 – Poruka o broju entiteta koji se prate*

Sa slike 19.6 je jasno uočljivo da se unutar konteksta ne nalazi nijedan objekat, i to upravo zbog primene AsNoTracking metode.