Unit of Work

Ljubić, Ante

Master's thesis / Diplomski rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:913712

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-29**



Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science







Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Diplomski studij matematike i računarstva

Ante Ljubić

Unit of work

Diplomski rad

Osijek, 2017.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Diplomski studij matematike i računarstva

Ante Ljubić

Unit of work

Diplomski rad

Mentor: izv. prof. dr. sc. Domagoj Matijević
Komentor: Ivana Šimić, mag. educ. math. et inf.

Osijek, 2017.

Sažetak. U ovom radu ćemo pokazati što je to Unit of Work, kako se implementira i na koji način koristi. Unit of Work je usko vezan uz transakcije na bazama podataka, pa ćemo iz tog razloga ukratko pojasniti što su to transakcije i kako funkcionira klijent-poslužitelj arhitektura. Implementacija je odrađena na poslužitelju u ASP.NET Web API softverskom okviru koristeći njegovu komponentu Entity framework te ćemo ukratko reći nešto i o tome.

Primjenu ćemo pokazati u posljednjem poglavlju. Objasniti ćemo ju uz jednostavne primjere iz aplikacije koja je izrađena u sklopu ovoga rada.

Ključne riječi: Unit of Work, transakcije, ASP.NET Web API, Entity framework

Summary. In this paper we will show what is Unit of Work, how we implement it and how can it be used. In general, Unit of Work is related with a database transactions, so we will explain transactions and client-server architecture. We will show Unit of Work implementation on ASP.NET Web API using Entity framework.

In the last chapter we will show how it's used. We will explain it on simple examples taken from the application build for that purpose.

Key words: Unit of Work, transactions, ASP.NET Web API, Entity framework

Sadržaj

1. Uvod **1**

2. Transakcije i klijent-poslužitelj arhitektura **3**
2.1. Transakcije 3
2.2. Klijent-poslužitelj arhitektura 4

3. REST **6**

4. ASP.NET Web API **9**

5. Unit of Work **11**
5.1. Entity framework 11
5.2. Klase i objekti 13
5.3. Primjer Unit of Work implementacije bazirane na Entity frameworku . 13
5.4. Korištenje Unit of Work paterna 18

6. Praktični dio rada **20**
6.1. Popis korištenih tehnologija 20
6.2. Struktura aplikacije 21
6.3. Primjer korištenja Unit of Work-a 21
6.4. Primjer u kojem Unit of Work nije potreban 27

7. Zaključak **29**

8. Životopis **32**

1. Uvod

Razvoj internetske mreže ili "World Wide Weba" imao je dubok utjecaj na živote ljudi cijeloga svijeta. U početku, web je pretežno samo pohranjivao informacije koje su bile dostupne i imao je slab utjecaj na programske sustave. Ti sustavi su bili pokretani na lokalnim računalima i bili su dostupni samo unutar organizacija. Oko 2000. godine web se počeo razvijati i sve se više funkcionalnosti dodavalo u web preglednike, što je ujedno značilo i da su se mogli početi razvijati web sustavi i aplikacije koje će se pokretati u njima. To je dovelo do razvoja brojnih proizvoda koji su pružali inovativne usluge, a pristupiti im se moglo jednostavno preko interneta. Ti proizvodi su uglavnom bili financirani samo reklamama koje su prikazivali, odnosno nisu zahtijevali nikakve novčane naknade, te su ih korisnici mogli besplatno pregledavati i koristiti. Detaljnije o tome možemo vidjeti u literaturi [9].

Osim što su se masovno počeli razvijati sistemski programi, razvoj web preglednika doveo je i do evolucije poslovnih i organizacijskih softvera. Naime, umjesto dostavljanja softvera korisniku na osobno računalo, softver se postavljao na web poslužitelj, a koristio se unutar web preglednika. Tome je pridonijela činjenica da oni, osim prikaza sadržaja, mogu obrađivati podatke na lokalnom računalu. Time su nadogradnje i popravci na softveru postali puno lakši i jeftiniji jer više nije bilo potrebe instalirati softver na svako računalo. Zbog toga su se mnoge tvrtke u gotovo svim segmentima poslovanja počele okretati korištenju web verzijama aplikacija.

Slijedeći korak u razvoju web sustava je pojava web servisa, odnosno komponente softvera koja omogućuje specifičnu i korisnu funkcionalnost, a dostupna nam je preko weba. Aplikacije su konstruirane tako da koriste te web servise, a koje ne mora pružati samo jedna tvrtka. U načelu, ne mora se ni prilikom svakog pokretanja aplikacije koristiti isti servis.

Posljednjih godina se tako razvio i pojam "softver kao servis" (eng. *Software as a service*). To su zapravo oni servisi koji su nam potrebni da se pokrenu i koriste web aplikacije. Dostupni su nam u "računalnom oblaku" odnosno u "cloud"-u (eng. *Cloud*), a pristupa im se preko interneta. Cloud je sustav koji povezuje velik broj računala spojenih preko interneta i omogućava mrežni pristup dijeljenim računalnim servisima. Tako, na primjer, kada šaljemo e-poštu preko web preglednika, koristimo web servise iz računalnog oblaka. Korisnicima je taj sustav dostupan i ne plaćaju ga, odnosno plaćaju samo one dijelove koje koriste ili ih dobivaju besplatno u zamjenu za pregledavanje reklama.

Pojava interneta i razvoj web aplikacija dovela je do značajne promjene u načinu organizacije poslovnog sustava. Prije weba, poslovne aplikacije su bile uglavnom kom-

paktne, jedinstvene i nedjeljive, pokretane na pojedinačnim računalima, a komunikacija se obavljala samo unutar organizacija. No, sada je softver široko rasprostranjen, ponekad i širom cijelog svijeta. Značajne izmjene u organizaciji softvera su tako dovele i do promjene u samom razvoju web sustava. Poslovne aplikacije se više ne razvijaju od temelja nego uključuju iskorištavanje već prije razvijenih komponenti i programa te se tako sada stvaranje novog sustava svodi na sklapanje postojećih komponenti u odgovarajuću cjelinu.

Razvijanja sustava koji se svode na sklapanje postojećih komponenti u cjelinu dovode do potrebe za novim pojednostavljenjima, a jedni od njih su softver dizajn paterni. Naime, softver dizajn patern je višestruko iskoristivo rješenje za rješavanje problema koji se učestalo pojavljuju. Više o paternima se može pronaći u literaturi [6]. Oni se ne mogu direktno unositi u programski kôd kao gotova rješenja, nego služe kao predložak koji se može koristiti u mnogim različitim situacijama, a "Unit of work" je jedan od njih.

Unit of Work je usko vezan uz transakcije na bazama podataka, tako da ćemo u prvom poglavlju najprije reći što su to transakcije. Nastavit ćemo s klijent-poslužitelj arhitekturom kako bi u nadolazećim poglavljima bilo jasno o čemu govorimo kada se spominju klijenti i poslužitelji. Spomenuti ćemo neke osnovne detalje o REST-u jer se komunikacija između klijenta i poslužitelja odvija putem tog protokola, a u sljedećem poglavlju ćemo reći nešto i o poslužitelju, odnosno o ASP.NET Web API-u. U poglavlju Unit of Work objasniti ćemo što je to Unit of Work, koji su njegovi zadaci te prikazati i objasniti njegovu implementaciju u Entity frameworku.

Na osnovu Unit of Work-a i Entity frameworka je izgrađena aplikacija u sklopu ovoga rada te ćemo u posljednjem poglavlju reći nešto o njoj. Objasniti ćemo strukturu aplikacije i prikazati na nekoliko primjera kako i kada se Unit of Work koristi.

2. Transakcije i klijent-poslužitelj arhitektura

Za razumijevanje glavne teme rada potrebno je najprije razumjeti transakcije te arhitekturu korištenu u praktičnom dijelu rada. Ovo poglavlje donosi osnovne definicije i principe vezane za navedene pojmove.

2.1. Transakcije

Transakcija je unaprijed definirana procedura na bazama podataka. Ona predstavlja jednu logičku cjelinu akcija koje moraju biti zajedno odrađene. Postoje 4 osnovna svojstva transakcija[1]:

- Atomičnost (eng. *Atomicity*) - Ili su sve promjene na bazi primjenjene ili nije niti jedna.
- Konzistentnost (eng. *Consistency*) - Prije izvršavanja transakcije baza se smatra konzistentnom, te takva mora ostati i nakon. Za vrijeme transakcije smatramo da baza nije konzistentna.
- Izolacija (eng. *Isolation*) - Za vrijeme izvršavanja jedne transakcije, druge transakcije čitaju podatke koji su bili u bazi podataka prije nego što je prva transakcija počela.
- Trajnost (eng. *Durability*) - Sigurnost da su, nakon uspješnog izvršavanja transakcije, sve promjene imale utjecaja na bazu.

Kada dohvaćamo, dodajemo, uređujemo ili brišemo podatke iz baze podataka, redoslijed operacija se može kontrolirati korištenjem transakcija[2].

Primjer 2.1 *Prodaja jedne avionske karte za zrakoplovnu kompaniju može predstavljati samo jednu transakciju, ali je obično u sklopu putovanja uključeno više letova. Recimo, ako je uz odlazak uključen povratak ili ako je potrebno više presjedanja bit će potrebno nekoliko operacija unosa u bazu.*

Kako bi se očuvao integritet baze, transakcija mora biti u potpunosti izvršena ili uopće ne smije biti izvršena.

Stoga, transakcija koja iz bilo kojeg razloga nije do kraja obavljena mora biti poništena, tj. svim podacima, koje je ona do trenutka prekida promijenila, se moraju vratiti početne vrijednosti.

[1] Detaljnije o svojstvima transakcija pogledati u literaturi 8.
[2] Više o transakcijama se može pronaći u literaturi [12].

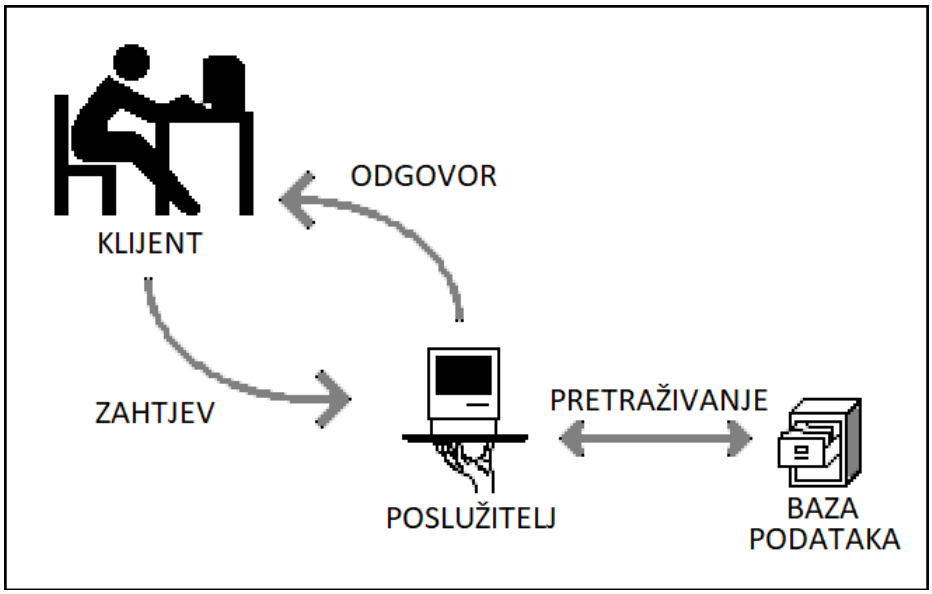
2.2. Klijent-poslužitelj arhitektura

Klijent-poslužitelj arhitektura[3] je vrsta komunikacije u kojoj sudjeluju minimalno dva računala. Od toga jedno računalo potražuje podatke od drugoga. Računalo koje potražuje podatke zovemo "klijent"(eng. *client*), a računalo koje ih poslužuje nazivamo "poslužitelj"[4]. Pri tome više klijenata može potraživati podatke od istog poslužitelja.

Kako bismo razumjeli što su upiti i odgovori možemo to prikazati i jednostavnim primjerom:

Primjer 2.2 *Promotrimo jednostavan scenarij, a to je da korisnik interneta unese www.google.com u web browseru. Time će mu se već unutar nekoliko milisekundi prikazati Google početna stranica, odnosno dobiti će odgovor za svoj jednostavan upit.*

Dakle, nakon što je upit poslan sa klijenta na poslužitelja, poslužitelj ga analizira i kreira odgovor koji se vraća klijentu.



Slika 1: Klijent-poslužitelj arhitektura - ilustracija[5]

Pogledajmo komunikaciju koja se događa u pozadini scenarija iz navedenog primjera. Kada je korisnik upisao u web preglednik "www.google.com", klijentsko računalo

[3]Detaljnije o klijent-poslužitelj arhitekturi potražite u literaturi [19].
[4]Naziv dolazi od engleske riječi "server" čiji je korijen riječi "serve", a to znači posluživati.
[5]Slika preuzeta sa HitecHTube.com ([7])

je poslužitelju poslalo zahtjev za prikazom stranice na toj adresi. Poslužitelj je pročitao podatke iz zahtjeva i na temelju toga poslao odgovor klijentskom računalu. Kada je odgovor stigao na klijentsko računalo, ono je korisniku prikazalo google tražilicu, te ju korisnik može koristiti.

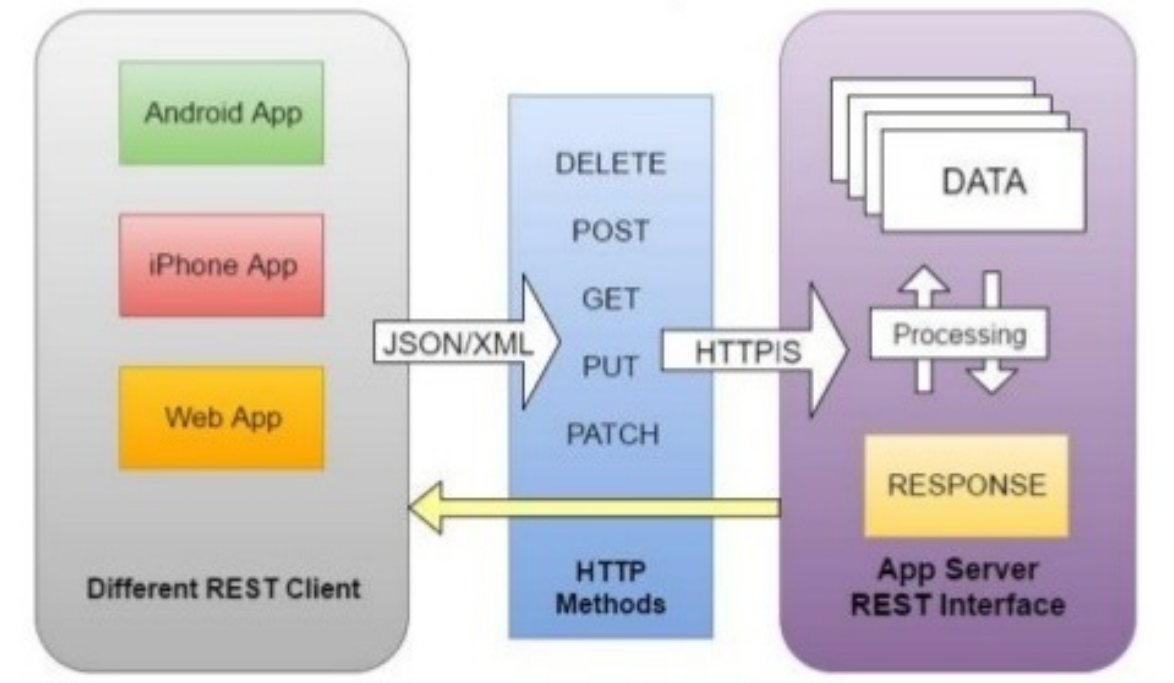
Međutim, često su upiti nešto kompliciraniji te zahtijevaju od poslužiteljskog računala da dohvati ili promijeni podatke koje je uobičajeno čuvati u bazama podataka. Poslužitelj bi tada prilikom obrade klijentskog zahtjeva morao od baze podataka zatražiti da odradi određene akcije, primjerice, uređivanje postojećih podataka, čitanje podataka ili unos novih. Zatim bi klijentskom računalu u odgovoru poslao rezultat akcija, a nakon toga bi klijentsko računalo krajnjem korisniku prikazalo rezultat njegova zahtjeva.

3. REST

Representational State Transfer[6] ili REST je predložena arhitektura koja služi za komunikaciju između različitih tehnologija gdje se pod komunikacijom smatra dohvaćanje podataka sa poslužitelja, dodavanje, izmjena ili brisanje postojećih podatka.

U svakom se zahtjevu s klijenta moraju nalaziti svi podaci koji su poslužitelju potrebni za obradu i odgovor. Naime, nije moguće uređivati neki objekt ako mu se ne proslijedi identifikator ili bilo koja druga oznaka koja ga čini jedinstvenim. Jednako tako se ne mogu dohvatiti podaci ako nije određeno što želimo dohvatiti.

Temeljni princip REST-a je pristup bez čuvanja stanja (eng. *Statelessness*[7]), što znači da klijent sa dobivenim informacijama može upravljati i vršiti operacije na njemu, ali se na poslužitelju neće pohranjivati informacije o klijentu. Odnosno, poslužitelj šalje klijentu podatke u zadanom obliku bez obzira je li klijent mobilna aplikacija, web ili neka druga. Ukratko, poslužitelj ne mora znati detalje o klijentu sa kojim komunicira.



Slika 2: Najbitnije značajke REST arhitekture[8]

[6] Više o REST-u potražite u literaturi [3].
[7] Detaljnije o *Statelessness* pristupu pronađite u literaturi [4].
[8] Slika preuzeta sa SlideShare-a ([13])

Na slici (2) možemo vidjeti najbitnije značajke REST arhitekture. S lijeve strane imamo klijentske aplikacije, a kao što vidimo, one mogu biti pisane za razne uređaje. Razlog tome je što REST ne definira detalje implementacije, nego samo pravila komunikacije. Na desnoj strani slike se nalazi poslužiteljski dio. To je zapravo naš API. Ovdje su u poslužiteljskoj cjelini prikazani i podaci, ali to su zapravo podaci koje dobijemo obradom podataka iz baze. Naime, baza podataka ne pripada poslužiteljskom dijelu aplikacije kod REST arhitekture nego poslužiteljski dio komunicira s njom. Klijentska aplikacija nikada ne komunicira direktno s bazom podataka, nego se komunikacija odvija preko poslužiteljske aplikacije. Zatim se obrađeni podaci, koristeći REST, šalju klijentskoj aplikaciji. Ako obratimo pažnju na sredinu slike, možemo vidjeti metode koje se koriste pri komunikaciji. One ovise o odabranom protokolu koje naše aplikacije koriste za komunikaciju i ovdje su prikazane HTTP metode. Primijetimo kako ni format ne mora biti jedinstven. Naime, format poruka je dobro odabrati prema tehnologijama na koje ciljamo pri izradi aplikacija. Format poruka tako može biti jedan od najčešćih kao što su JSON ili XML, ali tako i HTML, CSV, PNG ili bilo koji drugi binarni format.

Kako bi REST protokol, koji je općenito povezan sa web aplikacijama, mogao komunicirati među tehnologijama najčešće se koristi HTTP (eng. *Hypertext Transfer Protocol*) protokol. Naime, HTTP je aplikacijski protokol na kojemu se zasniva cijeli WWW (eng. *World Wide Web*), a njegove glavne karakteristike su u tome što ga podržavaju brojne platforme i što on omogućuje stvaranje upita i odgovora.

Iako se HTTP najčešće koristi, moguće je koristiti i neke druge protokole za prijenos podataka, recimo SNMP[9], SMTP[10] ili neke druge.

Upiti mogu biti GET, koji služi za dohvaćanje podataka, POST, za unos, PUT, za uređivanje i DELETE za brisanje podataka. Ti upiti su najčešći i najbitniji za komunikaciju iako osim njih postoje i PATCH, OPTIONS, TRACE, HEAD i CONNECT upiti.

Svaki odgovor koji dobijemo od poslužitelja je obilježen status kôdom koji obaviještava klijenta o uspješnosti analiziranja upita i kreiranja odgovora. Tako HTTP status kôdove možemo podijeliti na pet osnovnih vrsta:

- Informativni (1xx)
- Uspješni (2xx)

[9]SNMP - Simple Network Management Protocol - Protokol za nadzor i upravljanje mrežnim uređajima. Detaljnije o SNMP protokolu potražite u literaturi [15].

[10]SMTP - Simple Mail Transfer Protocol - Protokol koji se koristi prilikom slanja i primanja e-mail pošte. Detaljnije o SMTP protokolu se može pronaći u literaturi [18].

- Preusmjeravanje (3xx)
- Greška na klijentu (4xx)
- Greška na poslužitelju (5xx)

Svaka grupa kodova ima svoje detalje. Primjerice, ako je rezultat upita uspješan, status kôd odgovora će biti 200 OK nakon GET upita, a 201 CREATED nakon POST upita. Ako klijent nema dopuštenje da kreira upit, status kôd odgovora će mu biti 403 Forbidden, a ako podaci nisu pronađeni odgovor će biti 404 Not found. Dakle, općenito se uz svaki upit generira status kôd koji najbolje odgovara trenutnoj situaciji.

Ovo je bio samo kratki uvod u REST koji nam je potreban za shvaćanje i ispravno korištenje ASP.NET WebAPI arhitekture.

4. ASP.NET Web API

ASP.NET Web API ili sučelje za programiranje aplikacija (eng. *Application Programming Interface*) na webu je Microsoftov softverski okvir koji služi za kreiranje HTTP servisa.

Strukturna svojstva: ASP.NET Web API je izgrađen na nekoliko odrednica koje su mu osigurale lakše testiranje, modularnost i robusnost[11]. Neke važnije od njih su:

- *Async all the way:* ASP.NET Web API je dizajniran korištenjem asinkronog modela od početka do kraja. To pridonosi povećanoj skalabilnosti aplikacija pisanih u njemu.
- Mogućnost smještanja aplikacija na proizvoljnim web poslužiteljima.
- Ugrađena podrška za Dependency Injection[12] (u daljnjem tekstu DI): ASP.NET Web API podržava sve DI frameworke[13] preko jednostavnog servisnog sučelja.
- Mogućnost testiranja: ASP.NET Web API je dizajniran na način da se skoro svi dijelovi frameworka mogu testirati.

Naime, ASP.NET Web API nam omogućuje kreiranje HTTP servisa, nadahnut je ASP.NET MVC programskim modelom, njegovim konceptom i dizajnom, te je izgrađen sa sličnim apstrakcijama. Tako da oni koji su već upoznati sa ASP.NET MVC-om mogu primijetiti slične implementacije kontrolera, akcija, filtera i drugog. Neke najbolje odrednice ASP.NET MVC-a, kao što su usmjeravanje, povezivanje modela i potvrđivanje ispravnosti podataka su također dio i ASP.NET Web API-a. No, u Web API-u je poboljšana modularnost komponenti, a i osim toga neovisan je o MVC-u, tj. mogu se zajedno koristiti, ali ne moraju.

Glavna razlika između ta dva programska modela je u tome što je MVC baziran na kreiranju Web stranica i on direktno vrati stranicu, odnosno prikaže ju klijentu, dok Web API vraća poruku s informacijama te može povezati više raznih aplikacija.

Primjer 4.1 *Facebook sadrži API kojemu se osim putem web stranice kojoj pristupamo iz browsera može pristupiti i mobilnim aplikacijama kao što su Facebook, FacebookLite ili Messenger.*

[11] Kompletan popis odrednica i detaljnije potražite u literaturi [14].

[12] Dependency Injection - ukratko se može reći da je to dodjeljivanje svojstava jednog objekta drugome. Tako su objekti koji poprimaju svojstva drugih objekata ovisni o njima (eng. *Dependent*). Detaljnije o Dependency Injectionu potražite u literaturi [6].

[13] Framework - programski koncept koji pojednostavljuje strukturiranje kôda.

Kada se sadržajima aplikacije može upravljati preko HTTP-a koristeći metode koje smo već naveli, GET, POST, PUT i DELETE, može se reći da aplikacija sadržava web API koji mogu koristiti druge aplikacije. A kako je HTTP neovisan o platformama, HTTP servise mogu koristiti različiti uređaji na različitim platformama.

Osnovna ideja HTTP servisa je postojanje podataka koji se mogu identificirati svojom jedinstvenom oznakom URI (eng. *Uniform Resource Identifier*[14]).

Primjer 4.2 *Ako se radi o aplikaciji koja upravlja zaposlenicima neke tvrtke, tada je svaki zaposlenik jedan podatak kojim aplikacija upravlja.*

Za dohvaćanje takvog zaposlenika, odnosno recimo njegova profila, koristimo metodu GET, a URI bi nam izgledao na sljedeći način: `http://localhost:57733/api/userProfile/42` gdje nam je `http://localhost:57733/` početna stranica za koju vidimo da je pokrenuta na portu `57733`. `api` je samo oznaka koju možemo postaviti kako bismo identificirali da se radi o pozivu na Web API. Takva oznaka nije nužna ali je dobra praksa koristiti ju. Primjerice, klijentska aplikacija se može nalaziti na `http://localhost:29314`, a API na `http://localhost:57733/api/` te se tako lakše mogu razlikovati. `userProfile` bi bio naziv rute na koju šaljemo upit, a `42` identifikator traženog zaposlenika.

Svaki podatak se može identificirati jedinstvenom oznakom, ali to ne znači nužno da se slanjem upita na zadani URI odradi samo jedna operacija. Tako se kod GET upita može definirati da se osim korisničkog profila dohvate ujedno i korisničke uloge (u daljnjem tekstu role). Kod metode PUT se obično ne odradi samo jedna operacija nego i niz drugih. Na primjer ako želimo urediti korisnika i njegove role moramo osim `userProfile` tablice urediti i tablicu `userRole`. Scenarij je sličan i sa metodom POST, odnosno kod dodavanja novog profila.

Pojedine metode će biti uspješne samo ako se sve akcije izvrše i to u samo određenom poretku. Naime, ne bi imalo smisla i ne bismo mogli korisniku dodijeliti rolu ako nismo prije toga kreirali korisnika. No, što ako kreiranje korisnika bude uspješno, a ne i dodjeljivanje role?

Ako radimo izravno na bazama podataka, koristili bismo transakcije. Tako bismo spriječili nepotpune podatke u bazi. No, mi želimo aplikaciju koristiti neovisno o kojoj se bazi radi, pa tako ne želimo ovisiti o tome znamo li napisati kôd na svakom mogućem dijalektu SQL jezika. Osim toga dobra je praksa da se što više operacija rješava unutar kôda na poslužitelju, jer ga je tako lakše za održavati. ASP.NET i principi rada na njemu navode nas da izbjegavamo kodiranje u bazi podataka. Ovdje nam se stoga kao rješenje nudi ASP.NET patern pod nazivom Unit of Work.

[14]Detalje o URI-u potražite u literaturi [1].

5. Unit of Work

Možemo reći da je Unit of Work alternativa transakcijama. On nam, kao i transakcije, osigurava čuvanje stanja podataka ako neka od akcija ne bude uspješna, tj. osigurava da nam ili sve akcije budu odrađene ili niti jedna ne promijeni podatke u bazi.

Cijeli kôd koji Unit of Work generira se izvršava jednim upitom na bazu, što znači da tako ne narušavamo broj interakcija sa bazom. Ukratko, patern nam omogućuje pisanje transakcija u ASP.NET-u.

Unit of Work se najčešće koristi kada želimo izvesti skup akcija, a ako nam jedna od njih ne bude uspješna da se ništa ne dogodi, tj. da se ni jedna ne izvrši.

Generalno, Unit of Work ima dva važna zadatka[15]:

- Održavanje liste upita na jednom mjestu (lista može sadržavati više upita za unos, uređivanje i brisanje podataka)
- Slanje liste upita kao jednu transakciju na bazu

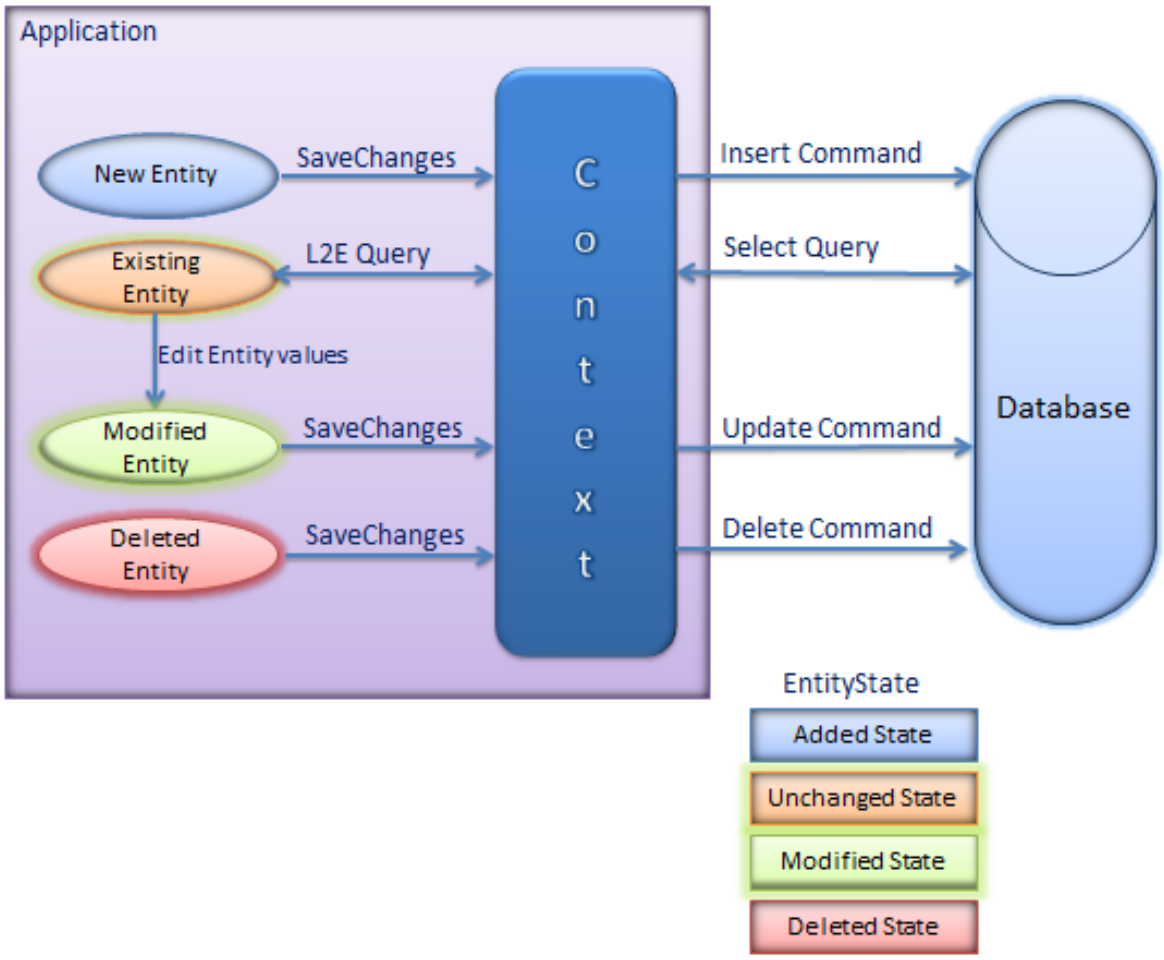
5.1. Entity framework

Entity framework je komponenta u sklopu .NET okruženja[16] koja osigurava objektni pristup programiranju. Ona omogućuje upravljanje podacima u objektnom obliku tako što poveže tablice iz baze podataka sa objektima u kôdu. Generiranje kôda iz baze se može automatski odraditi *database first* metodom gdje se objekti u .NET-u kreiraju na osnovu tablica iz baze, a vrijedi i suprotno, odnosno postoji *code first* metoda koja nam na osnovu kôda može generirati tablice.

Svaki podatak, kojim se upravlja preko entity framework `DbContext` konteksta, čuva svoje stanje i može imati jednu od sljedećih pet vrijednosti:

- Added
- Deleted
- Modified
- Unchanged
- Detached

[15]Detaljnije o zadacima Unit of Work-a u literaturi [2].
[16]Detaljnije o tome što je .NET okruženje potražite u literaturi [17]



Slika 3: Ilustracija statusa objekata kod Entiti framework-a[17]

Kako možemo vidjeti i na slici (3), kada se radi o novom objektu, on ima status `Added`, pa će kontekst izvršiti komande za unos podataka u bazu. Slično, kada dohvatimo podatke iz baze oni poprime `Unchanged` status jer su trenutno nepromijenjeni. No, kada ih izmijenimo, status im se postavi na `Modified`, pa će kontekst izvršiti komande za uređivanje podataka, odnosno `Update` na bazi podataka. Obrisani podaci će imati status `Deleted` za koje će kontekst izvršiti komande za brisanje iz baze podataka, a objekti će imati stanje `Detached` ukoliko nisu vezani za kontekst. Ukratko, `DbContext` kontekst, s obzirom na stanja, gradi i izvršava komande na bazi podataka. Više o upravljanju objektnim statusima potražite u [11], a o praćenju njihovih izmjena u literaturi [5].

[17]Slika preuzeta sa Entity Framework tutoriala ([5])

5.2. Klase i objekti

Kako bi prikazali implementaciju Unit of Work paterna reći ćemo najprije što su to klase, a što objekti.

Objekti su skupovi svojstava koje možemo povezati u smislenu cjelinu, a klase su skupovi pravila koji definiraju te objekte. Najlakše je to prikazati jednostavnim primjerom:

Primjer 5.1 *Ako promatramo sveučilište kao neki sustav, možemo reći kako ga čine studenti i predavači. Svi oni imaju neka zajednička svojstva kao što su ime, prezime, mjesto rođenja, datum i slično. No, to ne znači da ih možemo promatrati kao jednake objekte. Naime, studenti imaju neke specifične osobine kao što su godina studija, predmeti koje je upisao i odgovarajuće ocjene, itd. Jednako tako predavači imaju svoje osobine kao što su predmeti koje predavač predaje, datum zaposlenja, titula, itd. U ovome primjeru postoje dvije osnovne klase: klasa Student i klasa Predavač, dok su objekti ili instance ovih klasa konkretni studenti i predavači sa sveučilišta.*

Dakle, može se reći kako klase predstavljaju opise, odnosno predloške za opis zajedničkih osobina grupe objekata. Klasom se generalizira grupa objekata, tj. zanemaruju se osobine koje nisu zajedničke za grupu objekata u jednom kontekstu, na primjer, u kontekstu jednog sveučilišta nije bitno koju student ima boju kose ili nosi li na predavanje bilježnicu ili ne. Klasa treba sadržavati samo one atribute i metode bitne za sve objekte koje klasa modelira u nekom kontekstu. Sve ostale specifične karakteristike objekata se zanemaruju i ne opisuju klasom.

5.3. Primjer Unit of Work implementacije bazirane na Entity frameworku

Najprije definiramo klasu koju nazovemo `UnitOfWork` te na samom početku instanciramo novi kontekst kao `DbContext` varijablu. Ta varijabla nam predstavlja kontekst[18] baze podataka.

```
protected IMyDbContext DbContext { get; private set; }
```

Nakon toga se definira konstruktor:

```
public UnitOfWork(IMyDbContext dbContext)
{
```

[18] U literaturi[10] možete vidjeti detaljnije o kontekstu baze podataka, odnosno što predstavlja `DbContext`.

```
    if (dbContext == null)
    {
        throw new ArgumentNullException("DbContext");
    }
    DbContext = dbContext;
}
```

Konstruktor je funkcija koja nema povratnu vrijednost i koja se naziva jednako kao i klasa. Svaki puta kada se kreira objekt neke klase zapravo se najprije poziva konstruktor te klase, a najčešće se u konstruktorima odrađuje i inicijalizacija podataka elemenata klase. Konstruktori mogu biti s argumentima ili bez, a ako se ne definiraju podrazumijeva se da je generiran konstruktor bez argumenata. U našem slučaju se proslijedi `DbContext` varijabla kao argument i inicijalizira.

Kako bi upravljali podacima potrebno je definirati metode za to. Metoda za unos podataka se definira na sljedeći način:

```
public Task<int> AddAsync<T>(T entity) where T : class
{
    DbEntityEntry dbEntityEntry = DbContext.Entry(entity);

    if (dbEntityEntry.State != EntityState.Detached)
    {
        dbEntityEntry.State = EntityState.Added;
    }
    else
    {
        DbContext.Set<T>().Add(entity);
    }
    return Task.FromResult(1);
}
```

U njoj se provjeri je li objekt već u kontekstu. Ako je u kontekstu, status će mu biti različit od `Detached`, pa mu samo promijenimo status na `Added`, a ako nije onda ga dodamo na kontekst s `.Add(entity);` izrazom kako bi odmah imao `Added` status.

Vidimo da metoda prima objekt `entity` tipa `T` gdje je `T` klasa. Odnosno ta metoda je generička koja će vrijediti za bilo koju klasu. Kao rezultat vraća `Task` koji kao vrijednost ima broj. Naime, metoda bi mogla vraćati i čisti `Task`, ali dobra je praksa vraćati nekakav rezultat.

Ako želimo uređivati podatke, potrebna nam je `UpdateAsync` metoda:

```
public Task<int> UpdateAsync<T>(T entity) where T : class
{
    DbEntityEntry dbEntityEntry = DbContext.Entry(entity);

    if (dbEntityEntry.State == EntityState.Detached)
    {
        DbContext.Set<T>().Attach(entity);
    }
    dbEntityEntry.State = EntityState.Modified;

    return Task.FromResult(1);
}
```

U toj metodi najprije provjeravamo je li objekt izvan konteksta, a ako je izvan dodamo ga u njega s `.Attach(entity);` izrazom. Svakom objektu koji se proslijedi u ovu metodu dodijelimo status `Modified`.

Za brisanje koristimo sljedeće metode od kojih prva briše skup podataka, a druga podatak određen identifikatorom `ID`:

```
public Task<int> DeleteAsync<T>(T entity) where T : class
{
    DbEntityEntry dbEntityEntry = DbContext.Entry(entity);

    if (dbEntityEntry.State != EntityState.Deleted)
    {
        dbEntityEntry.State = EntityState.Deleted;
    }
    else
    {
        DbContext.Set<T>().Attach(entity);
        DbContext.Set<T>().Remove(entity);
    }

    return Task.FromResult(1);
}
```

```
public Task<int> DeleteAsync<T>(string ID) where T : class
{
    var entity = DbContext.Set<T>().Find(ID);

    if (entity == null)
    {
        return Task.FromResult(0);
    }

    return DeleteAsync<T>(entity);
}
```

Možemo primijetiti da se u prvoj metodi provjerava ima li objekt status `Deleted`. Ako nema dodijelimo mu ga, pa će kontekst pripremiti komande za brisanje iz baze podataka. Ako već ima status `Deleted`, objekt dodamo na kontekst te izrazom `.Remove()` taj objekt označimo za brisanje iz baze podataka zajedno s njegovim referencama.

U drugoj se metodi objekt dohvati iz baze podataka s obzirom na `ID` i samo proslijedi u prvu.

Kao što se može primijetiti, nema implementacije za dohvaćanje podataka, a to je iz tog razloga što nam Unit of Work koristi za uređivanje podataka.

Primjedba 5.1 *Ovaj način implementacije metoda za unos, uređivanje i brisanje podataka specifičan je za aplikaciju rađenu u sklopu ovoga rada jer je kao tip identifikatora u bazi korišten string. Obično se kao tip koristi Guid ili int, a najčešće je to ovisno o bazi podataka i programerskim praksama.*

Osim osnovnih metoda za dodavanje, uređivanje i brisanje koje su gore navedene, kako bi se dovršila transakcija potrebna nam je i `CommitAsync` metoda:

```
public async Task<int> CommitAsync()
{
    int result = 0;
    using (TransactionScope scope = new TransactionScope(TransactionSco
    peAsyncFlowOption.Enabled))
    {
        result = await DbContext.SaveChangesAsync();
        scope.Complete();
```

```
    }
    return result;
}
```

U ovoj metodi se najprije inicijalizira i omogući *TransactionScope* - "transakcijski scope"[19]. Zatim se pozove `SaveChangesAsync()` funkcija na `DbContext` varijabli koja sadrži sve objekte s pripremljenim statusima te se time izvrši cijela transakcija i spreme podaci.

Na kraju, definiramo metodu koja će poništiti cijelu transakciju ako neka od akcija ne bude uspješna:

```
public void Dispose()
{
    DbContext.Dispose();
}
```

Za sve te metode se obično kreira *interface*[20] na sljedeći način:

```
public interface IUnitOfWork : IDisposable
{
    Task<int> AddAsync<T>(T entity) where T : class;

    Task<int> CommitAsync();

    Task<int> DeleteAsync<T>(T entity) where T : class;

    Task<int> DeleteAsync<T>(string ID) where T : class;

    Task<int> UpdateAsync<T>(T entity) where T : class;
}
```

I na kraju Unit of Work factory koja ga kreira:

```
public interface IUnitOfWorkFactory
{
    IUnitOfWork CreateUnitOfWork();
}
```

[19]Transakcijski scope - skup kôda koji sudjeluje u jednoj transakciji. Detaljnije o transakcijskom scope-u potražite u literaturi [10].

[20]Interface - poveznica između dvije ili više zasebnih komponenti. Detaljnije o interface-ima možete pronaći u literaturi [6].

Primjedba 5.2 *Korištena je funkcionalnost iz Ninject Factory*[21] *paketa koja će nam kreirati implementaciju metode* `CreateUnitOfWork`.

5.4. Korištenje Unit of Work paterna

Primjer 5.2 *Razmotrimo sljedeći scenarij: imamo korisnike i njihove role i važno je da svaki korisnik ima pripadajuću rolu koja mu je pridružena.*

Unit of Work nam tu pomaže spremiti sve, korisnika zajedno sa njegovim rolama - ili ništa ne spremiti ako negdje dođe do greške.

Sve što moramo je kreirati Unit of Work u metodi gdje nam je potrebna, na primjer u novoj klasi koju ćemo nazvati `MyClass`:

```
public class MyClass
{
    IUnitOfWorkFactory uowFactory;

    public MyClass(IUnitOfWorkFactory uowFactory)
    {
        this.uowFactory = uowFactory;
    }

    public async Task<int> MyMethod()
    {
        // Kreiraj Unit of Work
        var unitOfWork = uowFactory.CreateUnitOfWork();
    }
}
```

Primjedba 5.3 `MyMethod` *treba biti asinkrona metoda jer su sve Unit of Work metode takve.*

Nakon toga nižemo upite na `unitOfWork` koji smo prethodno kreirali i to u poretku u kojem želimo da se metode izvršavaju.

[21] Ninject je jedan od DI frameworka koji pomaže podijeliti aplikaciju na manje komponente te ju kasnije ponovno povezati kao fleksibilnu cjelinu.

Primjer u kojem dodajemo korisnika, a zatim mu pridružimo role:

```
// Unošenje korisnika u user tablicu
var userId = await unitOfWork.AddAsync(user);

// Višestruki unosi korisnikovih rola:
foreach (var userRole in userRoles)
{
    // Pridružimo UserId prije unosa
    userRole.UserId = userId.Value;

    // Unošenje role u userRole tablicu
    await unitOfWork.AddAsync(userRole);
}
```

Na kraju nam preostaje samo još pozvati `CommitAsync` metodu:

```
await unitOfWork.CommitAsync();
```

Svi upiti koji su dodani u Unit of Work će biti pozvani kada pozovemo `CommityAsync` metodu. Kako ih Unit of Work održava sve u transakcijskom *scopeu*, osigurati će nam to da ni jedan nedovršeni upit ne napravi promjene na podacima, odnosno ili će sve akcije biti uspješne ili neće biti odrađena niti jedna.

6. Praktični dio rada

Projekt koji je rađen u sklopu ovog rada je aplikacija za praćenje kvalitete u visokom obrazovanju. Preko nje se mogu bilježiti boravci djelatnika Odjela za matematiku na drugim sveučilištima, kao i boravci drugih djelatnika na Odjelu. Mogu se bilježiti i pregledavati odlasci studenata Odjela za matematiku na strana sveučilišta, kao i boravci stranih studenata na Odjelu. Profili tih profesora i studenata se mogu uređivati, a isto tako i popisi institucija i sveučilišta s kojima Odjel surađuje.

Cilj projekta je bilježiti i pratiti informacije o događanjima na studiju i izvan njega putem aplikacije te tako zamijeniti papir. Aplikaciju bi koristili uredi za unapređivanje i osiguravanje kvalitete.

6.1. Popis korištenih tehnologija

Alati:

- Microsoft Visual Studio Ultimate 2013
- Microsoft Visual Studio Code
- MySQL Workbench 6.3

Poslužitelj

- ASP.NET 4.5.1
- AutoMapper 4.2.1
- Entity Framework 6.1.3
- Microsoft ASP.NET Web API 5.2.3
- Microsoft Owin 3.0.1
- MySql ADO.Net 6.9.8
- Ninject 3.2.0

Klijent

- Angular 1.4.4
- Bootstrap 3.3.5

- Underscore.js 1.8.2
- ngDialog.js 0.3.12
- smart-table.js 2.1.8

6.2. Struktura aplikacije

Aplikacija je strukturirana na način da se za svaku od funkcionalnosti kreira servis u ASP.NET-u. Primjerice, za bilježenje i praćenje informacija o konferencijama, kreira se servis koji je za njih zadužen. Taj servis je zapravo klasa koja uz osnovne metode za dohvaćanje podataka, unos, brisanje i uređivanje, sadrži i metode specifične tome servisu. Pod tim specifičnim metodama smatramo neke metode koje obrađuju podatke, a pripadaju istoj klasi.

Svaki servis nam služi za određivanje koje objekte i koja njihova svojstva te koje podatke želimo poslati klijentskoj aplikaciji. Ako u podacima imamo lozinke ili druge povjerljive podatke, logično bi bilo da ih ne želimo prikazati svim korisnicima. Servisi nam služe i za određivanje što želimo spremati u bazu podataka. Nekada nije potrebno spremiti sve podatke koji stignu s klijentske aplikacije. Stoga je u servisu potrebno obraditi pristigle podatke te odlučiti koje od njih je potrebno spremiti a koje ne.

6.3. Primjer korištenja Unit of Work-a

U sljedećem primjeru je potrebno koristiti Unit of Work patern jer se sve naredbe trebaju izvršiti i to u određenom poretku:

Primjer 6.1 *Korisnik želi urediti popis institucija koje su bile suorganizatori stručnog skupa. Recimo da želi jednu instituciju ukloniti s popisa i dodati dvije nove. Primjer sa slike (4).*

Kako bismo osigurali izvršenje navedenoga scenarija iz primjera, radnje moramo izvršiti u sljedećem poretku:

1. Iz popisa referenci stručnih skupova sa institucijama, potrebno je obrisati referencu na instituciju koju više ne želimo u popisu.
2. Zatim je potrebno unijeti nove reference na dvije institucije koje nisu bile u popisu.
3. Na kraju je potrebno ažurirati ostale podatke stručnog skupa.



Slika 4: Korisnik odznačuje jednu i označuje dvije nove institucije s popisa.

Najprije ćemo dohvatiti postojeće podatke jer moramo znati koji su podaci bili u bazi podataka prije nego ih počnemo uređivati.

```
public async Task<int> UpdateAsync(IConference entity)
{
    var existing = await Repository.SingleAsync<conference>(entity.ID);

    ...
}
```

Metoda `.SingleAsync()` će nam za poslani identifikator vratiti jedinstveni objekt iz baze podataka koji je određen tim identifikatorom. Njega spremamo u privremeni objekt kojeg ćemo nazvati `existing`.

Kako bismo znali koji podaci su izmijenjeni moramo usporediti objekt koji smo poslali s objektom iz baze podataka. Zapravo ono što želimo usporediti su popisi referenci stručnih skupova sa institucijama, a to možemo učiniti na sljedeći način:

```
var collectionForInsert = entity.ConferenceInstitutions
        .Where(choice => existing.conferenceinstitutions
        .SingleOrDefault(c => c.ID == choice.ID) == null)
        .ToArray();
```

Ovdje smo usporedili `ConferenceInstitutions` objekt iz poslanog `entity` objekta s `conferenceinstitutions` objektom iz postojećih `existing` podataka te tako odredili koje reference moramo unijeti. Izraz `.SingleOrDefault` vrati jedan element koji odgovara zadanim uvjetima ili `null` ako takav element ne postoji[22]. Kako mi želimo elemente koji nisu u `existing` objektu s obzirom na `ID`, u kolekciju ćemo spremati samo one elemente koji nam vraćaju `null` vrijednost. `.Where` izraz će provjeriti sve elemente, pa će nam taj objekt sadržavati popis svih referenci institucija koje nisu u `existing` objektu. U privremeni `collectionForInsert` objekt spremamo tu kolekciju.

```
var collectionForUpdate = entity.ConferenceInstitutions
        .Where(choice => existing.conferenceinstitutions
        .SingleOrDefault(c => c.ID == choice.ID) != null)
        .ToArray();

var collectionForDelete = existing.conferenceinstitutions
        .Where(choice => entity.ConferenceInstitutions
        .SingleOrDefault(c => c.ID == choice.ID) == null)
        .Select(c => c.ID)
        .ToArray();
```

Slično kako smo odredili koje reference još nisu unešene, tako možemo pronaći i one koje već jesu u popisu. Ovaj put samo tražimo elemente za koje će `SingleOrDefault` izraz biti različit od `null` vrijednosti. Njih spremamo u `collectionForUpdate` privremeni objekt, a reference koje želimo obrisati u `collectionForDelete`. Kod traženja elemenata koje ćemo obrisati vidimo da smo samo zamijenili u kojoj kolekciji tražimo koje elemente, odnosno provjeravamo koji elementi su u `existing`, a nisu više u poslanoj `entity` kolekciji. Za brisanje podataka je dovoljno proslijediti identifikator, pa smo u privremeni objekt s kolekcijom referenci koje želimo obrisati spremili samo identifikatore tih referenci. To smo odredili s `.Select(c => c.ID)` izrazom.

Zatim kreiramo Unit of Work naredbom koju smo već ranije spominjali:

```
var unitOfWork = Repository.CreateUnitOfWork();
```

[22]U literaturi [16] možete pronaći sve detalje o izrazima `.SingleOrDefault`, `.FirstOrDefault` i sličnima.

Kad smo ga kreirali, možemo nizati naredbe u poretku kojem želimo da se izvršavaju.

```
await this.AddChoiceRangeForDeleteAsync(
    unitOfWork,
    collectionForDelete);
await this.AddChoiceRangeForUpdateAsync(
    unitOfWork,
    collectionForUpdate);
await this.AddChoiceRangeForInsertAsync(
    unitOfWork,
    entity.ID,
    collectionForInsert);
await this.AddForUpdateAsync(unitOfWork, entity);
```

Najprije kolekciju za brisanje proslijedimo u metodu `AddChoiceRangeForDeleteAsync`.

```
public async Task<int> AddChoiceRangeForDeleteAsync(
    IUnitOfWork unitOfWork,
    string[] ids)
{
    var result = 0;
    foreach (var id in ids)
    {
        result += await unitOfWork
            .DeleteAsync<conferenceinstitution>(id);
    }
    return result;
}
```

Vidimo da će nam se u toj metodi za svaki identifikator pokrenuti Unit of Work `DeleteAsync` metoda. Zatim kolekciju onih referenci koje već jesu u popisu, odnosno `collectionForUpdate`, proslijedimo u metodu `AddChoiceRangeForUpdateAsync`.

```
public async Task<int> AddChoiceRangeForUpdateAsync(
    IUnitOfWork unitOfWork,
    IConferenceInstitution[] entities)
{
    var result = 0;
```

```
    foreach (var entity in entities)
    {
        result += await unitOfWork.UpdateAsync<conferenceinstitution>(
            Mapper.Map<conferenceinstitution>(entity));
    }
    return result;
}
```

U toj metodi će nam se ažurirati svaki objekt iz popisa referenci. Naime, u ovom primjeru ta metoda nije potrebna jer se vjerojatno neće morati ažurirati ni jedan unos, ali ju je dobro imati u slučaju da se nekada proširi objekt s popisom referenci. Trenutno `conferenceinstitution` objekt sadrži samo vlastiti identifikator, identifikator konferencije i identifikator institucije. Kada bi sadržavao još neki dodatni atribut, on bi se vjerojatno mijenjao kod uređivanja i ta bi nam metoda bila neophodna.

Sljedeće što nam se proslijedi je kolekcija referenci koju moramo unijeti, odnosno `collectionForInsert` proslijedimo u metodu `AddChoiceRangeForInsertAsync`.

```
public async Task<int> AddChoiceRangeForInsertAsync(
    IUnitOfWork unitOfWork,
    string conferenceId,
    IConferenceInstitution[] entities)
{
    var result = 0;
    foreach (var entity in entities)
    {
        entity.ConferenceId = conferenceId;
        entity.ID = Guid.NewGuid().ToString("N");
        var map = Mapper.Map<conferenceinstitution>(entity);
        result += await unitOfWork.AddAsync<conferenceinstitution>(map);
    }
    return result;
}
```

Kako `collectionForInsert` objekt sadrži reference na institucije koje unosimo, moramo za svaki element `entity` iz popisa dodijeliti referencu na konferenciju. To odradimo s `entity.ConferenceId = conferenceId;`, gdje smo `conferenceId` proslijedili kao atribut. Zatim generiramo novi identifikator s `Guid.NewGuid().ToString("N");` i pozovemo Unit of Work `AddAsync` metodu kojoj pošaljemo taj objekt.

U posljednjoj naredbi još želimo ažurirati ostale podatke stručnog skupa, pa objekt koji je poslan s klijentske aplikacije pošaljemo u `AddForUpdateAsync` metodu.

```
public Task<int> AddForUpdateAsync(
    IUnitOfWork unitOfWork,
    IConference entity)
{
    var map = Mapper.Map<conference>(entity);
    map.conferenceinstitutions = null;
    return unitOfWork.UpdateAsync<conference>(map);
}
```

Ovdje vidimo da se neće odraditi ništa posebno osim što će se ukloniti `conference-institutions` objekt jer smo ga već ranije odredili preko prethodnih metoda i Unit of Work-a. Pa samo još proslijedimo obrađeni objekt na Unit of Work `UpdateAsync` metodu.

Na kraju nam preostaje samo još pozvati `CommitAsync` metodu kao i u primjeru (5.2) i vratiti rezultat tog poziva:

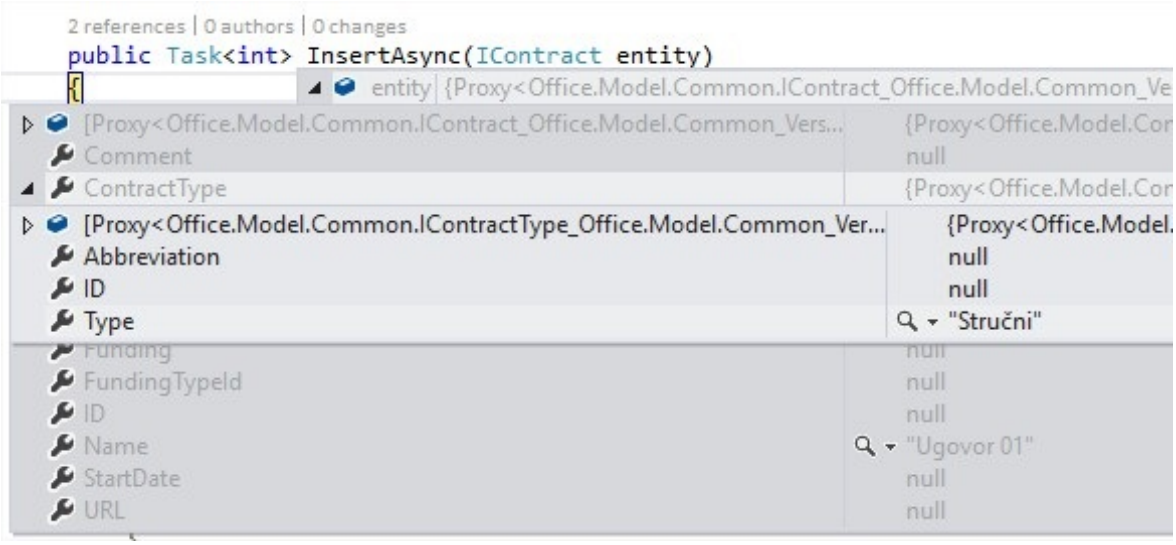
```
return await unitOfWork.CommitAsync();
```

Svi upiti koji su dodani u Unit of Work će i u ovom primjeru biti pozvani tek kada pozovemo `CommityAsync` metodu, pa smo si time osigurali poredak i izvršavanje svih metoda.

6.4. Primjer u kojem Unit of Work nije potreban

Primjer 6.2 *Korisnik želi unijeti novi ugovor i kao vrstu ugovora mu postaviti neku koja nije na popisu, odnosno ne postoji u bazi podataka.*

U ovom slučaju se može u kôdu podesiti da se ta nova vrsta ugovora unese zajedno sa svim podacima o ugovoru.



Slika 5: Vrsta ugovora

Na slici (5) možemo vidjeti kako se za vrstu ugovora `Type` poslala vrijednost "Stručni" te po vrijednosti atributa `ID` znamo da taj tip ne postoji u bazi podataka. Kada bi postojao i bio odabran preko klijentske aplikacije, poslao bi se identifikator postojećeg ugovora.

Stoga u kôdu trebamo postaviti provjeru je li poslan objekt s tipom ugovora, odnosno `entity.ContractType != null` te dodatno provjeriti je li `ID` iz tog objekta bez vrijednosti, odnosno `entity.ContractType.ID == null`. Sada je dovoljno generirati novi identifikator koji ćemo dodijeliti tom `ID`-u i spremiti ga kao referencu u `entity` objektu na `ContractTypeId` atribut.

```
public Task<int> InsertAsync(IContract entity)
{
    ...

    if (entity.ContractType != null
```

```
        && entity.ContractType.ID == null)
    {
        entity.ContractType.ID = Guid.NewGuid().ToString("N");
        entity.ContractTypeId = entity.ContractType.ID;
    }

    ...
}
```

Kada pozovemo `InsertAsync` metodu kojoj proslijedimo objekt poslan s klijentske aplikacije zajedno sa podobjektom `ContractType` te ako on ima potrebne vrijednosti, a to su u ovom slučaju `ID` i `Type`, u bazu će nam se spremiti oba objekta u istom pozivu.

Neće biti potrebno odraditi zasebne pozive u kojima bi najprije spremili `ContractType` objekt, a zatim `Contract` objekt. Tako da neće biti potrebno ni koristiti Unit of Work patern u ovom slučaju. Cijeli kôd takve metode bi izgledao ovako:

```
public Task<int> InsertAsync(IContract entity)
{
    entity.ID = Guid.NewGuid().ToString("N");

    if (entity.ContractType != null
        && entity.ContractType.ID == null)
    {
        entity.ContractType.ID = Guid.NewGuid().ToString("N");
        entity.ContractTypeId = entity.ContractType.ID;
    }

    return Repository.InsertAsync<contract>(
    Mapper.Map<contract>(entity));
}
```

Vidimo da nije potrebno baš uvijek koristiti Unit of Work patern jer su neke funkcionalnosti implementirane i u samom Entity frameworku. Bio bi to nepotreban dio kôda, pa treba razmisliti o tome što je već implementirano da se ne bi kompliciralo.

7. Zaključak

Razvojem Weba, izrada aplikacija se uvelike počela mijenjati. Ljudi su se sve više počeli okretati izradi i korištenju web aplikacija u odnosu na klasične desktop aplikacije. Razlog tome je taj što web aplikacije nije potrebno instalirati na svako računalo na kojemu ih želimo koristiti. Dovoljan je web preglednik i pristup internetu.

Aplikacije su postale modularnije, te je dio često pisan kao API kojeg mogu koristiti razne klijentske aplikacije. To znači da više ne moramo pisati kôd za poslužitelja svaki puta kada želimo dodati novu klijentsku aplikaciju. Pri tome je bitno dobro strukturirati kôd radi lakšeg održavanja i nadograđivanja aplikacije. Tu nam mogu pomoći softver dizajn paterni, a Unit of Work je jedan od njih.

Vidjeli smo da nam Unit of Work omogućuje veću kontrolu nad mijenjanjem podataka u bazi podataka. Pomoću njega se možemo osigurati da će se sve radnje, koje želimo odraditi na bazi, uspješno izvršiti, ili neće biti izvršena niti jedna. To nas štiti od nepotpunih podataka u bazi kao što je, na primjer, korisnik aplikacije koji nema svoju rolu.

Ipak, trebamo pripaziti da ne koristimo paterne onda kada nisu potrebni. Jedan takav primjer smo vidjeli u posljednjem poglavlju ovog rada.

Dakle, softver dizajn paterni, pa tako i Unit of Work, nam omogućuju bolje strukturiranje kôda i daju nam rješenja za česte probleme pri izradi softvera. No, uvijek treba dobro promisliti i vidjeti hoće li nam patern pomoći ili samo zakomplicirati kôd i njegovo održavanje.

Literatura

[1] B. Lakshmiraghavan, *Practical ASP.NET Web API*, 2013.

[2] Code Project - S. Koirala, *Unit of Work Design Pattern*, URL: https://www.codeproject.com/articles/581487/unit-of-work-design-pattern

[3] D. Lozić, dr. sc. A. Šimec, *Pametna komunikacija na Internetu preko REST protokola*, Tehničko veleučilište u Zagrebu, Zagreb

[4] E. DelBono, *ASP.NET Web API Succinctly*, Morrisville, 2013.

[5] Entity Framework Tutorial, *Entity Lifecycle*, URL: http://www.entityframeworktutorial.net/entity-lifecycle.aspx

[6] G. McLean Hall, *Adaptive Code via C# - Agile coding with design patterns and SOLID principles*, Washington, 2014.

[7] HitecHTube.com, *What is Client-Server Architecture?*, URL: http://hitechtube.blogspot.com/2014/12/what-is-client-server-architecture.html

[8] I. Ben-Gan, D. Sarka, A. Machanic, K. Farlee, *T-SQL Querying*, Washington, 2015.

[9] I. Sommerville, *Software engineering - 9th edition*, Boston, Massachusetts, 2011.

[10] LosTechies, *Survey of Entity Framework Unit of Work Patterns*, URL: https://lostechies.com/derekgreer/2015/11/01/survey-of-entity-framework-unit-of-work-patterns/

[11] MSDN, *Entity Framework Add and Attach and Entity States*, URL: https://msdn.microsoft.com/en-us/library/jj592676(v=vs.113).aspx

[12] R. Manger, *Baze podataka*, skripta, Zagreb, 2003.

[13] SlideShare, *Service approach for development Rest API in Symfony2*, URL: https://www.slideshare.net/savchenko1/symfony2-rest-api-59772368

[14] T. Ugurlu, A. Zeitler, A. Kheyrollahi, *Pro ASP.NET Web API: HTTP Web Services in ASP.NET*, New York, 2013.

[15] TechNet, *What is SNMP?*, URL: https://technet.microsoft.com/en-us/library/cc776379(v=ws.10).aspx

[16] Technical Overload, *LINQ Single vs SingleOrDefault vs First vs FirstOrDefault*, URL: http://www.technicaloverload.com/linq-single-vs-singleordefault-vs-first-vs-firstordefault

[17] Wikipedia, *.NET Framework*, URL: https://bs.wikipedia.org/wiki/.NET_Framework

[18] Wikipedia, *Simple Mail Transfer Protocol*, URL: https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol

[19] Znanje.org, *Klijent server koncepcija*, URL: http://www.znanje.org/abc/tutorials/internet_abc/01/005_client_server.htm

8. Životopis

Ante Ljubić rođen je 26.02.1989. u Vinkovcima. Osnovnu školu pohađao je u Osnovnoj školi Josipa Lovretića u Otoku. Nakon osnovne škole upisao je Prirodoslovno matematičku gimnaziju Matije Antuna Reljkovića u Vinkovcima. Tijekom osnovne i srednje škole sudjelovao je na općinskim i županijskim natjecanjima iz matematike, fizike i njemačkog jezika. 2007. godine upisao je Fakultet elektrotehnike strojarstva i brodogradnje u Splitu, smjer elektrotehnika. 2008. godine ispisao se s fakulteta i upisao preddiplomski studij matematike i informatike na Odjelu za matematiku Sveučilišta J. J. Strossmayera u Osijeku. 2014. godine diplomirao je na preddiplomskom studiju te stekao akademski naziv Sveučilišni prvostupnik (baccalaureus) matematike (univ.bacc.math). Tijekom studiranja položio je MTA (*Microsoft Technology Associate*) certifikat: *Database Administration Fundamentals* (2015.) te stekao iskustvo rada software developera u tvrtci Mono u Osijeku.