Primjena baza podataka u izgradnji mobilnih aplikacija

Hlevnjak, Tomislav

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Hlevnjak

PRIMJENA BAZA PODATAKA U IZGRADNJI MOBILNIH APLIKACIJA

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Hlevnjak

JMBAG: 0016136509

Studij: Informacijski sustavi

PRIMJENA BAZA PODATAKA U IZGRADNJI MOBILNIH APLIKACIJA

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Kornelije Rabuzin

Varaždin, rujan 2021.

Tomislav Hlevnjak

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema ovog završnog rada obrađuje ulogu i značaj baza podataka u izgradnji mobilnih aplikacija. Najprije se u teorijskom dijelu rada uspoređuju različite vrste baza podataka, konkretno relacijske (SQL) i ne-relacijske (NoSQL) baze podataka te koja je njihova uloga i kako se koriste u izgradnji mobilnih aplikacija. U samoj usporedbi tih baza podataka usredotočenost je na onim svojstvima koja su bitna u izgradnji mobilnih aplikacija i koja utječu na performanse. U praktičnom dijelu rada opisana je implementacija mobilne aplikacije koja koristi bazu podataka. Sustav za upravljanje bazom podataka korišten za implementaciju mobilne aplikacije je SQLite, aplikacija je implementirana u Xamarin platformi, a programski jezik korišten za pisanje koda je C#. Baze podataka imaju široku primjenu u mnogim aplikacijama, posebice mobilnim, a s obzirom na povećanje informacija i podataka u današnje vrijeme, i povećanjem broja mobilnih uređaja, gotovo je nezamislivo da se baze podataka ne koriste kod mobilnih aplikacija.

Ključne riječi: baze podataka, SQL, NoSQL, mobilna aplikacija

Sadržaj

Sadržaj iii
1. Uvod 1
2. Metode i tehnike rada 2
3. Relacijske i NoSQL baze podataka 3
3.1. Relacijske baze podataka 3
3.2. NoSQL baze podataka 5
3.3. SQL vs NoSQL 6
4. Uloga baza podataka u izgradnji mobilnih aplikacija 9
4.1. Relacijske mobilne baze podataka 9
4.2. NoSQL mobilne baze podataka 10
5. Implementacija mobilne aplikacije za rezervaciju stola 11
5.1. ERA model 11
5.2. Baza podataka u C# 12
5.3. CRUD funkcije 15
6. Zaključak 27
Popis literature 28
Popis slika 30
Popis tablica 31

1. Uvod

Pojava baza podataka i veliki napredak u tehnologiji otvorili su put razvojnim programerima da svoja programska rješenja, odnosno aplikacije, implementiraju na manje, lakše prenosive i praktičnije, mobilne uređaje umjesto na dotadašnja stolna računala. Samim time porasla je i traženost za programerima mobilnih aplikacija jer je mobilna industrija postala jedna od najvećih industrija u svijetu.

Gotovo sve mobilne aplikacije koje se u današnje vrijeme koriste imaju potrebu pohranjivati podatke, bilo da se radi o mrežnom (eng. *online*) ili izvanmrežnom (eng. *offline*) načinu rada uređaja. Kada govorimo o mrežnom načinu rada aplikacija, mrežne mobilne aplikacije obično ne spremaju podatke na lokalni uređaj, već su „uživo“ povezane s poslužiteljima na mreži da bi dohvaćale podatke.

S druge strane, izvanmrežne mobilne aplikacije najbolje odgovaraju korisnicima koji nemaju pristup mreži ili im je pristup ograničen, pa stoga podaci moraju biti spremljeni lokalno na uređaju da bi aplikacija mogla raditi. Često su takve aplikacije brže od onih koje trebaju internetsku vezu, pa se samim time nalaze izvorno na uređaju, a za pohranu podataka potrebnih za rad tih aplikacija zaslužne su baze podataka.

Upravo je primjena baza podataka kod mobilnih aplikacija tema ovog završnog rada, a motivacija za obradu ove teme jest izniman interes za baze podataka, ali i razvoj mobilnih aplikacija koje koriste baze podataka, u programskom jeziku C#.

Teorijski dio rada biti će usredotočen na opis i usporedbu relacijskih (SQL) i ne-relacijskih (NoSQL) baza podataka te koji je njihov značaj i uloga kod izgradnje mobilnih aplikacija, dok će u praktičnom dijelu rada biti implementirana mobilna aplikacija koja koristi bazu podataka. Tema aplikacije je rezervacija stola, jednostavna simulacija u kojoj se u bazu upisuju gosti, te se za određenog gosta stvara rezervacija stola. Sustav za upravljanje bazom podataka (SUBP) koji će biti korišten kod izrade aplikacije je SQLite, a razlog odabira tog sustava je prethodno znanje i iskustvo u radu zajedno s programskim jezikom C#.

2. Metode i tehnike rada

Rad je podijeljen na dva tematski povezana dijela. Prvi dio rada predstavlja teorijski pogled na temu. Metode korištene kod izrade teorijskog dijela su deskriptivna i komparativna metoda. Najprije će biti opisane, a zatim i uspoređene dvije vrste baza podataka, relacijske i NoSQL baze podataka, te će zatim biti specificiran njihov značaj kod izgradnje mobilnih aplikacija. Drugi dio rada predstavlja praktičnu izradu mobilne aplikacije koja koristi bazu podataka.

Za potrebe izrade praktičnog dijela rada korišteni su sljedeći alati: draw.io, Microsoft Visual Studio 2019 i Android Emulator. Draw.io je besplatan, *open-source* alat za stvaranje dijagrama tokova, procesa i organizacijskih dijagrama, UML i ERA modela, mrežnih dijagrama i dr. [1]. Vrlo lagan i intuitivan za korištenje, a poslužio je za izradu ERA modela za bazu podataka korištenu u praktičnom dijelu.

Microsoft Visual Studio 2019 jest integrirano razvojno okruženje (eng. *IDE*) koga pravi Microsoft. Koristi se za razvoj računarskih programa za Windows, web stranica, aplikacija i usluga [2].

Android Emulator simulira Android uređaje na računalu s namjenom da se izrađene aplikacije testiraju na raznim uređajima i verzijama Android sustava, pa stoga nema potrebe za posjedovanjem svakog fizičkog uređaja [3].

Sama mobilna aplikacija izrađena je na Xamarin platformi, besplatnoj *open-source* .NET platformi za izgradnju izvornih i visokoučinkovitih Android, iOS i dr. aplikacija u programskom jeziku C# [4].

Sustav za upravljanje bazom podataka (SUBP, eng. *Database Management System - DBMS*) korišten za izradu baze jest SQLite, relacijski sustav (RSUBP) temeljen na C programskoj biblioteci i najkorišteniji je SUBP na svijetu [5].

3. Relacijske i NoSQL baze podataka

Da bismo mogli govoriti o relacijskim, odnosno SQL bazama podataka i ne-relacijskim, odnosno NoSQL bazama podataka, najprije moramo definirati pojam baze podataka. Rabuzin [6] navodi da ima mnogo različitih definicija tog pojma, od skupa podataka do kolekcije povezanih slogova. Jednostavno rečeno, baza podataka je organizirani skup podataka.

Dakle, najčešća podjela baza podataka je na sljedeće dvije velike vrste ili kategorije: relacijske (SQL) baze i ne-relacijske (NoSQL) baze. U današnje vrijeme još uvijek prevladavaju relacijske baze podataka, iako se posljednjih godina NoSQL baze podataka sve više i više upotrebljavaju. Fatima [7] tvrdi kako organizacije i poduzeća neke od relacijskih i NoSQL baza podataka mogu koristiti pojedinačno, ili u kombinaciji, a to ovisi o prirodi podataka i potrebnoj funkcionalnosti.

Kada govorimo o relacijskim i NoSQL bazama podataka moramo spomenuti i termin „novi SQL“, tj. „*newSQL*“. *NewSQL* je termin koji označava novi pristup relacijskim bazama podataka koji želi kombinirati ACID svojstva (eng. *Atomicity*, C*onsistency*, *Isolation*, *Durability*) relacijskih sustava za upravljanje bazama podataka (RSUBP) i svojstvo horizontalne skalabilnosti NoSQL sustava [8]. Drugim riječima, *NewSQL* bi trebao omogućiti ono što NoSQL bazama još uvijek nedostaje, a to je konzistentnost u transakcijama. Više o svojstvima relacijskih i NoSQL baza podataka će biti dalje u tekstu.

3.1. Relacijske baze podataka

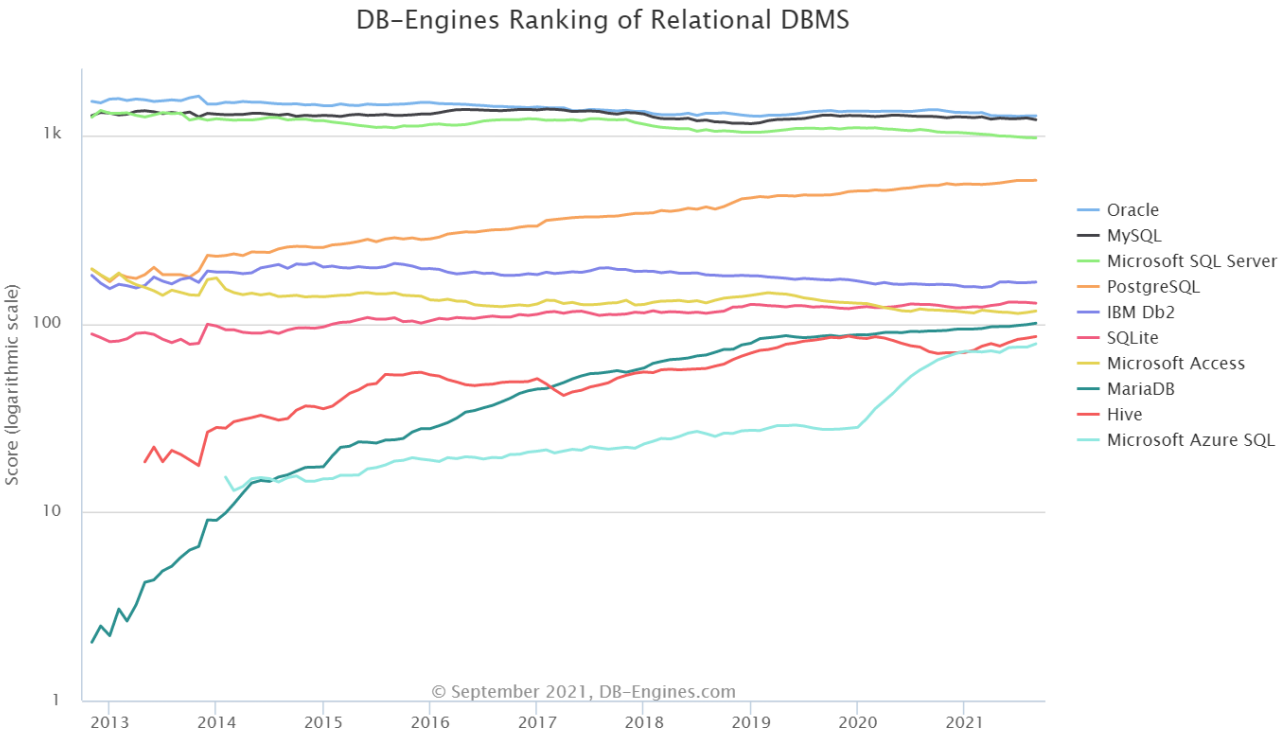
Relacijska, ili kraće SQL baza podataka vrsta je baza podataka koja je organizirana na temelju relacijskog modela podataka kojeg je predložio E. F. Codd još davne 1970. godine [9]. Prema tom modelu, podaci su organizirani u jednu ili više tablica (relacija) s redcima i stupcima, gdje svaki redak ima svoj jedinstveni ključ, poznatiji kao „primarni ključ“ (eng. *Primary Key*). Prema tome, svaki tip entiteta koji je opisan u bazi ima svoju tablicu s redcima i stupcima. Drugim riječima, redak predstavlja instancu te vrste entiteta, dok stupci predstavljaju vrijednosti pripisane toj instanci.

Budući da svaki redak u tablici ima svoj jedinstveni ključ, redci u jednoj tablici mogu se povezati s redcima iz druge tablice na način da se u drugoj tablici spremi taj jedinstven ključ s kojim bi trebali biti povezani. Taj jedinstven ključ onda je poznatiji kao „vanjski ključ“ (eng. *Foreign Key*) [9].

Gotovo svi sustavi za upravljanje bazama podataka koriste strukturirani upitni jezik, odnosno SQL (eng. *Structured Query Language*). Rabuzin [6] tvrdi da se unatoč prijevodu

„strukturirani *upitni* jezik“, SQL ne koristi samo da bi postavljali upite, već i da bi kreirali ili brisali objekte u bazi (tablice, poglede, indekse) i manipulirali podacima.

Na Slici 1. možemo vidjeti najbolje rangirane relacijske sustave za upravljanje bazom podataka.



Slika 1. Najbolje rangirani RSUBP, rujan 2021.
(Izvor: DB-Engines, https://db-engines.com/en/ranking/relational+dbms)

Već je ranije spomenuto kako su relacijske baze podataka i dan danas najkorištenija vrsta baza podataka u svijetu. Često se postavlja pitanje zašto su i nakon pojave NoSQL baza podataka relacijske baze još uvijek toliko popularne, kada NoSQL baze mogu baratati s većom količinom podataka velikom brzinom i u kratkom vremenu. Serra [9] tvrdi da su razlozi za dominacijom relacijskih baza sljedeća svojstva: jednostavnost, robusnost, fleksibilnost, performanse, skalabilnost i kompatibilnost u upravljanju generičkim podacima.

Prema Smallcombeu [10] neka od važnijih svojstava relacijskih baza podataka su predefinirana shema za definiranje i upravljanje podacima te vertikalna skalabilnost. S druge strane Anderson i Nicholson [11] tvrde kako su fleksibilni upiti i ACID svojstva vrlo važna jer garantiraju valjane transakcije i omogućuju podršku za različita radna opterećenja, u kojima bi se baza mogla naći.

3.2. NoSQL baze podataka

Zbog sve većeg povećanja prijenosa podataka putem Interneta, više podataka o proizvodima, događajima pa čak i o samim korisnicima i sl., sve te podatke treba pohraniti i njima rukovoditi, a klasična relacijska baza podataka ne može više ispuniti sve te zahtjeve. Iz tog razloga počele su se razvijati ne-relacijske baze podataka, poznatije pod imenom NoSQL, gdje NoSQL znači „*not only SQL*“ ili „*non SQL*“.

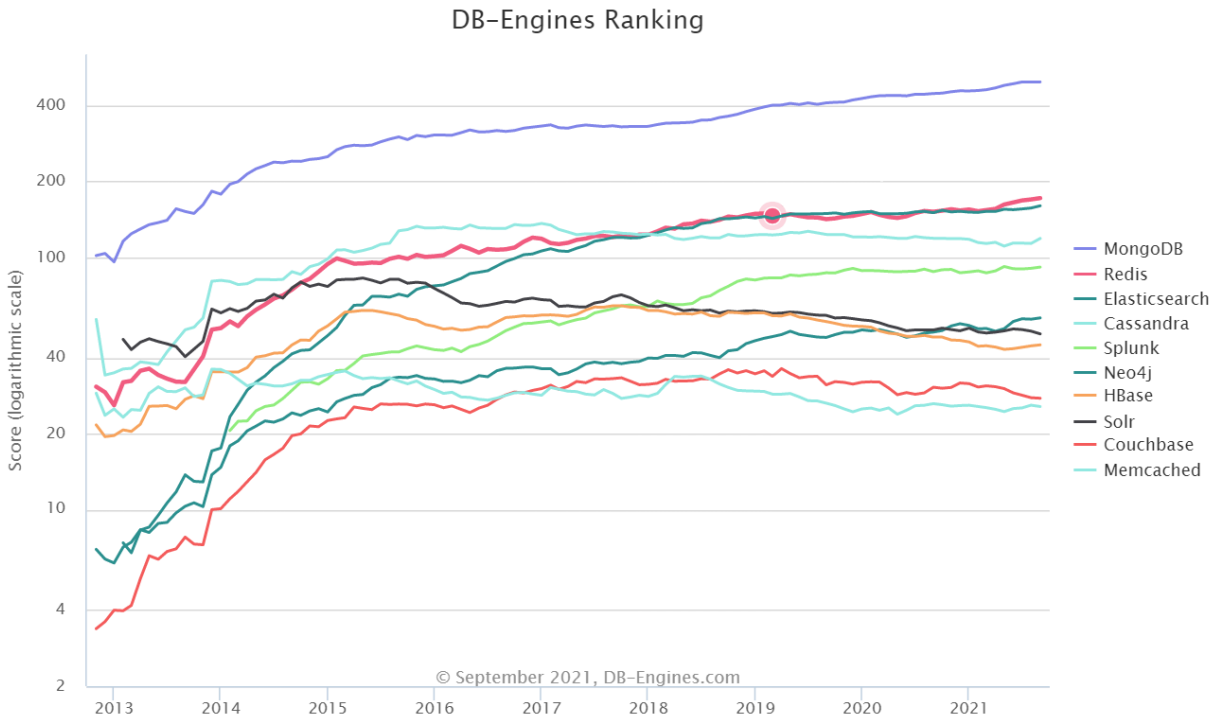
Za razliku od relacijskih baza koje su temeljene na tablicama, NoSQL baze mogu biti temeljene na dokumentima, parovima oblika ključ-vrijednost, stupcima i grafovima. Često se u NoSQL baze podataka ubrajaju i tražilice.

Foote [12] je četiri kategorije NoSQL baza objasnio na sljedeći način. Ključ-vrijednost baze podataka su sustavi dizajnirani za pohranu, dohvaćanje i upravljanje „asocijativnim nizovima“. Dokument baze podataka su sustavi dizajnirani za pohranu, dohvaćanje i upravljanje „dokumentno-orijentiranim informacijama“, drugim imenom polu-strukturiranim podacima. Stupac baze podataka koristi tablicu, redove i stupce, ali za razliku od relacijskih baza, imena i formati stupaca mogu se mijenjati iz reda u red unutar iste tablice. Na taj su način imena i formati fleksibilniji. Grafovska baze podataka u osnovi je skup odnosa, pa tako svaki čvor simbolizira entitet (posao, osobu ili objekt), čvorovi su povezani s nekim drugim čvorom, a veza između njih naziva se „rub“ (brid) i predstavlja odnos između čvorova.

U odnosu na relacijske baze podataka, Smallcombe [10] tvrdi kako NoSQL baze podataka imaju dinamičku, tj. fleksibilnu shemu za nestrukturirane podatke, te horizontalnu skalabilnost. Da bi se radilo s većom količinom podataka nije potrebno pojačavati snagu jednog poslužitelja, već dodati više poslužitelja za određenu bazu. Moglo bi se reći da bolje je iskoristiti više starijih i jeftinijih računala, umjesto jednog vrlo jakog i skupog da bi NoSQL baza uspješno funkcionirala, ukoliko bi se količina podataka povećala.

Osim dinamičke sheme i horizontalne skalabilnosti, brzi upiti navode se kao prednost NoSQL baza podataka. Podaci u relacijskim bazama su često normalizirani, pa je za upite potrebno spajati (eng. *join*) tablice da bi se dohvatili podaci o entitetu, a ukoliko se radi s velikim tablicama gubi se na brzini izvođenja upita. S druge strane, u NoSQL bazama podaci su spremljeni tako da su već optimizirani za upite. Dobar primjer takve prakse je MongoDB, jedan od najpoznatijih SUBP na svijetu [13]. Međutim, iako NoSQL baze izvode upite brže, ipak nisu poželjne ako trebamo raditi s kompleksnim upitima.

Na Slici 2. možemo vidjeti najbolje rangirane ne-relacijske sustave za upravljanje bazom podataka.



Slika 2. Najbolje rangirani NoSQL SUBP, rujan 2021.
(Izvor: DB-Engines, https://db-engines.com/en/ranking)

3.3. SQL vs NoSQL

Često puta se postavlja pitanje odabira relacijske ili NoSQL baza podataka. Prilikom odabira vrlo je važno razmišljati o podacima: kako izgledaju, koliko će ih biti, koje su vrste, kako će se filtrirati, grupirati i sl. Dakle, podaci su prvi čimbenik koji je važan u odabiru baze podataka.

Chen [14] tvrdi kako je relacijska baza podataka najbolji odabir ukoliko su podaci strukturirani. Primjerice, za sustave koji su transakcijsko-orijentirani poput računovodstva ili platformi e-trgovina i sl., relacijska baza ima veliku prednost jer bi svaki redak u tablici bio različit, npr. kupac, a svaki stupac neki atribut koji opisuje tog kupca. I upravo zbog tih različitih, strukturiranih veza između stupaca i redaka, SQL je najbolji odabir. S druge strane, ako su podaci s kojima želimo raditi nestrukturirani, bolji odabir je NoSQL. Kako ne postoji predefinirana shema, podaci mogu biti smješteni u ranije spomenute ključ-vrijednost, dokumentno-orijentirane, stupčaste ili grafovske baze podataka. To omogućuje manje prethodnog planiranja, odnosno više fleksibilnosti prilikom upravljanja bazom podataka.

Sljedeći važan čimbenik na kojeg treba obratiti pažnju prilikom odabira baze podataka jest skaliranje. Kako se relacijske baze skaliraju vertikalno, a NoSQL baze horizontalno, vrlo je važno razmišljati kako i koliko će podaci rasti u budućnosti.

Budući da su relacijske baze dizajnirane da rade na jednom poslužitelju, nije ih lako skalirati da bi se sačuvao integritet podataka. To znači da bi se povećanjem podataka trebale pojačati i performanse poslužitelja, prvotno procesorska snaga, količina radne memorije i veličina vanjske memorije (HDD, SSD). Za razliku od strukturiranih relacijskih baza, svaki objekt (entitet), pohranjen u NoSQL bazi je poprilično samostalan i neovisan, pa se tako objekti mogu vrlo lako pohraniti na više poslužitelja. Samim time poveća se i (horizontalna) skalabilnost baze i to je jako velika prednost u odnosu na relacijske baze podataka [14].

Vrlo važno svojstvo i funkcionalnost baze čine transakcije. Već je ranije spomenuto kako relacijske baze slijede tzv. ACID teorem kada se radi o transakcijama. Atomnost, konzistentnost, izolacija i trajnost svojstva su transakcija u relacijskim bazama podataka. Atomnost je svojstvo koje osigurava da su svi podaci u bazi nužno potvrđeni. Drugim riječima, ako transakcija nije pravilno izvedena proces se mora vratiti u početno stanje. Svojstvo konzistentnosti (dosljednosti) osigurava da izvršena transakcija ne naruši strukturni integritet baze. Izolacija osigurava da je svaka transakcija izolirana od drugih transakcija, prema tome ne može ugroziti integritet neke druge transakcije. Trajnost predstavlja svojstvo koje osigurava da podaci koji se odnose na određenu transakciju ne smiju izgubiti ukoliko transakcija ne uspije zbog eventualnog pada sustava ili drugog problema [15].

S druge strane NoSQL baze podataka ne koriste ACID teorem, već BASE teorem (eng. *Basically Available*, *Soft State*, *Eventually Consistent*). Dakle, uobičajena dostupnost je svojstvo koje osigurava stalnu dostupnost podacima u bazi. Promjena stanja je svojstvo koje omogućuje da se sustav stalno mijenja, tj. da baza može biti ili konzistentna ili nekonzistentna što je ujedno i treće svojstvo, svojstvo eventualne konzistentnosti. Drugim riječima, u nekom trenutku baza može biti nekonzistentna te se nakon nekog vremena može vratiti u konzistentno stanje [15].

Zaključak je da su ACID svojstva, odnosno relacijske baze bolji izbor za poduzeća i organizacije kojima je cilj konzistentnost, predvidljivost i pouzdanost u transakcijama, dok su BASE svojstva, odnosno NoSQL baze bolji izbor za poduzeća i organizacije koje na prvo mjesto stavljaju visoku dostupnost, skalabilnost i fleksibilnost u transakcijama podataka.

Pogledajmo u Tablici 1. rezime navedenih razlika između relacijskih i NoSQL baza podataka.

	Relacijske baze podataka	**NoSQL baze podataka**
Struktura (model) podataka	Relacijska tablica	Ovisno o bazi: dokument, ključ-vrijednost, stupac, graf, tražilica
Shema	Predefinirana shema	Dinamička shema
Skalabilnost	Vertikalna skalabilnost	Horizontalna skalabilnost
Upiti	SQL jezik, jednostavni i složeni upiti	Ne postoji univerzalni programski jezik, brzi i uglavnom jednostavni upiti
Transakcije	ACID svojstva	BASE svojstva
Prioriteti	Integritet, konzistentnost i stabilnost podataka	Fleksibilnost i skalabilnost podataka, brzi upiti

Tablica 1. Usporedna tablica relacijskih i NoSQL baza (vlastita izrada)

4. Uloga baza podataka u izgradnji mobilnih aplikacija

Objasnili smo koje su glavne značajke relacijskih i NoSQL baza podataka te smo ih usporedili, a sada ćemo objasniti njihov značaj i ulogu u izgradnji mobilnih aplikacija. Kako je svijet mobilnih uređaja stalno u pokretu, potrebno je držati korak za najnovijim aplikacijama i željama korisnika. Da bi u novije vrijeme razvili mobilnu aplikaciju potrebno je odabrati bazu podataka koja je brza, skalabilna i sigurna. Kao i kod svih ostalih aplikacija koje koriste baze podataka, i kod mobilnih je aplikacija česta rasprava treba li koristiti relacijsku ili NoSQL bazu podataka.

Baze podataka za mobilne uređaje, ili kraće mobilne baze podataka su baze koje se nalaze na PDA uređajima (eng. *Personal Digital Assistant*), pametnim telefonima (eng. *smartphone*), tabletima i ponekad u prijenosnim računalima. Uglavnom, koriste se na uređajima s ograničenom memorijom, procesorskom brzinom i snagom baterije [16].

Budući da se nalaze na uređajima s ograničenim resursima, mobilne baze podataka često su mnogo manje od baza podataka koje su na poslužiteljima. Često puta su ugrađene (eng. *embedded*) u neku mobilnu aplikaciju, dakle nemaju poslužitelja i uglavnom je to jedna mala podatkovna datoteka. Iako se u današnje vrijeme mobilne aplikacije sve više oslanjaju na vanjske poslužitelje, većina zadanih (eng. *native*) aplikacija koriste ugrađenu bazu kao spremište podataka. Primjer takve ugradive baze podataka koja podatke sprema lokalno jest SQLite.

4.1. Relacijske mobilne baze podataka

Kada govorimo o relacijskim mobilnim bazama podataka prva asocijacija je uvijek SQLite. SQLite sustav za upravljanje bazama podataka je doslovce najstariji mobilni sustav za baze podataka, neki čak govore i jedini „uspostavljeni“ sustav za mobilne baza podataka [17]. Razlog tome je prisutnost SQLite-a od samih početaka iOS i Android uređaja.

SQLite sustav radi bazu podataka koja sadrži samo jednu datoteku na disku, pa je stoga lako prenosiva. Kako neke organizacije ili poduzeća koriste više od jedne baze važno je da baza bude prenosiva [18]. Osim toga, SQLite može poslužiti za izradu aplikacija za više platformi (eng. *cross-platform*), primjerice za React Native, Javu ili kao što će biti u ovom završnom radu, Xamarin platformu. Panchal [18] tvrdi kako razvojni programeri često puta imaju problema s testiranjem aplikacije kada je baza komplicirana, pa je SQLite dobar odabir jer je dobar za testiranje.

Glavni nedostatak SQLite je to da svaki korisnik može čitati/pisati u bazu bez ikakve specijalne dozvole zato jer u SQLite-u ne postoji upravljanje korisnicima (eng. *user management*). Mogući su napadi SQL injekcijama, a time pada sigurnost baze podataka.

SQLite nije jedini SUBP koji se koristi u izgradnji mobilnih podataka, tu su još i MySQL, SQL Server, PostgreSQL i MariaDB, no njihove implementacije često nisu besplatne.

4.2. NoSQL mobilne baze podataka

Jako je teško zamijeniti SQLite kada se nalazi u gotovo svim mobilnim uređajima u svijetu. Međutim, NoSQL baze pronašle su svoj put i do mobilnih uređaja. Današnje aplikacije uglavnom su implementirane u objektno-orijentiranim jezicima, a to znači da aplikacija mora raditi s objektima. Relacijske baze rade sa stupcima i redcima nije moguće izravno pohranjivati podatke u objekt, već je potrebno mapiranje pomoću ORM-a. NoSQL baza koja je objektno-orijentirana neće trebati kompleksna mapiranja da bi se entiteti iz baze „preslikali“ u objekte u programskom jeziku.

Nakon objavljivanja mobilne aplikacije, često je u tijeku proces rada uživo za stalno prilagođavanje i poboljšanje aplikacije (ispravci programskih pogrešaka, nove značajke, promjene zbog promjena u softveru ili pravnom okruženju). Provedba promjena u mobilnoj aplikaciji s relacijskom bazom podataka klijenta zahtijeva promjene u shemi baze podataka, što rezultira dodatnim radom. NoSQL baze podataka obično rade bez sheme (obično s nekim provedbenim pravilima provjere valjanosti podataka). Dodavanje novih polja prema potrebi i spremanje različitih podataka zajedno prema potrebi znači minimalan napor za implementaciju baze podataka. Međutim, opet sve ovisi o specifičnoj implementaciji baze podataka i slučaju uporabe [17].

Kako postoji potreba za rukovanjem sa sve više podataka (grubo rečeno, količina podataka se udvostručuje svaka 24 mjeseca), očigledno je da je prostor za pohranu na mobilnim uređajima još uvijek velika briga, koja obično izostavlja jednostavno pohranjivanje svake moguće podatkovne točke za potencijalnu uporabu. Ipak, većina NoSQL baza podataka favorizira brzinu i pohranjuje nestrukturirane podatke. Dakle, za mobilne aplikacije koje rukuju velikom količinom (nestrukturiranih) podataka, NoSQL pristup može biti puno bolje rješenje od SQL-a [17]. Drugim riječima, stručnjaci smatraju da je bolje koristiti NoSQL baze podataka upravo zbog sve većeg povećanja podataka i informacija, koje zahtijevaju od korisnika stalne promjene i ažuriranja, što relacijske baze zbog svoje predefinirane sheme i strukturiranog načina zapisa podataka ne mogu ispuniti. Međutim, kada je riječ o malim aplikacijama, koje bi radile i mrežno i izvanmrežno, dobra je praksa koristiti relacijske baze.

5. Implementacija mobilne aplikacije za rezervaciju stola

Nakon što je objašnjena uloga baza podataka u izgradnji mobilnih podataka na red je stigla i izrada jednostavne mobilne aplikacije koja koristi bazu podataka. Scenarij koji je zamišljen i koji je motivacija za izradu ove aplikacije opisan je u sljedećim koracima.

Neki restoran vodi evidenciju svih gostiju koji se javljaju za rezervaciju stola. Administrator koji vodi bazu podataka najprije popisuje goste koji se jave, a zatim i na temelju njihovih zahtjeva izrađuje rezervacije koje bi se zatim eventualno slale voditelju restorana ili nekoj drugoj osobi koja je zadužena za taj dio. Administrator može u bilo kojem trenutku urediti podatke o gostu, ali i obrisati gosta s popisa svih gostiju. Nakon što se odabere gost s popisa gostiju, može se odabrati stol za rezervaciju, ali samo onaj stol koji nije već prethodno zauzet. Zatim se uz odabir stola biraju datum i vrijeme rezervacije, kao dodatni atributi koji bi spriječili gužvu u restoranu. Administrator baze podataka osim popisa gostiju vidi i popis svih rezervacija koje su napravljene, te popis svih stolova koji su u restoranu. Samim time, on može uređivati rezervacije na temelju zahtjeva korisnika i uređivati stolove u smislu popisivanja kapaciteta stola i zauzetosti. U slučaju da gost odustane od rezervacije, može se obrisati s popisa rezervacija ili čak i s popisa gostiju.

Sljedeće što je potrebno napraviti je identificirati glavne podatke, a to možemo postići izradom ERA modela.

5.1. ERA model

Na Slici 3. možemo vidjeti ERA model baze koja je napravljena prema ranije opisanom scenariju. ERA model (eng. *Entity, Relationship, Attributes*), odnosno ERA dijagram je vrsta dijagrama koja omogućuje da se prikažu korisnički zahtjevi, tj. definira struktura podataka [19].

Slika 3. ERA model baze za aplikaciju (vlastita izrada)

Poslovno pravilo po kojem je napravljen ovaj ERA model glasi: gost može rezervirati više stolova, a svaki stol može biti rezerviran od strane više gostiju. Ono što je važno napomenuti jest da se drugi dio odnosi na rezerviranje u drugom vremenskom razdoblju, odnosno kada stol nije zauzet. Prema tome, u bazi postoje dva ključna entiteta, odnosno dva tipa entiteta, a to su *Gost* i *Stol*. Svojstva koja nas zanimaju za *Gosta* i *Stol* su ujedno i svojstva koja ćemo zapisati u bazu, a ta su svojstva zapravo atributi entiteta. Tablica *Gost* sadrži sljedeće atribute: *ID_gost, Ime, Prezime, Adresa* i *Telefon*. Tablica *Stol* sadrži sljedeće atribute: *ID_stol*, *Kapacitet* i *Zauzet*.

Osvrtom na poslovno pravilo možemo primijetiti kako se između glavnih entiteta nalazi tip binarne veze više naprama više. To je veza koja se ne može izravno implementirati već je potrebno u model dodati novu tablicu čime dijelimo jednu vezu više naprama više na dvije veze jedan naprama više. Za veze jedan naprama više znamo da se mogu implementirati pomoću vanjskih ključeva, pa pogledom na ERA model vidimo kako je primarni ključ u novoj tablici složen, a sastoji se od vanjskih ključeva. Dakle, nova tablica *Rezervacija* sastoji se od atributa: *ID_gost*, *ID_stol*, *Datum_rezervacije*, *Vrijeme_rezervacije*. Datum i vrijeme rezervacije dodatni su atributi koji omogućuju da jedan gost rezervira isti stol, ali neki drugi dan.

Sada kada smo završili s modeliranjem baze podataka možemo krenuti s pisanjem programskog kôda za aplikaciju. Prvo što je potrebno napraviti su klase koje će predstavljati tablice iz baze podataka. Tehnika kojom se to radi zove se objektno-relacijsko mapiranje (eng. *Object-relational mapping, ORM*), gdje su objekti dio u programskim jezicima, relacije dio u tablicama, a mapiranja poveznice između objekata i tablica. ORM koji je korišten za izradu aplikacije jest *SQLite.NET* biblioteka.

5.2. Baza podataka u C#

U uvodu je rečeno kako će mobilna aplikacija biti izrađena na Xamarin platformi, pa stoga nakon ulaska u Visual Studio 2019 otvaramo novi *Mobile App (Xamarin.Forms)* projekt i možemo krenuti s izradom prve klase na temelju prve tablice, tablice *Gost*.

Klase *Gost*, *Stol* i *Rezervacija* smjestit ćemo u mapu *Models* te ih uključiti u svakoj drugoj klasi koju ćemo koristiti. Kao što je bilo specificirano ERA modelom, tablica *Gost* sadržavala je atribute *ID_gost*, *Ime*, *Prezime*, *Adresu* i *Telefon*. Sukladno tome, klasa *Gost* imat će ista svojstva, kao što možemo vidjeti u sljedećem kôdu:

```
public class Gost
    {
        [PrimaryKey, AutoIncrement]
        public int ID_gost { get; set; }
```

```
        public string Ime { get; set; }
        public string Prezime { get; set; }
        public string Adresa { get; set; }
        [MaxLength(10)]
        public string Telefon { get; set; }
        [ManyToMany(typeof(Rezervacija))]
        public List<Stol> Stolovi { get; set; }
        public string ImePrezime
        {
            get { return Ime + " " + Prezime; }
        }
        public string AdresaTelefon
        {
            get { return Adresa + ", " + Telefon; }
        }
    }
```

Možemo primijetiti kako se iznad prvog atributa nalaze ključne riječi *PrimaryKey* i *AutoIncrement*, što označava da će *ID_gost* biti primarni ključ koji će se automatski povećavati za 1 prilikom svakog unosa u bazu. Također, osim svojstava naznačeno je da se radi o više naprama više vezi između *Gosta* i *Stola*, tako da se dodala ključna riječ *ManyToMany* iznad liste koja je tipa *Stol*, a razlog toga je implementacija vanjskog ključa u klasi *Rezervacija* koji će se referencirati na ovu tablicu, tj. klasu. Uz glavne atribute u klasi *Gost* još se nalaze i dvije metode koje služe za ljepši ispis podataka na ekran uređaja.

Klasa *Stol* implementirana je na gotovo isti način, jedina razlika je što je sada ključna riječ *ManyToMany* dodana iznad liste koja je tipa *Gost*, da bi se referencirala klasa *Rezervacija*. Također, kao i kod klase *Gost*, postoje metode nevezane uz bazu podataka, a koje imaju dizajnersku svrhu.

```
public class Stol
    {
        [PrimaryKey, AutoIncrement]
        public int ID_stol { get; set; }
        public int Kapacitet { get; set; }
        public bool Zauzet { get; set; }
        [ManyToMany(typeof(Rezervacija))]
        public List<Gost> Gosti { get; set; }
        public string StolBr
        {
            get { return "Stol " + ID_stol; }
        }
```

```
        public string KapacitetZauzetost
        {
            get
            {
                if (Zauzet == true)
                {
                     return "Br. osoba: " + Kapacitet + ", ZAUZET";
                }
                else return "Br. osoba: " + Kapacitet + ", SLOBODAN";
            }
        }
        public string Zauzetost
        {
            get
            {
                if (Zauzet == true)
                {
                     return "ZAUZET";
                }
                else return "SLOBODAN";
            }
        }
    }
```

Klasa *Rezervacija* zanimljiva je jer se u njoj zapravo referenciraju vanjski ključevi na glavne tablice. Atribut *ID_gost* vanjski je ključ tipa *Gost*, a atribut *ID_stol* vanjski je ključ tipa *Stol*. Ta dva vanjska ključa čine jedan složeni primarni ključ koji je dio tablice *Rezervacija*. Uz ta dva glavna atributa, klasa još sadrži svojstva *Datum_rezervacije* i *Vrijeme_rezervacije* te metodu *DatumVrijeme* koja također kao i metode u prijašnjim klasama, služi za ljepši ispis na ekran.

```
public class Rezervacija
    {
        [PrimaryKey, ForeignKey(typeof(Gost))]
        public int ID_gost { get; set; }
        [PrimaryKey, ForeignKey(typeof(Stol))]
        public int ID_stol { get; set; }

        public DateTime Datum_rezervacije { get; set; }
        public TimeSpan Vrijeme_rezervacije { get; set; }
        public string DatumVrijeme
```

```
        {
            get { return Datum_rezervacije.ToShortDateString() + ", " +
Vrijeme_rezervacije; }
        }
    }
```

Atributi *ForeignKey* i *ManyToMany* nisu standardni dio *SQLite.NET* biblioteke, već su dio *SQLiteNetExtensions* biblioteke, pa je bilo potrebno uključiti i tu biblioteku kako bi se radilo s vanjskim ključevima. Vanjski ključevi važni su kod održavanja referencijalnog integriteta, primjerice, brisanje „roditelja“ neće se izvršiti ako se vrijednost „roditelja“ koristi u tablici „dijete“. Za ovu aplikaciju važna je opcija *Cascade*, koja omogućuje da se nakon brisanja „roditelja“ obrišu i sva „djeca“ tog „roditelja“ [6]. Međutim kako ta opcija nije dostupna, bilo je potrebno napisati upite koji će odraditi taj dio posla. U slučaju ove mobilne aplikacije nakon što se obrišu *Gost* ili *Stol*, obrisat će se i njegova *Rezervacija*.

Gotov je dio vezan uz stvaranje potrebnih modela, odnosno klasa za rad s aplikacijom. Sljedeći važan korak je kreiranje baze i spajanje na bazu, a zatim stvaranje CRUD (eng. *Create, Read, Update, Delete*) funkcija.

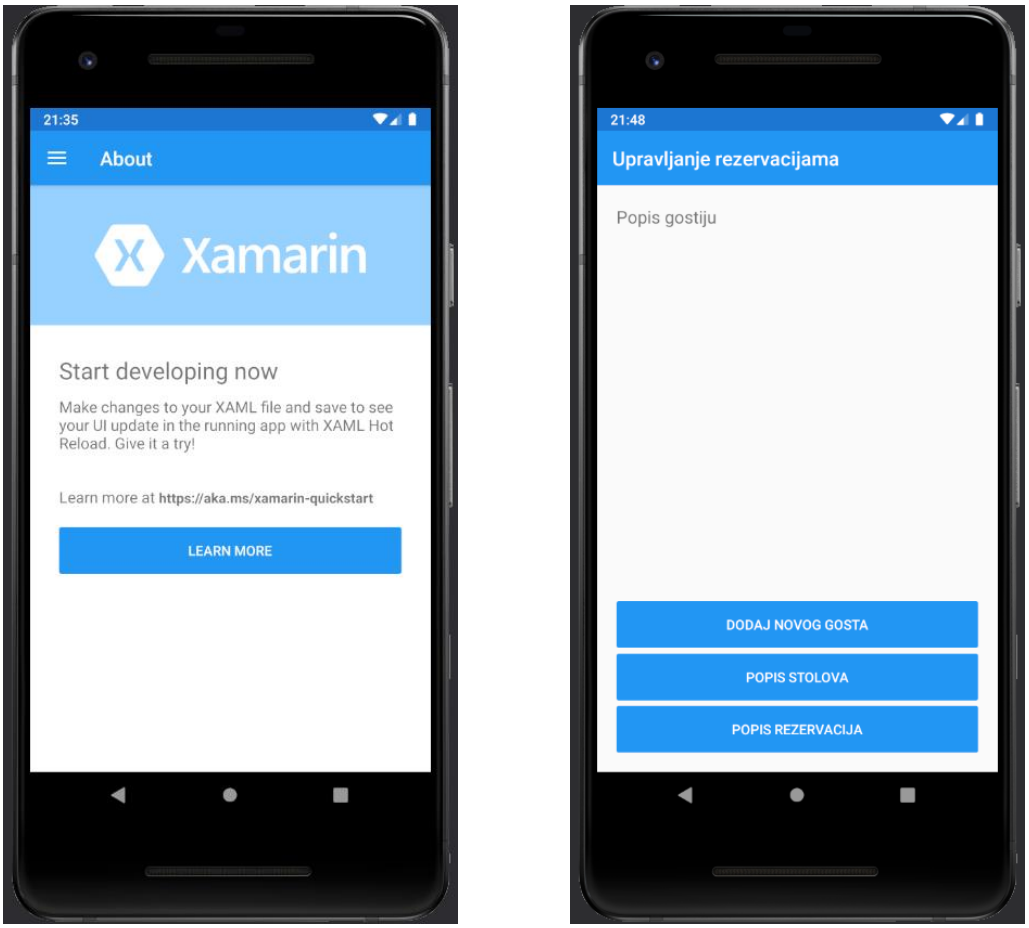
5.3. CRUD funkcije

Po zadanim postavkama Xamarin formi u projektu se nalazi klasa *MainActivity* koja u sebi sadrži *OnCreate()* metodu. Ta metoda je zapravo pokretač cijele aplikacije, pa su iz tog razloga u tu metodu dodane sljedeće 3 važne linije za stvaranje baze podataka:

```
string baza = "MojaBaza.db3";
string putanja = System.Environment.GetFolderPath(System.Environment.
SpecialFolder.LocalApplicationData);
string putanjaDoBaze = Path.Combine(putanja, baza);
```

Tim linijama specificira se ime baze, direktorij na koji će se spremiti i putanja do same baze. Prema tome, ime baze je *MojaBaza.db3* i bit će smještena u *LocalApplicationData* direktoriju. Putanja do baze je vrlo važna jer se ta vrijednost šalje kao argument konstruktoru za pokretanje aplikacije, u metodi *LoadApplication(new App(putanjaDoBaze))*. Time smo konstruktoru klase *App* poslali putanju na kojoj baza treba biti i sada aplikacija stvara bazu na toj putanji, spremnu za stvaranje tablica i unos podataka.

Nakon uspješnog stvaranja baze potrebno je napraviti tablice, te forme za unos, čitanje, ažuriranje i brisanje podataka. No, postoji jedan korak prije samih izrada tablica, a to je izrada sadržajnih stranica (eng. *ContentPage*) za aplikaciju. Na Slici 4. lijevo vidimo što bi se dogodilo ukoliko bi korisnik bez ikakvih promjena pokušao pokrenuti aplikaciju u Android Emulatoru.



Slika 4. Stranica *MainPage* i *Pocetna* (vlastita izrada)

Korisnik pokretanjem Emulatora vidi zadanu Xamarin stranicu pod nazivom *MainPage*, te ima dvije opcije: preuređivati postojeću zadanu ili napraviti novu sadržajnu stranicu. U svrhu rada je napravljena nova stranica, *Pocetna*, te je proslijeđena konstruktoru za pokretanje aplikacije umjesto *MainPage* stranice. Novu početnu stranicu možemo vidjeti desno na Slici 4. Budući da je *Pocetna* nova početna stranica, prigodno je da se njenim pokretanjem stvore tablice koje tražimo. U metodi *OnAppearing()* dodane su linije za stvaranje naših triju tablica:

```
protected override void OnAppearing()
        {
            base.OnAppearing();
            using(SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                veza.CreateTable<Gost>();
                veza.CreateTable<Stol>();
                veza.CreateTable<Rezervacija>();
            }
        }
```

Nakon što su tablice kreirane, možemo krenuti s prvom CRUD operacijom, a to je kreiranje, odnosno umetanje (eng. *Insert*) u bazu. Za tu svrhu napravljena je nova sadržajna stranica u obliku forme na kojoj administrator popunjava tražene podatke o gostu. Pogledajmo kôd za stranicu *DodajGosta*.

```
public partial class DodajGosta : ContentPage
    {
        public DodajGosta()
        {
            InitializeComponent();
        }
        private void dodajButton_Clicked(object sender, EventArgs e)
        {
            Gost gost = new Gost()
            {
                Ime = imeEntry.Text,
                Prezime = prezimeEntry.Text,
                Adresa = adresaEntry.Text,
                Telefon = telefonEntry.Text
            };
            using(SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                veza.CreateTable<Gost>();
                int brRedaka = veza.Insert(gost);
            }
            Navigation.PopAsync();
        }
    }
```

Kao što možemo vidjeti, stvaramo novu instancu klase *Gost*, sa svim atributima koji su u bazi, osim primarnog ključa. U takozvane unose (eng. *Entry*) upisuju se podaci u aplikaciju, zatim se pomoću izraza *using* spajamo na bazu i konačno pomoću *Insert*-a upisujemo te podatke u bazu. To je vrlo jednostavan način za upisivanje u bazu, a da bi provjerili je li unos ispravan možemo ispitati vrijednost varijable *brRedaka*, koja je veća od 0 u slučaju ispravnog unosa.

Međutim, nakon unosa u bazu u aplikaciji ne vidimo ništa. To je zato jer još nismo nigdje čitali ono što smo upisali, pa je vrijeme da implementiramo drugu CRUD operaciju, operaciju čitanja (eng. *Read*).

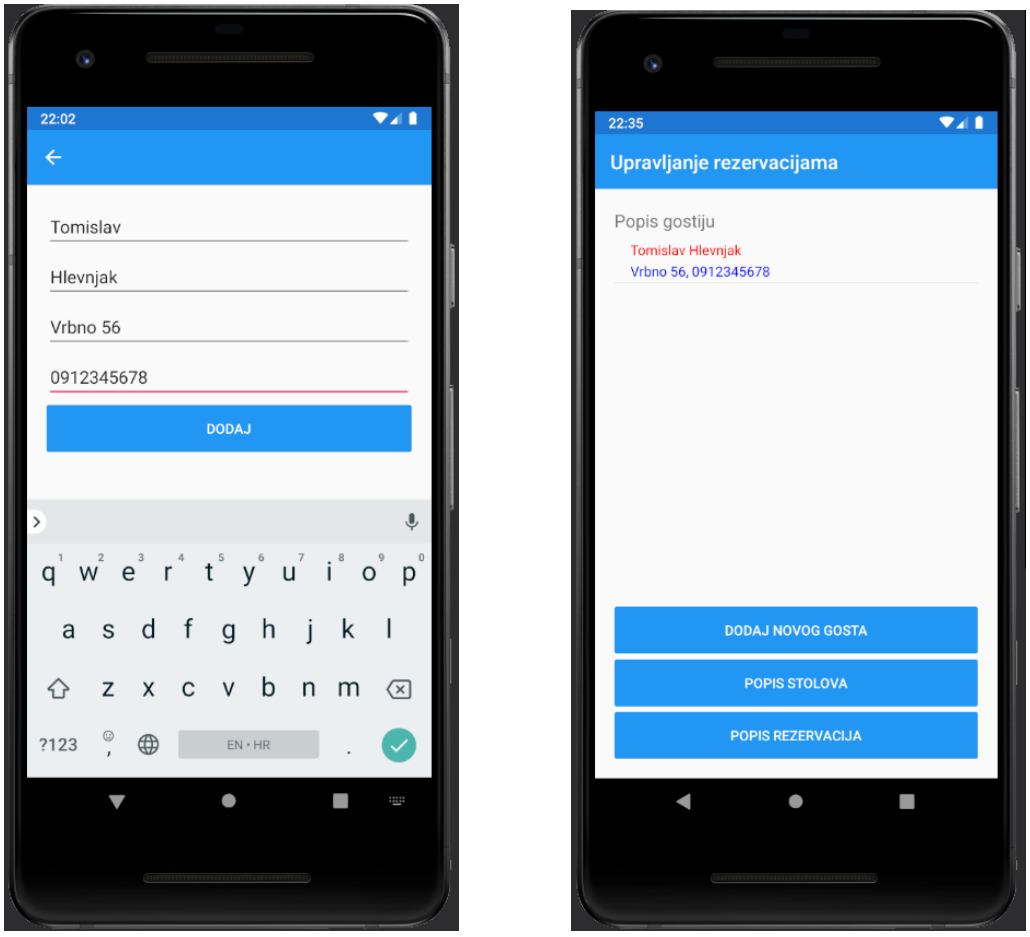
Operacija čitanja najjednostavnija je CRUD operacija jer sve što treba napraviti jest dohvatiti iz tablica podatke i spremiti ih u listu. Također se pomoću izraza *using* spajamo na bazu, te doslovce jednom linijom dohvaćamo podatke:

```
using(SQLiteConnection veza = new SQLiteConnection(App.PutanjaDoBaze))
    {
        var gosti = veza.Table<Gost>().ToList();
    }
```

Nakon što su podaci dohvaćeni u listu, tu listu potrebno je prikazati na ekran. Za to služi prikaz liste (eng. *ListView*) i njezino svojstvo *ItemsSource*. Listu *gosti* jednostavno pridružimo *ListViewu* pomoću svojstva *ItemsSource*:

```
sviGostiListView.ItemsSource = gosti;
```

Pogledajmo na Slici 5. kako izgleda dodavanje gosta i popis svih gostiju.



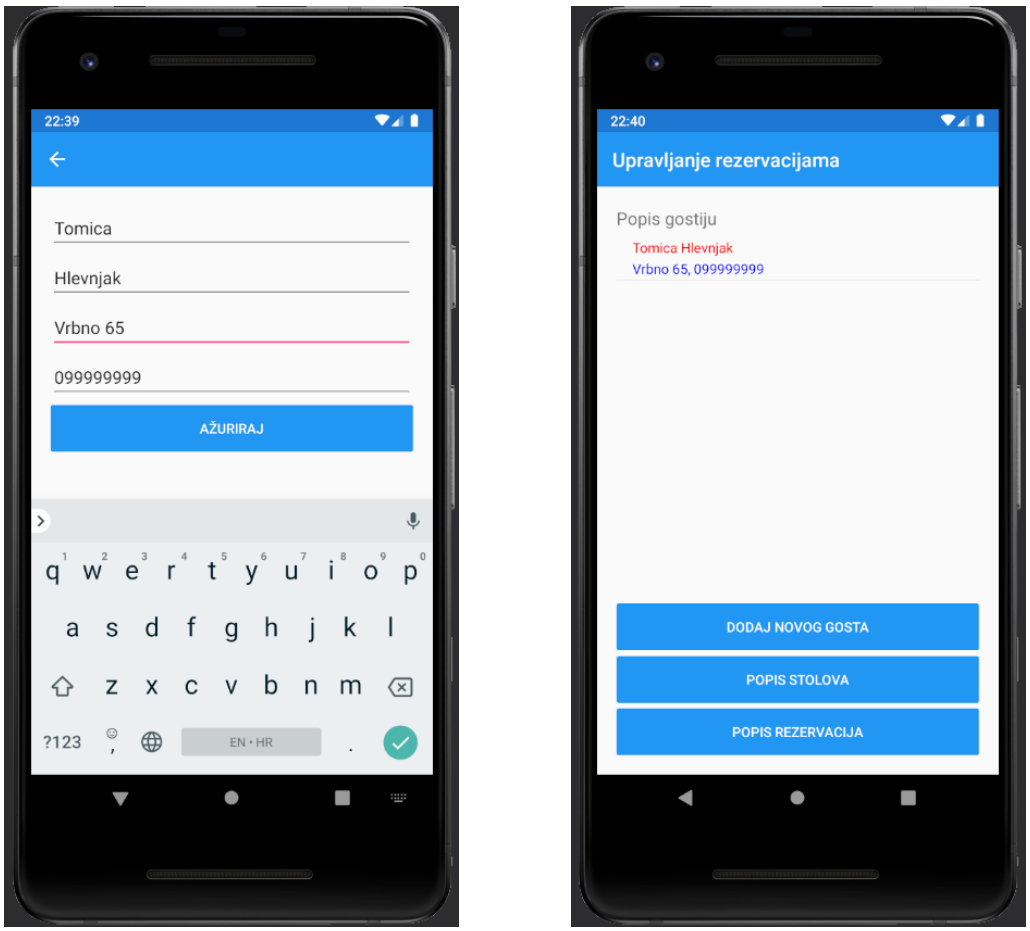
Slika 5. Dodavanje gosta i popis gostiju (vlastita izrada)

Često puta se dogodi da unesene podatke trebamo i promijeniti ili ažurirati. Zbog toga je vrlo važna implementacija treće CRUD operacije, operacije ažuriranja (eng. *Update*). Ažuriranje je u principu vrlo slično umetanju, razlika je što se ne dodaje novi ID, već se prema

postojećem mijenjaju podaci. U slučaju aplikacije, dohvaćamo s popisa gosta kojem želimo promijeniti podatke te otvaramo formu za ažuriranje. Konstruktoru nove forme, odnosno sadržajne stranice šaljemo dohvaćenu vrijednost s popisa te zatim instanciramo novi objekt klase, ali mu šaljemo već postojeći ID. Mijenjamo tražene podatke i klikom na gumb za ažuriranje izvodi se *Update()* metoda u izrazu *using*.

```
public partial class AzurirajGosta : ContentPage
    {
        private Gost _gost;
        public AzurirajGosta()
        {
            InitializeComponent();
        }
        public AzurirajGosta(Gost gost)
        {
            InitializeComponent();
            _gost = gost;
            imeEntry.Text = gost.Ime;
            prezimeEntry.Text = gost.Prezime;
            adresaEntry.Text = gost.Adresa;
            telefonEntry.Text = gost.Telefon;
        }
        private void azurirajButton_Clicked(object sender, EventArgs e)
        {
            Gost gost = new Gost()
            {
                ID_gost = Convert.ToInt32(_gost.ID_gost),
                Ime = imeEntry.Text,
                Prezime = prezimeEntry.Text,
                Adresa = adresaEntry.Text,
                Telefon = telefonEntry.Text
            };
            using (SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                veza.CreateTable<Gost>();
                int brRedaka = veza.Update(gost);
            }
            Navigation.PopAsync();
        }
    }
```

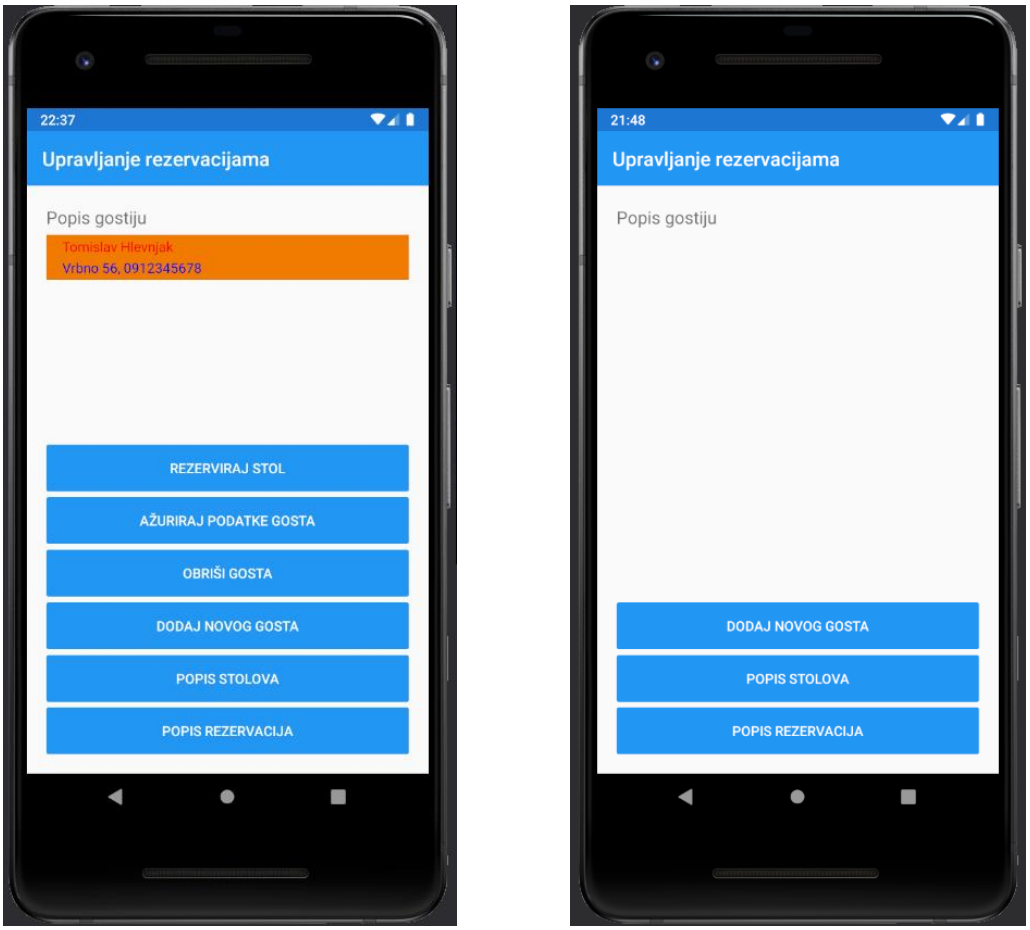
Da se radi o uspješnom ažuriranju znamo ako je vrijednost varijable *brRedaka* veća od 0, a kako to izgleda u aplikaciji možemo vidjeti na Slici 6.



Slika 6. Ažuriranje gosta i popis nakon ažuriranja (vlastita izrada)

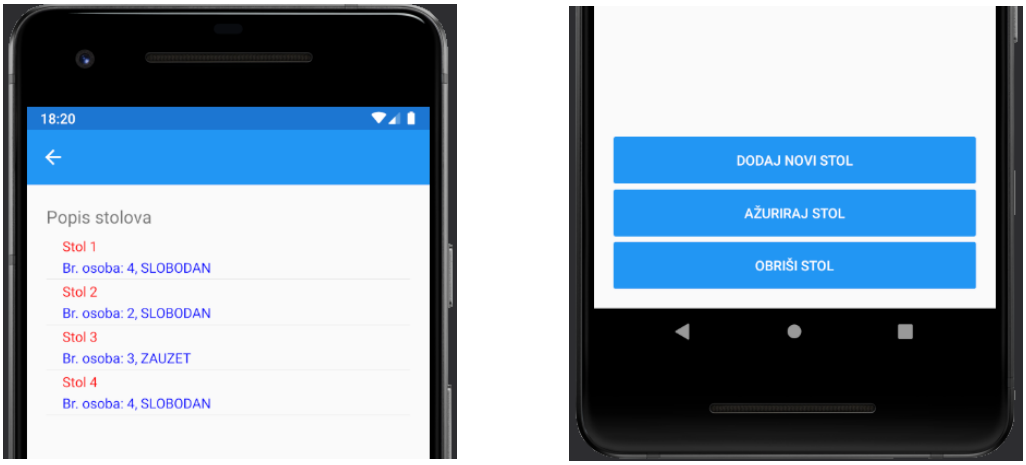
Posljednja operacija koja je preostala jest operacija brisanja (eng. *Delete*). Brisanje je također vrlo jednostavno implementirano u *SQLite.NET* biblioteci. Brisanje funkcionira na način da se metodi *Delete()* pošalje ili cijeli objekt ili primarni ključ objekta. Što god da poslali, metoda će sama zaključiti o kojem se tipu podatka radi i izvršit će se. U našem slučaju, s popisa gostiju odabire se gost kao objekt kojeg želimo obrisati, i on se proslijedi metodi za brisanje. Kôd za brisanje i kako brisanje izgleda u praksi vidimo u nastavku:

```
private void btnObrisi_Clicked(object sender, EventArgs e)
        {
            using (SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                int brRedaka = veza.Delete(_gost);
            }
        }
```



Slika 7. Odabir gosta i rezultat brisanja (vlastita izrada)

Time smo implementirali sve CRUD operacije vezane za entitet *Gost*. Implementacija CRUD operacija za *Stol* ne razlikuje se u mnogočemu od implementacije *Gosta*. Isto je bilo potrebno dodati novu sadržajnu stranicu na kojoj će se nalaziti popis svih stolova, te gumbe za dodavanje, ažuriranje i brisanje stolova. Pogledajmo na Slici 8. izgled stranice *PopisStolova*.



Slika 8. Popis stolova i CRUD operacije (vlastita izrada)

Preostalo je još implementirati CRUD operacije za *Rezervaciju*. Kako je *Rezervacija* složeni entitet bilo je potrebno osigurati da se prilikom brisanja ili *Stola* ili *Gosta* automatski obriše i njegova *Rezervacija*. Ranije je spomenuto kako *SQLite.NET* biblioteka standardno ne podržava mogućnost *Cascade* operacije, te se mora isprogramirati da se svakim brisanjem „roditelja“ obriše i njegovo „dijete“. Ta praksa nije baš najbolja ukoliko bi se radilo s velikim brojem tablica, no u ovom slučaju kada ih nema puno možemo iskoristiti.

Krenimo najprije s operacijom kreiranja, tj. umetanja rezervacije u bazu. Zamišljeno je da se klikom na gosta s popisa gostiju može kliknuti gumb „*Rezerviraj stol*“. Dakle, selektirani objekt *gost* se šalje konstruktoru nove sadržajne stranice *RezervirajStol* i zatim se izdvaja svojstvo, tj. atribut *ID_gost* koji je zapravo vanjski ključ u tablici *Rezervacija* da bi se mogao iskoristiti kod instanciranja novog objekta klase *Rezervacija*. Osim gosta, potrebno je naravno odabrati i stol koji se želi rezervirati, a to je implementirano kao padajući izbornik pomoću klase *Picker*. Prvo što je potrebno napraviti jest dohvatiti sve stolove koji nisu zauzeti, a to smo postigli sljedećom linijom:

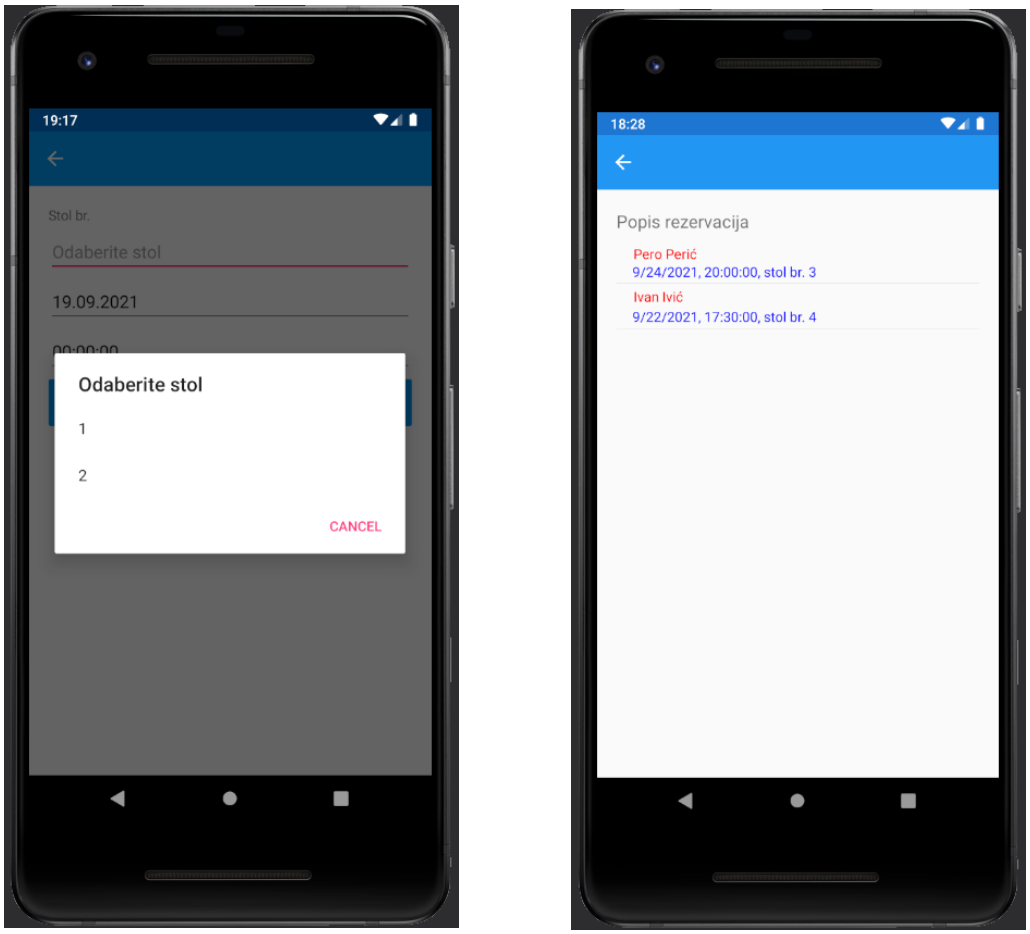
```
slobodniStolovi = veza.Table<Stol>().Where(s=>s.Zauzet == false).ToList();
```

Time su se u listu *slobodniStolovi* upisali oni stolovi koji su zadovoljili traženi upit, koji je zahtijevao sve one stolove čija je vrijednost svojstva *Zauzet* jednaka *false*. Zatim se u *Pickeru* kao *ItemsSource* dodjeljuje lista *slobodniStolovi*, a na događaj (eng. *Event*) *SelectedIndexChanged()* se onda uzima objekt *stol*, točnije njegovo svojstvo *ID_stol* koje je drugi vanjski ključ u tablici *Rezervacija*. Time su riješeni vanjski, odnosno primarni ključevi tablice i budući da imamo svojstva za odabir datuma i vremena, potrebno je i njih odabrati te je sve spremno za instanciranje novog objekta klase *Rezervacija* koji će biti dodan u bazu podataka. U nastavku možemo vidjeti programski kôd za dodavanje rezervacije:

```
public partial class RezervirajStol : ContentPage
    {
        private Gost _gost;
        private Stol _stol;
        public RezervirajStol()
        {
            InitializeComponent();
        }
        public RezervirajStol(Gost gost)
        {
            _gost = gost;
            InitializeComponent();
            using(SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
```

```
            {
                var slobodniStolovi = veza.Table<Stol>().Where(s =>
s.Zauzet == false).ToList();
                BindingContext = slobodniStolovi;
                stoloviPicker.ItemsSource = slobodniStolovi;
            }
        }
        private void stoloviPicker_SelectedIndexChanged(object sender,
EventArgs e)
        {
            var picker = sender as Picker;
            int selectedIndex = picker.SelectedIndex;
            if (selectedIndex != -1)
            {
                _stol = picker.ItemsSource[selectedIndex] as Stol;
            }
            datumDatePicker.MinimumDate = DateTime.Now.AddHours(2);
            datumDatePicker.MaximumDate = DateTime.Now.AddDays(30);
        }
        private void rezervirajButton_Clicked(object sender, EventArgs e)
        {
            Rezervacija rezervacija = new Rezervacija()
            {
                ID_gost = _gost.ID_gost,
                ID_stol = _stol.ID_stol,
                Datum_rezervacije = datumDatePicker.Date,
                Vrijeme_rezervacije = vrijemeTimePicker.Time
            };
            using (SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                veza.CreateTable<Rezervacija>();
                int brRedaka = veza.Insert(rezervacija);
                veza.Query<Stol>("UPDATE Stol SET Zauzet = true WHERE
ID_stol = ?", rezervacija.ID_stol);
            }
            Navigation.PopAsync();
        }
    }
```

Nakon unosa rezervacije preostalo je još ažurirati odabrani Stol, tj. vrijednost *Zauzet* na *true*. Pogledajmo na Slici 9. formu za rezervaciju.



Slika 9. Rezervacija stola i popis rezervacija (vlastita izrada)

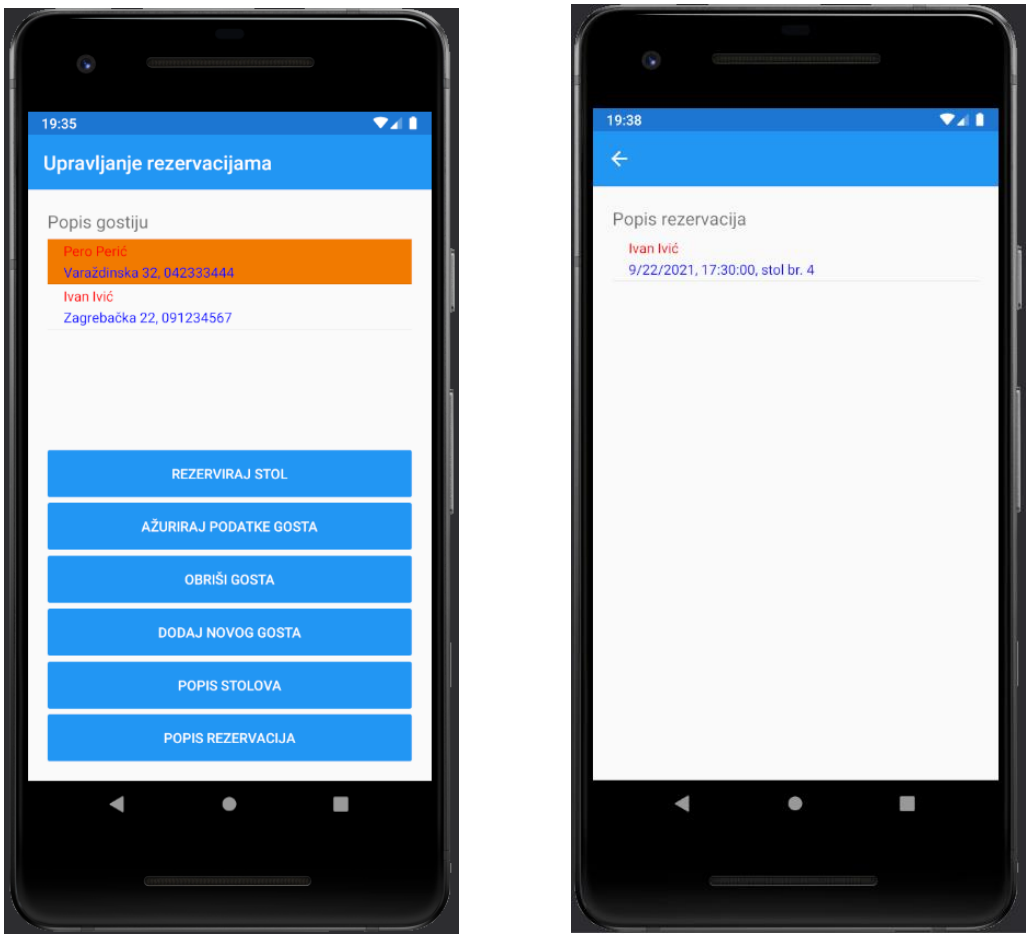
Implementacija ažuriranja rezervacije uključuje promjenu datuma i vremena, pa taj dio ne utječe previše na funkcionalnost aplikacije. S druge strane, operacija brisanja nam je veoma zanimljiva zbog problema referencijalnog integriteta.

Klikom na gumb „*Obriši rezervaciju*“ dogodi se upravo ono što i želimo, rezervacija je obrisana. Poslan je jednostavan upit bazi kojeg možemo vidjeti u sljedećoj liniji:

```
veza.Query<Rezervacija>("DELETE FROM Rezervacija WHERE ID_stol = ? AND
ID_gost = ?", _rezervacija.ID_stol, _rezervacija.ID_gost);
```

S popisa svih rezervacija dohvaćen je objekt *rezervacija* koji u sebi sadrži svojstva *ID_gost* i *ID_stol* pa je brisanje vrlo lagano jer se upitu proslijede ta svojstva. Međutim, želimo da se brisanjem gosta ili stola automatski obriše i rezervacija, što nije tako jednostavno budući da nema ograničenja *Cascade*.

Prema tome, napravljen je jedan upit koji je prilikom brisanja gosta automatski obrisao i njegovu rezervaciju te još jedan upit koji je prilikom brisanja stola automatski obrisao rezervacije tog stola. Pogledajmo na Slici 10. rad prvog upita.

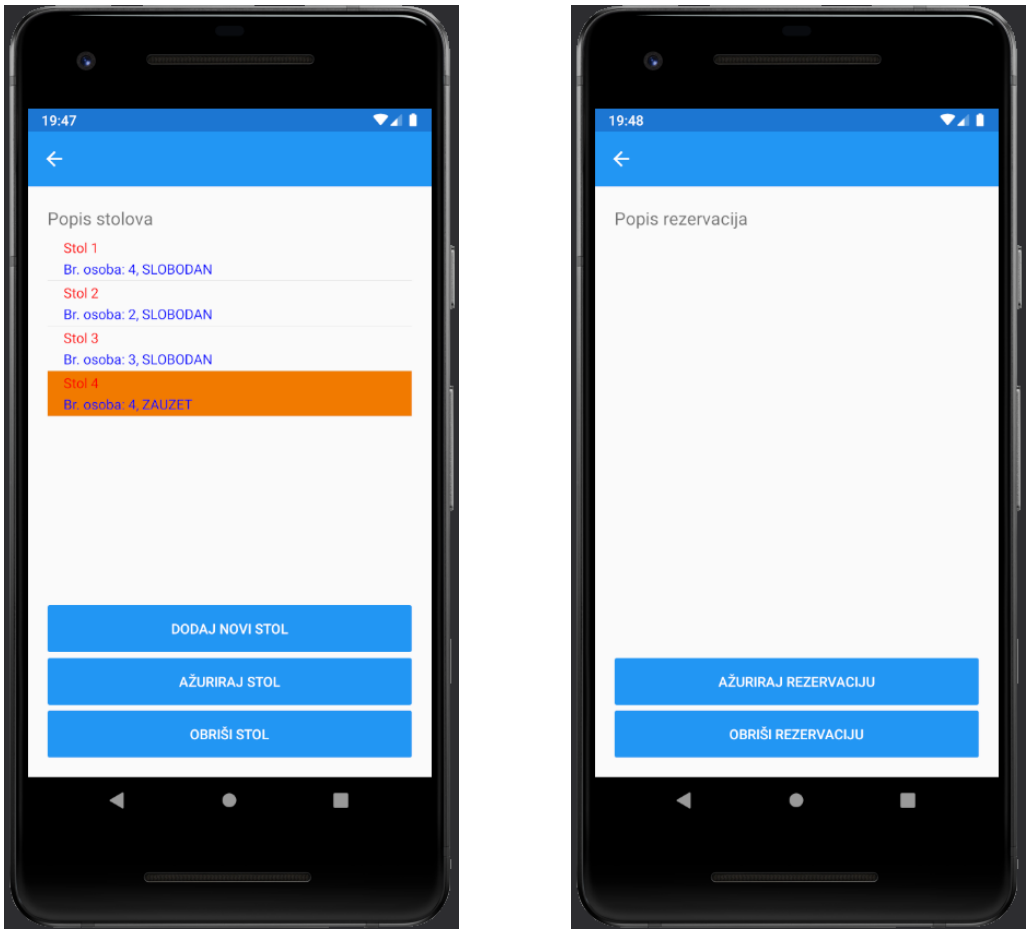


Slika 10. Rezervacije nakon brisanja gosta (vlastita izrada)

Na Slici 9. smo mogli vidjeti kako su Pero Perić i Ivan Ivić imali rezervacije, no brisanjem Pere Perića s popisa gostiju automatski je obrisana i njegova rezervacija. U kodu to izgleda ovako:

```
private void btnObrisi_Clicked(object sender, EventArgs e)
        {
            using (SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                veza.Query<Rezervacija>("DELETE FROM Rezervacija WHERE
ID_gost = ?", _gost.ID_gost);
                int brRedaka = veza.Delete(_gost);
                sviGostiListView.ItemsSource = veza.Table<Gost>().ToList();
            }
        }
```

Na Slici 10. možemo vidjeti kako postoji još jedna rezervacija gdje je rezerviran stol br. 4. Pogledajmo na Slici 11. što se dogodi kada obrišemo stol 4 iz baze podataka.



Slika 11. Rezervacije nakon brisanja stola (vlastita izrada)

Na Slici 10. mogli smo vidjeti kako je postojala jedna rezervacija stola 4, no brisanjem tog stola iz baze obrisana je i rezervacija. Kako je to implementirano u kodu možemo vidjeti u nastavku:

```
private void btnObrisi_Clicked(object sender, EventArgs e)
        {
            using (SQLiteConnection veza = new
SQLiteConnection(App.PutanjaDoBaze))
            {
                veza.Query<Rezervacija>("DELETE FROM Rezervacija WHERE
ID_stol = ?", _stol.ID_stol);
                int brRedaka = veza.Delete(_stol);
                sviStoloviListView.ItemsSource =
veza.Table<Stol>().ToList();
            }
        }
```

Time je implementacija mobilne aplikacije za upravljanje rezervacijama stola gotova. Implementirane su tražene CRUD operacije i administrator može lagano upravljati bazom.

6. Zaključak

Osim u poslužiteljskim računalima, SQL i NoSQL baze podataka svoju primjenu pronašle su i kod mobilnih uređaja. Kako svakodnevno svjedočimo rastu količine (ne uvijek i kvalitete) podataka nije čudno da želimo sve informacije imati nadomak ruke kako bi držali korak za svjetskim trendovima. Glavne značajke obiju vrsta baza podataka koriste se svugdje u mobilnim aplikacijama, od strukture podataka, preko skalabilnosti pa sve do transakcija.

Iako je SQLite vodeća mobilna baza podataka, sve veća popularnost NoSQL baza podataka u objektno-orijentiranom programiranju mogla bi značiti preokret u mobilnoj industriji. Međutim, nije baš da će vodeće mobilne tvrtke prestati ugrađivati SQL baze podataka samo zato jer su NoSQL baze sve popularnije. Važno je odmjeriti koja će baza više doprinijeti, bilo da se radi o povećanju podataka ili o nekim drugim čimbenicima. U današnje vrijeme postoji mnogo različitih online servisa na kojima svaka osoba može naučiti razvijati mobilne aplikacije. To je vrlo traženo područje koje je dobar i stabilan odabir za budućnost.

Kod implementacije bilo koje mobilne aplikacije, vrlo je važno obratiti pažnju na korisničko iskustvo. Ipak je krajnji korisnik onaj kome je aplikacija namijenjena te je on onaj koji daje povratne informacije kako bi razvojni programer eventualno popravio ili unaprijedio aplikaciju. Aplikacija napravljena u ovom završnom radu tek je pola ukupnog posla, jer aplikacija treba biti stavljena u pogon i za korisnika imati neku svrhu. No, tema je bila primjena baza podataka u izgradnji mobilnih aplikacija, a ne korisničko iskustvo.

Bilo da se radi o implementaciji aplikacije za osobnu upotrebu ili za širu distribuciju, jedno je sigurno – baze podataka su uvijek potrebne i imaju veliku ulogu, bila to mala baza s dvije tablice ili ogromna baza s puno tablica.

Popis literature

[1] „Draw.io – Diagrams.net“, (bez dat.). Dostupno: https://app.diagrams.net/ [pristupano 13.09.2021.].

[2] „Microsoft Visual Studio“, (bez dat.). u *Wikipedia, the Free Encyclopedia*. Dostupno: https://en.wikipedia.org/wiki/Microsoft_Visual_Studio [pristupano 13.09.2021.].

[3] „Run apps on the Android Emulator“, (bez dat.), u *Android Developers.* Dostupno: https://developer.android.com/studio/run/emulator [pristupano 13.09.2021.].

[4] Microsoft (bez dat.), *Xamarin* | *Open-source mobile app platform* [Na internetu]. Dostupno: https://dotnet.microsoft.com/apps/xamarin [pristupano 13.09.2021.].

[5] „What Is SQLite“, (bez dat.), u *SQLite*. Dostupno: https://www.sqlite.org/index.html [pristupano 13.09.2021.].

[6] K. Rabuzin, *Uvod u SQL*. Varaždin: Fakultet organizacije i informatike, Sveučilište u Zagrebu, 2011.

[7] N. Fatima, „A Quick Overview of Different Types of Databases“, 11.06.2019. Dostupno: https://www.astera.com/type/blog/a-quick-overview-of-different-types-of-databases/ [pristupano 14.09.2021.].

[8] G. Simsek, „What Is New About NewSQL?“, 24.02.2019. Dostupno: https://softwareengineeringdaily.com/2019/02/24/what-is-new-about-newsql/ [pristupano 14.09.2021.].

[9] J. Serra, „Relational databases vs Non-relational databases“, 27.08.2015. Dostupno: https://www.jamesserra.com/archive/2015/08/relational-databases-vs-non-relational-databases/ [pristupano 14.09.2021.].

[10] M. Smallcombe, „SQL vs NoSQL: 5 Critical Differences“, 23.07.2021. Dostupno: https://www.xplenty.com/blog/the-sql-vs-nosql-difference/ [pristupano 14.09.2021.].

[11] B. Anderson and B. Nicholson, „SQL v NoSQL Databases: What's the Difference?“, 25.06.2021. Dostupno: https://www.ibm.com/cloud/blog/sql-vs-nosql [pristupano 14.09.2021.]

[12] D. K. Foote, „A Brief History of Non-Relational Databases“, 19.06.2018. Dostupno: https://www.dataversity.net/a-brief-history-of-non-relational-databases/ [pristupano 14.09.2021.].

[13] MongoDB (bez dat.) *NoSQL vs SQL Databases* [Na internetu]. Dostupno: https://www.mongodb.com/nosql-explained/nosql-vs-sql#what-are-the-benefits-of-nosql-databases [pristupano 14.09.2021.]

[14] M. Chan, „SQL vs. NoSQL – what's the best option for your database needs?“, 03.05.2019. Dostupno: https://www.thorntech.com/sql-vs-nosql/ [pristupano 14.09.2021.].

[15] M. Berga and T. Franco, „SQL vs NoSQL: When to use?“, 01.04.2021. Dostupno: https://www.imaginarycloud.com/blog/sql-vs-nosql/ [pristupano 14.09.2021.]

[16] O. Wolfson (bez dat.), „Mobile Database“. [Na internetu]. Dostupno: https://link.springer.com/referenceworkentry/10.1007/978-0-387-39940-9_1362 [pristupano 15.09.2021.].

[17] Greenrobot (bez dat.), „Mobile databases: SQLite and SQLite alternatives for Android and iOS“ [Na internetu]. Dostupno: https://greenrobot.org/news/mobile-databases-sqlite-alternatives-and-nosql-for-android-and-ios/ [pristupano 15.09.2021.].

[18] K. Panchal, „Everything you need to know about SQLite Mobile database“, 19.02.2019. Dostupno: https://ourcodeworld.com/articles/read/737/everything-you-need-to-know-about-sqlite-mobile-database [pristupano 15.09.2021.].

[19] K. Rabuzin, *SQL – Napredne teme*. Varaždin: Fakultet organizacije i informatike, Sveučilište u Zagrebu, 2014.

Popis slika

Slika 1. Najbolje rangirani RSUBP, rujan 2021. 4

Slika 2. Najbolje rangirani NoSQL SUBP, rujan 2021. 6

Slika 3. ERA model baze za aplikaciju.....11

Slika 4. Stranica *MainPage* i *Pocetna*.....16

Slika 5. Dodavanje gosta i popis gostiju 18

Slika 6. Ažuriranje gosta i popis nakon ažuriranja.....20

Slika 7. Odabir gosta i rezultat brisanja 21

Slika 8. Popis stolova i CRUD operacije 21

Slika 9. Rezervacija stola i popis rezervacija 24

Slika 10. Rezervacije nakon brisanja gosta 25

Slika 11. Rezervacije nakon brisanja stola.....26

Popis tablica

Tablica 1. Usporedna tablica SQL i NoSQL baza 8