FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Seminar

Kolegiji:

Objektno oblikovanje

Naziv projekta:

BetterConnect

Uvod

Ideja ovog projekta je bila napraviti sustav za lako povezivanje ljudi s određenim interesima. U skladu sa sve većom informatizacijom društva i s željom da doprinesemo boljem povezivanju ljudi i socijalizaciji društva, uvodimo aplikaciju BetterConnect. Aplikacija je namijenjena osobama s različitim zanimanjima i područjima interesa, omogućuje im pronalazak ljudi s određenim interesima za dopisivanje i upoznavanje.

Korisnik se registrira u sustav pomoću korisničkog imena, lozinke i broja mobitela. Nakon registracije korisnik uređuje svoj profil odabirući područja interesa i pisanjem opisa profila. Prilikom svakog novog pristupa sustavu korisnik se treba logirati pomoću korisničkog imena i lozinke. Korisnik može pregledavati ostale registrirane korisnike, samostalno definirajući kriterij pretraživanja pomoću filtera interesa. Korisnik može započeti interakciju s nekim od izlistanih korisnika klikom na gumb “slanje zahtjeva”.

Slanjem zahtjeva korisnik traži broj mobitela drugog korisnika, kako bi mogao stupiti u kontakt s njim. Kada korisniku kojem je poslan zahtjev dođe obavijest o istom, ponuđene su mu dvije opcije, prihvatiti i odbiti pristigli zahtjev. Korisnik u bilo kojem trenutku može nastaviti s uređivanjem svog profila, dodati ili obrisati prethodno postavljeni interes, urediti opis i slično. Ponašanje registriranih korisnika kontrolira administrator sustava koji u slučaju ne primjerenog ponašanja blokira korisnika te njegovo ime prelazi u “Blocked user”, a broj mobitela u “000 0000 000”.

Requirements

Use cases

1. Registracija korisnika
2. Log in korisnika
3. Dodavanje interesa
4. Uklanjanje interesa
5. Uređivanje opisa profila
6. Pretraživanje korisnika
7. Slanje zahtjeva
8. Odgovaranje na primljene zahtjeve
9. Blokiranje korisnika od strane administratora

Use case 1: Registracija korisnika

Primary actor: Korisnik

Precondition: Uređaj na kojem se pokreće aplikacija je ispravan

Postconditions: Korisnik je uspješno registriran u sustav

Glavni uspješni scenarij:

1. Korisnik pokreće aplikaciju
2. Aplikacija je uspješno pokrenuta i traži od korisnika podatke za prijavu
3. Korisnik odabire opciju za registraciju te unosi korisničko ime, lozinku i broj mobitela
4. Podatci o korisniku se spremaju u sustav

Alternativni scenarij:

2a. Aplikacija nije uspješno pokrenuta

2a1. Ispisuje se poruka o pogrešci

3a. Korisnik ne unosi ispravno sve potrebne podatke za registraciju

3a1. Aplikacija ispisuje poruku o pogrešci pri unosu te omogućava korisniku ponovni unos potrebnih podataka

Use case 2: Log in korisnika

Primary actor: Korisnik

Precondition: Uređaj na kojem se pokreće aplikacija je ispravan, korisnik je uspješno registriran u sustav

Postconditions: Korisnik je uspješno prijavljen u sustav

Glavni uspješni scenarij:

1. Korisnik pokreće aplikaciju
2. Aplikacija je uspješno pokrenuta i traži od korisnika podatke za prijavu
3. Korisnik unosi korisničko ime i lozinku
4. Korisnik se prijavljuje u sustav

Alternativni scenarij:

4a. Ne postoji korisnik s unesenim korisničkim imenom

4a1. Ispisuje se poruka da korisnik ne postoji

4b. Unesena je pogrešna lozinka

4b1. Ispisuje se poruka o pogrešnom unosu lozinke

Use case 3: Dodavanje interesa

Primary actor: Korisnik

Precondition: Korisnik je uspješno prijavljen u sustav

Postconditions: Korisniku je pridijeljen određeni interes

Glavni uspješni scenarij:

1. Korisnik unutar aplikacije otvara detaljan prikaz svog profila
2. Odabere opciju uređivanje interesa
3. Dodaje vlastite interese

Use case 4: Uklanjanje interesa

Primary actor: Korisnik

Precondition: Korisnik je uspješno prijavljen u sustav

Postconditions: Korisniku je uklonjen određeni interes

Glavni uspješni scenarij:

1. Korisnik unutar aplikacije otvara detaljan prikaz svog profila
2. Odabere opciju uređivanje interesa
3. Uklanja neke od prethodno dodanih interesa

Use case 6: Pretraživanje korisnika

Primary actor: Korisnik

Precondition: Korisnik je uspješno prijavljen u sustav

Postconditions: Izlistani su svi korisnici s navedenim interesima

Glavni uspješni scenarij:

1. Korisnik u tražilicu unosi interese po kojima želi pretražiti korisnike
2. Odabire način filtriranja korisnika
3. Pretraga korisnika

Use case 7: Slanje zahtjeva

Primary actor: Korisnik

Precondition: Izlistani su korisnici s željenim interesima

Postconditions: Određenom korisniku je poslan zahtjev za slanjem broja mobitela

Glavni uspješni scenarij:

1. Korisnik odabire jednog izlistanog korisnika
2. Otvara se detaljniji pregled profila od odabranog korisnika
3. Pritiskom na gumb “Slanje zahtjeva” šalje se zahtjev

Use case 8: Odgovaranje na primljene zahtjeve

Primary actor: Korisnik

Precondition: Korisnik je primio zahtjev za slanjem broja mobitela

Postconditions: Primljeni zahtjev je obrađen

Glavni uspješni scenarij:

1. Korisnik odabere jedan od primljenih zahtjeva na koji želi odgovoriti
2. Određeni zahtjev za slanje broja mobitela prihvaća ili odbija

Use case 9: Blokiranje korisnika od strane administratora

Primary actor: Administrator

Stakeholders: Korisnik

Precondition: Korisnik se neprimjereno ponašao

Postconditions: Korisnik je blokiran

Glavni uspješni scenarij:

1. Administrator se prijavljuje u sustav
2. Traži određenog korisnika
3. Uklanja korisnika iz sustava

Opis (objektnog) modela

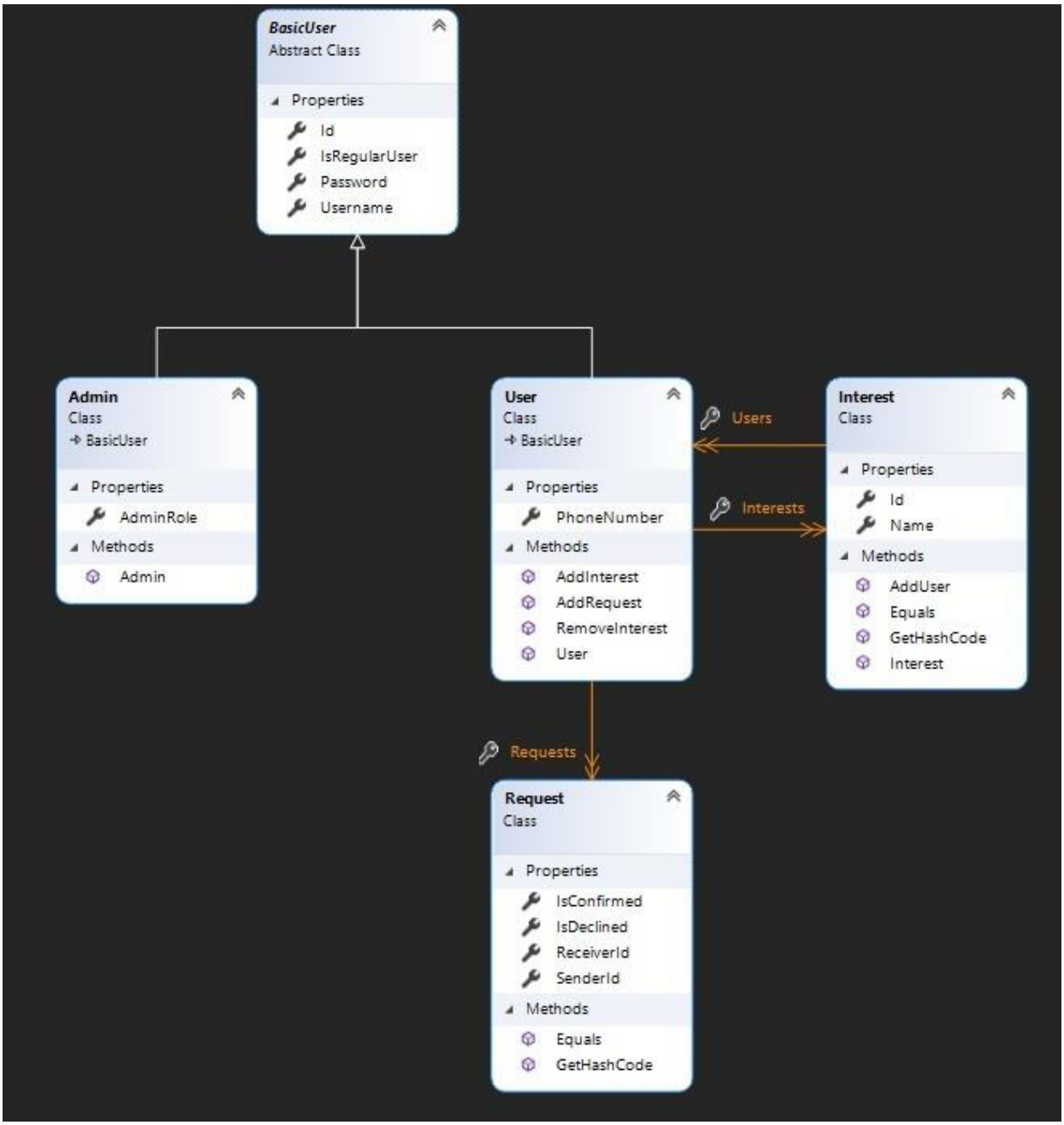
Klasa *BasicUser* osnova je za implementaciju korisnika koji ima profil na aplikaciji *BetterConnect*. Postoje dvije vrste korisnika, *User* koji je obični korisnik aplikacije i *Admin*, korisnik s većim ovlastima koji može blokirati ostale korisnike u slučaju dodavanja nekih neprimjerenih interesa, npr. koji su povezani sa sadržajem za odrasle ili nekim ilegalnim stvarima. Svaki korisnik u aplikaciji ima *id* i on predstavlja identifikator za pojedinog korisnika. Sastavni dio svakog korisnika je također i *username* koji mora biti jedinstven, kao i lozinka koja ne mora biti jedinstvena. Uz to u klasi *BasicUser* prisutan je *property IsRegularUser* koji govori je li korisnik običan korisnik ili administrator s većim ovlastima. Taj je *property* nužan kako bi se prilikom logiranja korisnika u sustav dokučilo treba li sustav vratiti model *User* ili *Admin*, koji su implementacija *BasicUser* apstraktne klase.

Svaki regularni korisnik u aplikaciji može imati popis interesa koji ga zanimaju. Klasa *Interest* i *User* model povezani su *Many-To-Many* vezom zato što svaki korisnik može imati više interesa i svaki interes može biti pridružen više korisnika. Interes se stvara tako da korisnik upiše ime interesa koji ga zanima, npr. “Nogomet”. Ako taj interes u bazi podataka već postoji, tada će se taj korisnik dodati u polje korisnika modela *Interest*. Ako je taj interes prvi tog imena, stvorit će se nova instanca klase *Interest*. Tako se jednostavno mogu dohvatiti svi korisnici koji imaju traženi interes putem tražilice u aplikaciji. Korisnik također može obrisati svoje interese. Bitno je napomenuti da ako određeni interes ima samo jednog korisnika, i taj korisnik odluči maknuti taj interes, ta instanca interesa se neće obrisati već će i dalje postojati u bazi podataka s već pridijeljenim identifikatorom.

Korisnici u sustavu si međusobno mogu slati zahtjeve. Svaki korisnik može vidjeti sve zahtjeve koji su pristigli od drugih korisnika, kao što može vidjeti i zahtjeve koje je on poslao prema drugim korisnicima. Na pristigle zahtjeve može se odgovoriti ili prihvaćanjem zahtjeva ili odbijanjem zahtjeva. Ako korisnik prihvati zahtjev, tada on neće vidjeti informacije o korisniku koji mu je taj zahtjev poslao, nego obrnuto, dakle isključivo korisnik koji je zahtjev poslao može vidjeti informacije o korisniku koji je isti taj zahtjev prihvatio. Ako korisnik prihvati zahtjev trenutno logiranog korisnika, tada će logirani korisnik pritiskom na taj zahtjev vidjeti dodatne informacije. Ukoliko korisnik zahtjev odbije, tada više nije moguće slati zahtjeve tom korisniku.

U svakom trenutku, trenutno logirani korisnik može vidjeti stanje zahtjeva koje je poslao ostalima. Takav zahtjev može biti na čekanju, može biti prihvaćen u kojem slučaju se prikazuju detaljnije informacije o korisniku koji je zahtjev poslao, ili može biti odbijen pri čemu se prikazuje poruka da je zahtjev odbijen.

Specijalni oblik korisnika je *Admin*, on ima mogućnost da blokira korisnike koji imaju neprimjerene interese. Ako administrator odluči blokirati korisnika tada se blokiranom korisniku *username* postavlja na “*Blocked*” i broj telefona na “000 000 000”. U tom slučaju ni jedan korisnik koji ima nekakvu vezu(primljeni ili poslani zahtjev) s blokiranim korisnikom, neće vidjeti prave informacije o njemu.


BasicUser
Abstract Class
Properties
Id
IsRegularUser
Password
Username
Admin
Class
BasicUser
Properties
AdminRole
Methods
Admin
User
Class
BasicUser
Properties
PhoneNumber
Methods
AddInterest
AddRequest
RemoveInterest
User
Users
Interests
Interest
Class
Properties
Id
Name
Methods
AddUser
Equals
GetHashCode
Interest
Requests
Request
Class
Properties
IsConfirmed
IsDeclined
ReceiverId
SenderId
Methods
Equals
GetHashCode

Opis implementacije perzistencije

Implementacija perzistencije podataka ostvarena je pomoću *frameworka* NHibernate, točnije FluentNHibernate koji pruža mogućnost mapiranja modela pomoću C# koda, dok klasični NHibernate to radi pomoću XML file-ova. Također, korišten je *Repository pattern* u kombinaciji sa servisima koji implementiraju logiku izvođenja željenih metoda. U nastavku će biti objašnjena struktura implementacije dijela za perzistenciju podataka.

Domenski sloj

U ovom sloju se nalazi čista implementacija modela domene čije je ponašanje objašnjeno u prethodnom poglavlju i čija se struktura može vidjeti na domenskom dijagramu razreda. Ovaj sloj je u potpunosti izoliran i on ne referencira niti sloj repozitorija niti sloj servisa jer za to nema nikakve potrebe.

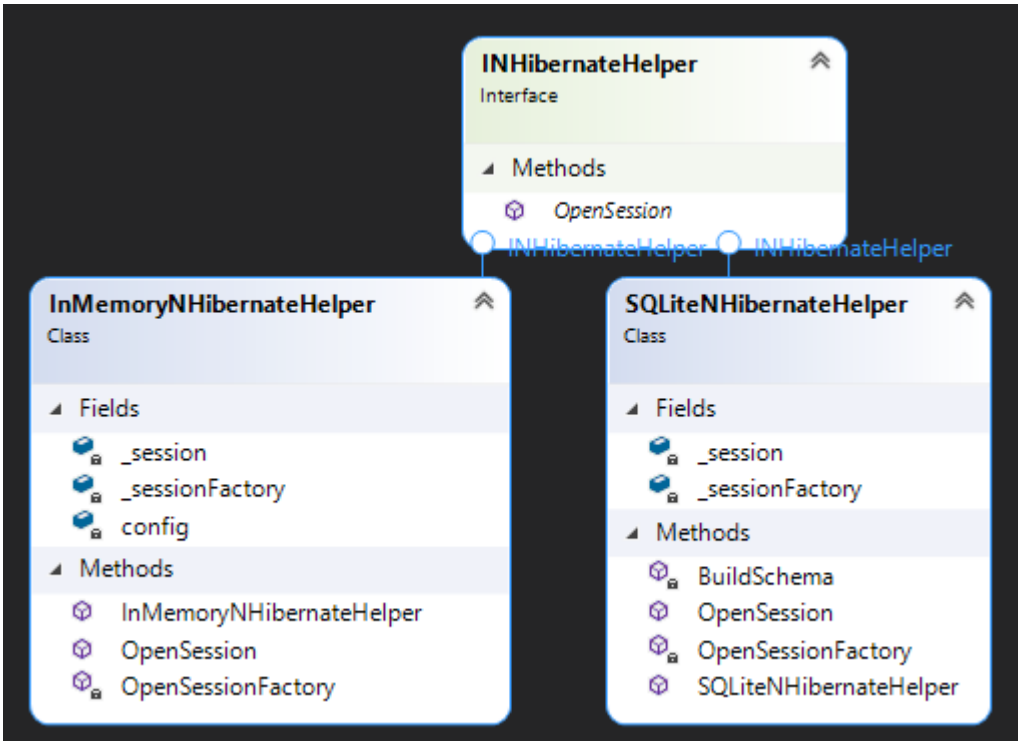
Opis mapiranja

- *BasicUser* – primarni ključ modela je *id* (*integer*), *username* je *not null* i *unique property*, *password* je *not null*
- *Interest* – primarni ključ modela je id (*integer*), *name* (naziv interesa) je *not null* i *unique property*
- *Request* – primarni ključ je kompozicija *id* pošiljatelja zahtjeva i *id* primatelja zahtjeva
- *User* i *Interest* modeli su povezani *Many-To-Many* vezom, što znači da *User* ima popis svojih interesa, dok interes ima popis *usera* koji ga posjeduju
- *User* i *Request* modeli su povezani *One-To-Many* vezom jer svaki *user* ima popis svih zahtjeva povezanih s njim, dalje na temelju *propertya* tih zahtjeva se određuje je li konkretni *user* pošiljatelj Ili primatelj tog zahtjeva

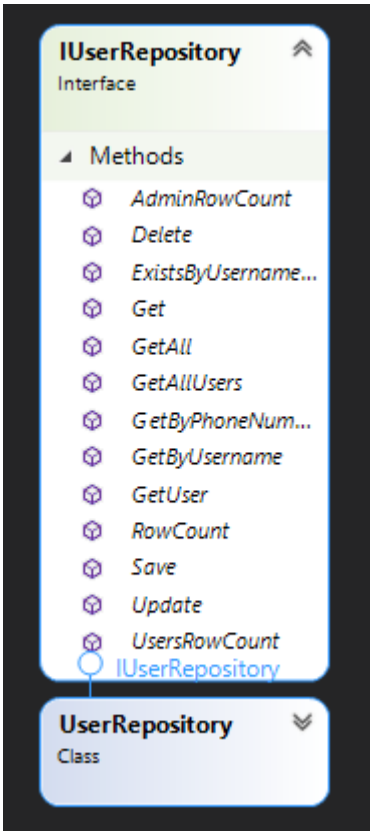
Repository sloj

Ovaj sloj predstavlja implementaciju isključivo dijela koji je zadužen za CRUD operacije nad slojem domene. Repozitoriji su ono što podatke čini trajnima, tj. ono što će pamtiti podatke i nakon gašenja aplikacije. Repozitoriji su implementirani kao sučelja i svi objekti koji referenciraju repozitorije ih referenciraju isključivo preko sučelja. To je nužno kako bi se ostvarila okolina za testiranje, ali i kako bi se omogućile različite implementacije tih repozitorija. Prvi i glavni razlog zašto želimo da su repozitoriji sučelja jest taj što bismo ga možda htjeli implementirati kao običnu listu u fazi testiranja koja će glumiti bazu podataka. Drugi razlog je taj što kada želimo staviti aplikaciju u produkciju, nećemo raditi s listama nego ćemo raditi s pravom bazom podataka, kao npr. SQLite baza. Zbog toga nam je nužna potreba za različitim implementacijama repozitorija. Od sada na dalje se referenciram isključivo na repozitorije koji za perzistenciju koriste

SQLite bazu podataka koja je korištena u ovom projektu. NHibernate pruža ISession objekt koji služi za komunikaciju sa SQLite bazom podataka. Taj objekt pruža sve CRUD metode, *Save*, *Update*, *Get* i *Delete.* SQLite pruža dvije mogućnosti implementacije baze koje su nama bile potrebne. Jedna je *in-memory* baza koja je korištena prilikom testiranja aplikacije. Druga baza je ona koja bi trebala ići u produkciju, tj. online. U ovom projektu je također implementirana baza koja može ići u produkciju, no nismo je stavili online. Možemo primijetiti da se pojavio problem, a to je kako implementirati sustav da dinamički mijenja implementaciju baze, *in-memory* za testiranje i bazu koja bi eventualno išla u produkciju. Taj problem smo riješili *Constructor Dependency Injection* tehnikom. Morali smo napraviti da u različitim situacijama sustav koristi različite ISession objekte, jedan ISession objekt koji radi s *in-memory* bazom i jedan koji radi s pravom bazom podataka. Za to smo iskoristili sučelje koje će nam ovisno o okolini (*test* ili *production*) dati željenu implementaciju ISession objekta. Željena implementacija se dohvaća pomoću *Factory patterna*. *Class* dijagram koji prikazuje rješenje problema je dan ispod teksta.



INHibernateHelper sučelje se injektira u konstruktor svakog konkretnog repozitorija koji radi sa SQLite bazom podataka i preko tog sučelja dobivamo željeni ISession objekt. Lijeva implementacija se koristi kod testiranja i ona nam daje *in-memory* bazu. Primjer sučelja i konkretne implementacije jednog repozitorija dan je na slici ispod.



Tada *constructor dependency injection* INHibernateHelper sučelja, u konkretnoj implementaciji repozitorija, u ovom slučaju *UserRepository* klasu u kodu izgleda ovako:

```
public class UserRepository : IUserRepository
{
    private ISession session;
    1 reference
    public UserRepository(INHibernateHelper helper)
    {
        session = helper.OpenSession();
    }
```

Znači, kada pozovemo konstruktor *UserRepository* klase, njemu ćemo predati jednu od dvije konkretne implementacije INHibernateHelper sučelja.

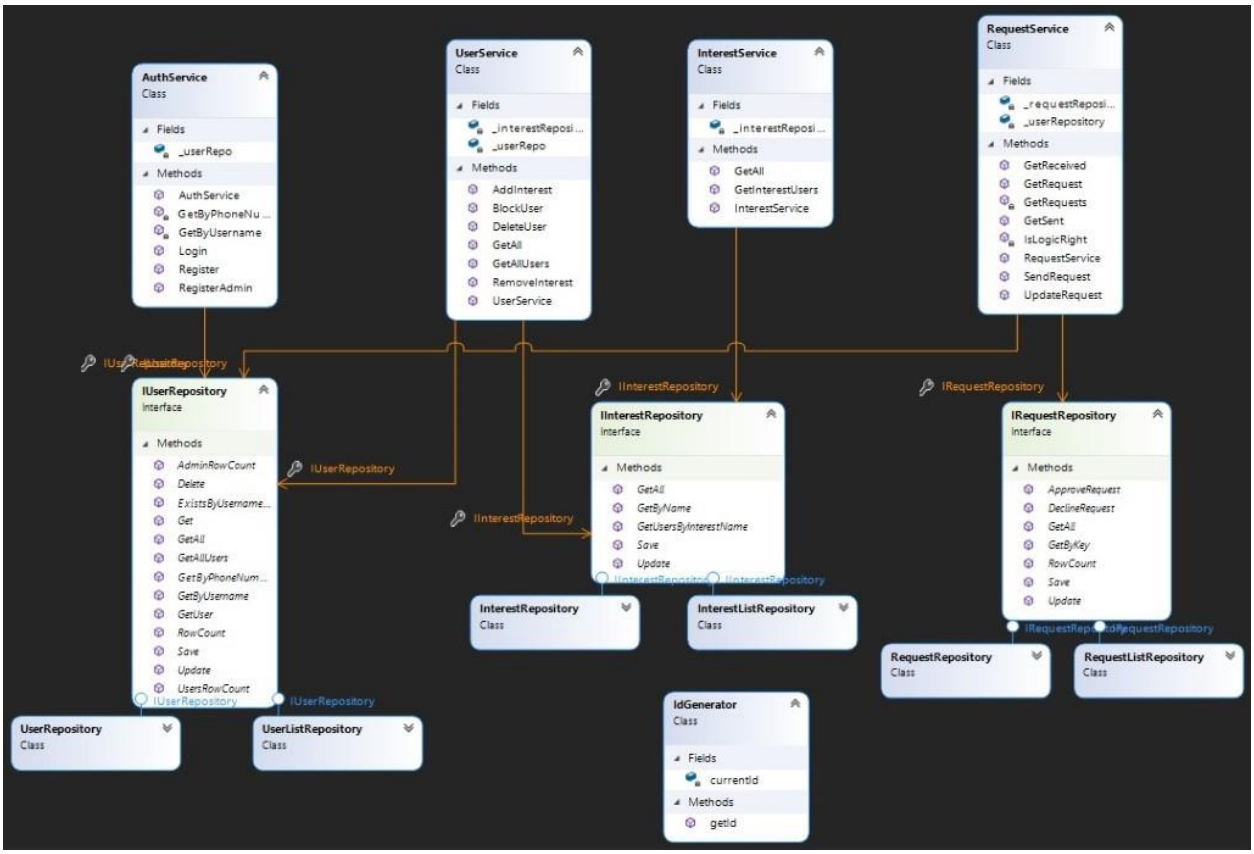
U sustavu postoji 3 repozitorija, *UserRepository* koji služi za CRUD operacije vezane uz model korisnika, *InterestRepository* koji služi za CRUD operacije vezane uz model interesa te *RequestRepository* koji služi za CRUD operacije modela zahtjeva koji se šalju između korisnika. Naravno, u sustavu postoje neka pravila kako se korisniku dodaju interesi, kada korisnik može drugom poslati zahtjev i slično. Nije zadaća repozitorija da provjeravaju logiku kojom se metode moraju izvršiti, već je to zadaća servisa koji će na ispravan način komunicirati s repozitorijima. O njima se govori u sljedećem potpoglavlju.

Service sloj

Servisi nam služe kako bismo uspješno proveli komunikaciju s repozitorijima. Na primjer, ako želimo registrirati novog korisnika, naš servis će prvo provjeriti jesu li podaci za registraciju ispravni, ako korisnik s tim korisničkim imenom već postoji, dojavit će se greška i komunikacija s repozitorijem se neće izvršiti. Ako su podaci ispravni, komunikacija s repozitorijem će se izvršiti i taj novo-registrirani korisnik će biti pohranjen u bazu podataka. U ovom projektu su prisutna četiri servisa:

- *AuthService* - služi za svu logiku vezanu uz registraciju i login korisnika
- *UserService* - služi za dohvat korisnika, dodavanje i micanje interesa korisnika
- *InterestService* - služi za dohvat svih interesa i za dohvat svih korisnika koji posjeduju definirani interes
- *RequestService* - služi za dohvat zahtjeva po ključu, dohvat svih poslanih i primljenih zahtjeva korisnika, kao i slanje, prihvat i odbijanje zahtjeva

Class dijagram servisa i repozitorija dan je na slici ispod.



Možemo vidjeti da svaki servis ima reference na repozitorije koji su mu potrebni za izvršavanje zahtjeva. Ovdje se također koristi *constructor dependency injection* pa tako lako možemo mijenjati implementacije repozitorija koje servis koristi. Primjer injektiranja repozitorija može se vidjeti na slici ispod.

```
public class RequestService
{
    private IRequestRepository _requestRepository;
    private IUserRepository _userRepository;

    5 references
    public RequestService(IRequestRepository requestRepo, IUserRepository userRepo)
    {
        this._requestRepository = requestRepo;
        this._userRepository = userRepo;
    }
```

Naravno, prilikom instanciranja servisa, u ovom slučaju *RequestService*, potrebno je definirati ispravnu implementaciju sučelja repozitorija. Kako se ispravno instanciraju servisi, može se vidjeti u WEB API projektu koji izlaže resurse pomoću kojih se pristupa bazi podataka putem URL-ova i HTTP komunikacije s API kontrolerima. O WEB API projektu koji služi za komunikaciju baze podataka i frontenda bit će riječi kasnije.

Testiranje

Domenski sloj i sloj repozitorija testirani su uz pomoć *in-memory* SQLite baze podataka. Ono najbitnije, sloj servisa testiran je uz pomoć *frameworka* Moq koji nudi metode za mockanje podataka. Svaki servis koji se poziva koristi neki repozitorij koji mu je potreban. Nužno je mockati metode repozitorija koje se pozivaju u metodama servisa koje se ispituju.

Povezivanje Backend-Frontend

Kako bismo mogli primati zahtjeve za resursima koje šalje frontend, u našem slučaju *Desktop* i *Flutter* aplikacija, moramo definirati ulazne točke u backend sustav. To smo napravili pomoću WEB API projekta. To se radi tako da se u projektu naprave kontroleri koji primaju zahtjeve, dalje te zahtjeve prosljeđuju servisima backend-a, koji zatim komuniciraju s repozitorijima i sa samom bazom podataka. Prije samog naziva klase kontrolera, definira se anotacija koja prikazuje vršni put tog kontrolera. Primjer je dan na slici u nastavku.

```
[ApiController]
[Route("/api/auth")]
1 reference
public class AuthController : Controller
{
    IUserRepository _userRepo;
    IInterestRepository _interestRepo;

    AuthService _authService;
    UserService _userService;

    0 references
    public AuthController(IUserRepository userRepository, IInterestRepository interestRepository)
    {
        this._userRepo = userRepository;
        this._interestRepo = interestRepository;

        _authService = new AuthService(_userRepo);
        _userService = new UserService(_userRepo, _interestRepo);
    }
```

Vidimo da konstruktor kontrolera također prima sučelja repozitorija što znači da negdje moramo zadati koju će implementaciju sučelja kontroler koristiti. WEB API framework pruža nam odličnu stvar za to, a to je *Dependency Injection Container*. U njemu definiramo konkretne implementacije za dana sučelja i to se radi na vrlo jednostavan način. U Startup.cs file-u postoji funkcija public void ConfigureServices(IServiceCollection services) koja konfigurira svu početnu konfiguraciju WEB API projekta. Način definiranja konkretnih implementacija sučelja dan je na slici ispod.

```
services.AddTransient<INHibernateHelper, InMemoryNHibernateHelper>();
services.AddSingleton<IUserRepository, UserListRepository>();
services.AddSingleton<IInterestRepository, InterestListRepository>();
services.AddSingleton<IRequestRepository, RequestListRepository>();
```

Sada, kada kontroler primi neki zahtjev, on pomoću URL-a koji mu je poslan zna koju metodu kontrolera treba pozvati, pa tako npr. ako pošaljemo zahtjev na URL:

https://localhost:5001/api/auth/getAllUsers

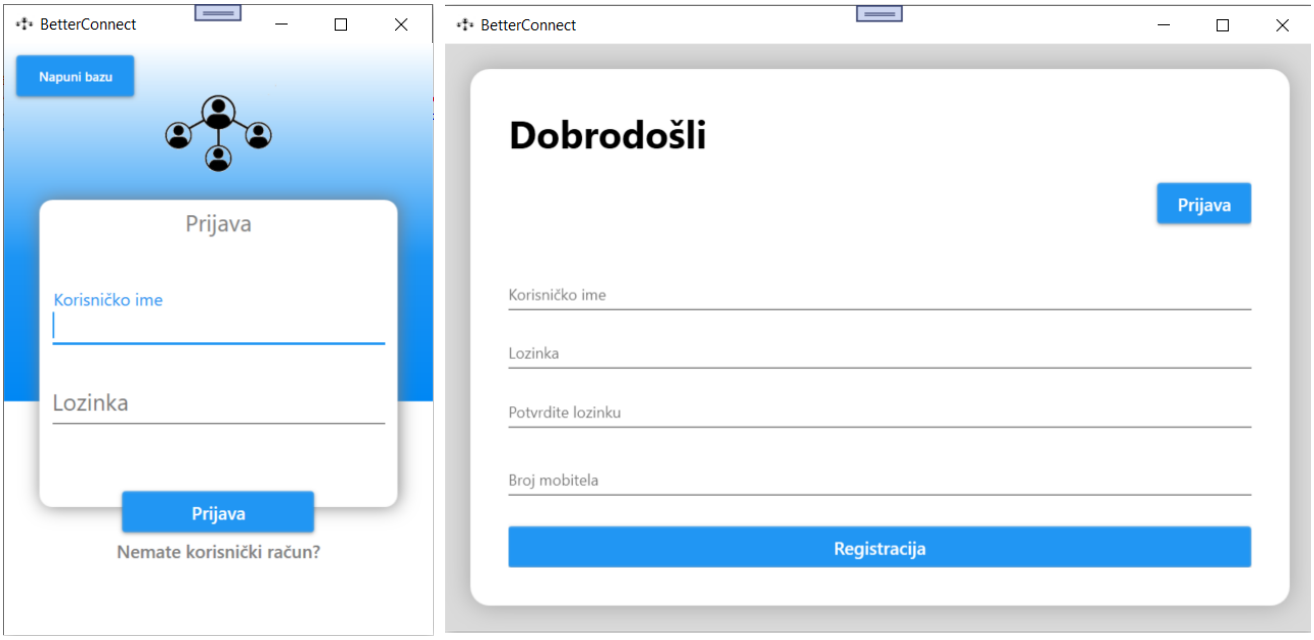
kontroler će znati da mora otići u metodu čiji je *path* završava na *getAllUsers*, a to je upravo metoda na sljedećoj slici.

```
[HttpGet("getAllUsers")]
0 references
public IList<User> GetAllUsers()
{
    return _userService.GetAllUsers();
}
```

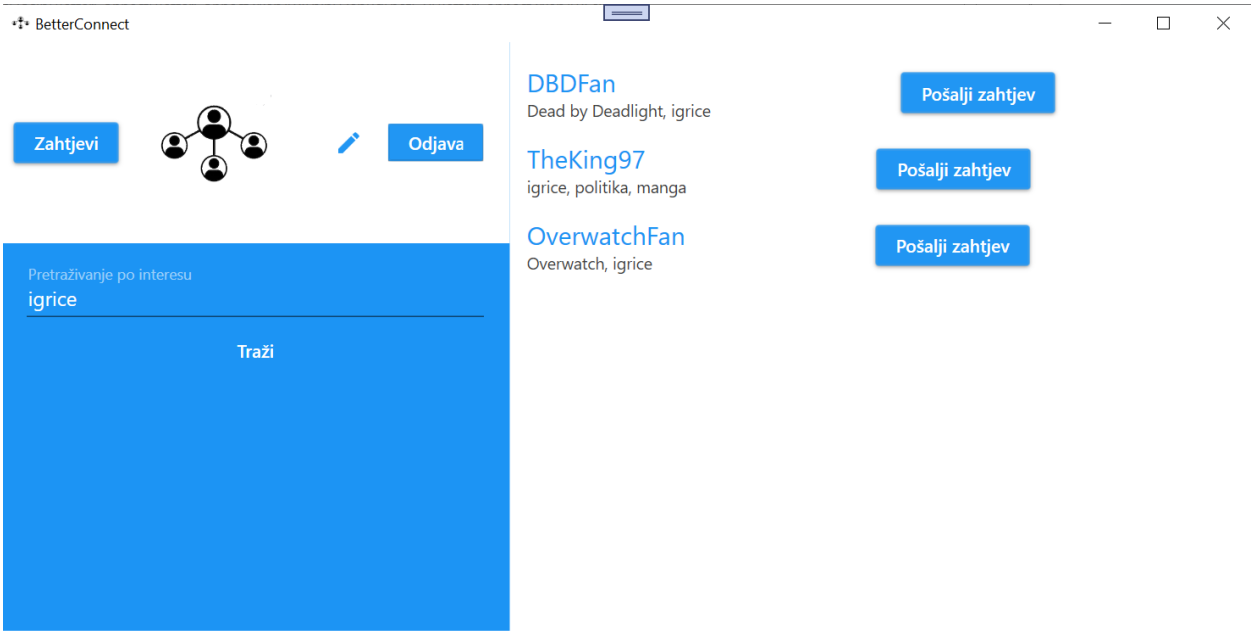
Tako funkcionira cijeli WEB API projekt, uz malu napomenu da prilikom *POST* i *PUT* zahtjeva obično šaljemo neke informacije na server, a te informacije modeliramo *Data Transfer Object (DTO)* objektima koji su sačinjeni samo od *propertya* koje šaljemo na server.

Opis izgrađene desktop aplikacije

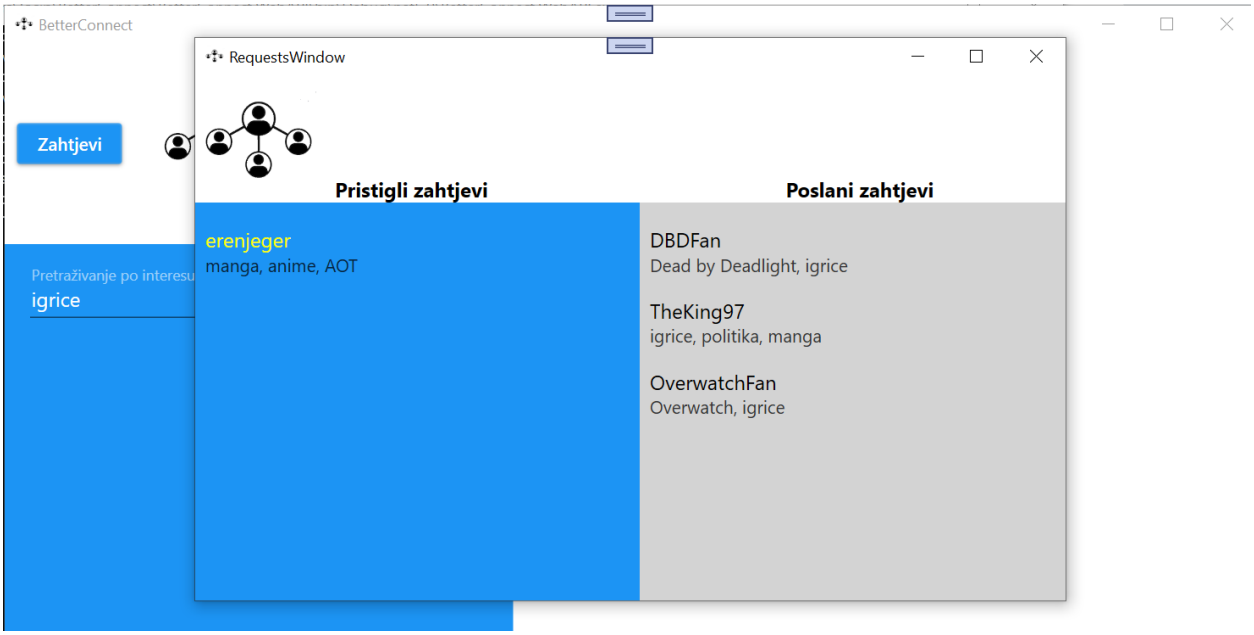
BetterConnect Desktop aplikacija ostvarena je u WPF tehnologiji. WPF aplikacije savršeno idu uz MVVM oblikovni obrazac. U tom obrascu želimo izdvojiti logiku izvođenja u modele koji se nazivaju ViewModel i upravo je to napravljeno u ovoj aplikaciji. Arhitektura se sastoji od modela domene (Models), view-modela (ViewModel) i od view-a (u WPF-u to su Window objekti s pripadajućim dizajnom). Primjerice, na glavnom ekranu aplikacije prikazani su svi postojeći korisnici u sustavu, i objekt Window (koji prikazuje taj ekran) vuče podatke iz objekta UsersViewModel koji sadrži sve korisnike, kao i odgovarajuće akcije koje se trebaju dogoditi kada se klikne na jednog korisnika. Tako se postiže veće odvajanje i izolacija cjelina, pa tako u potpunosti odvajamo dizajn, modele i *business* logiku. Način komunikacije između 2 prozora, tj. 2 ekrana smo ostvarili koristeći *Delegate pattern*. Imamo situaciju u kojoj u jednom prozoru upisujemo nekakve podatke, a u drugom prozoru se ti podaci također moraju mijenjati, znači 2 prozora nekako moraju komunicirati. *Delegate pattern* nam omogućava da jedan prozor pristupa drugom prozoru isključivo preko sučelja. To je jako bitno jer putem tog sučelja ćemo dozvoliti da prozor dijete (imamo prozor roditelj i prozor dijete) komunicira s roditeljem samo preko strogo definiranih pravila (tj. metoda). Bit je da prozor dijete ne može koristiti svaku metodu i vidjeti svaku varijablu roditelja već samo one koje mu dozvolimo putem delegate sučelja. Prozor roditelj implementira delegate sučelje i postavi se na delegate property od djeteta I tako su njih dvoje povezani. Bitno je naglasiti da svaki model u MVVM oblikovnom obrascu mora naslijediti sučelje INotifyPropertyChanged kako bi *window* znao da mora osvježiti svoje stanje u slučaju ako se taj model promijeni.



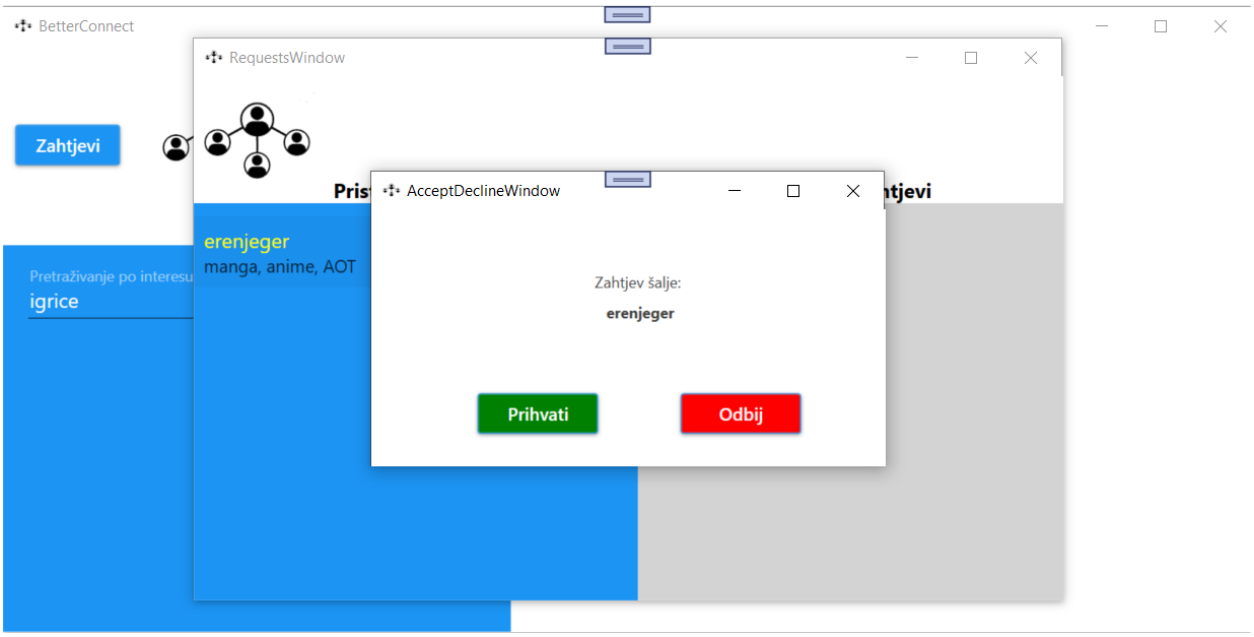
Login i registracija korisnika, prilikom pokretanja aplikacije pritiskom na gumb “Napuni bazu“ baza se puni s određenim vrijednostima unesenim u *source* kodu



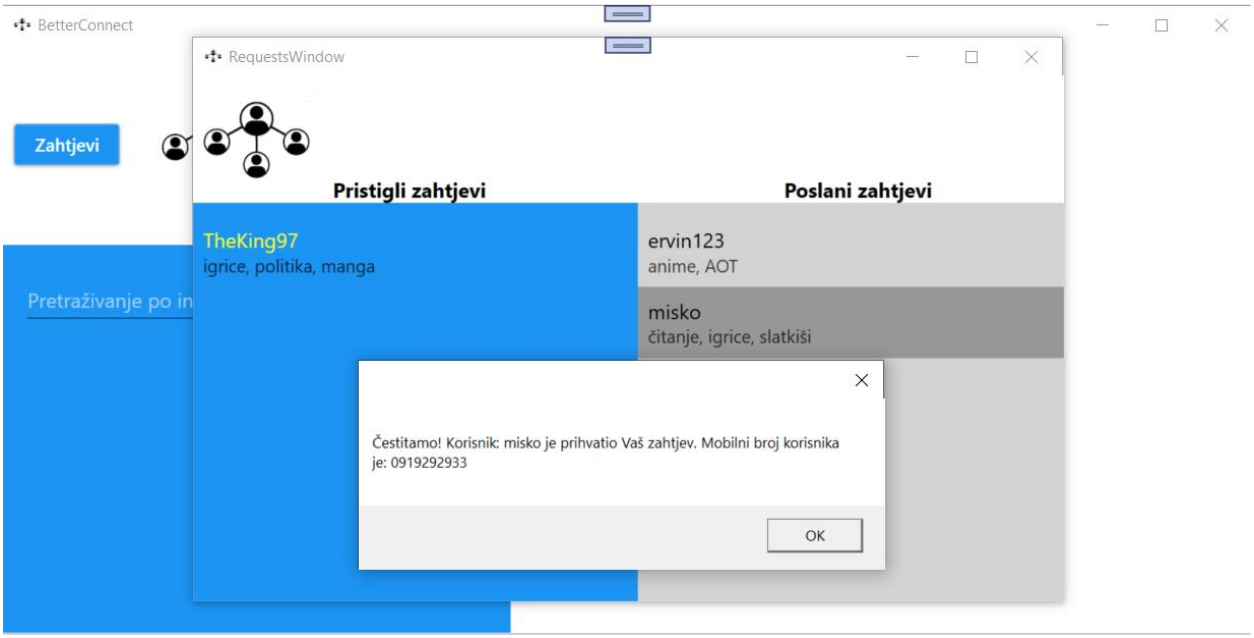
Pretraga korisnika po interesima



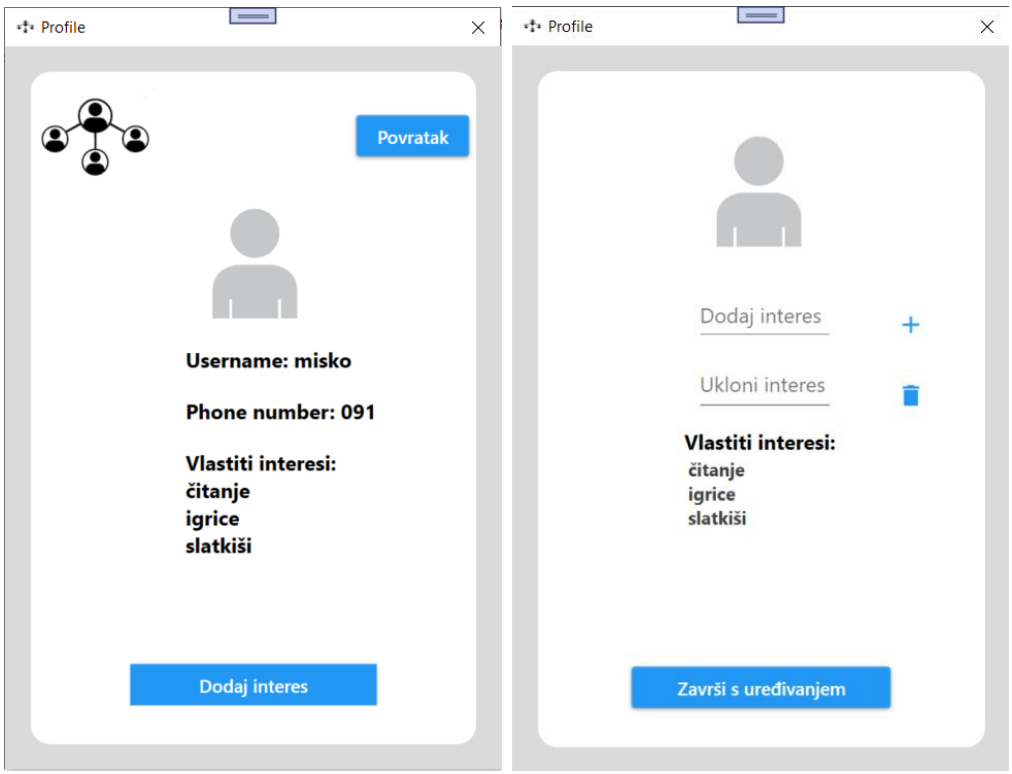
Pregled pristiglih zahtjeva



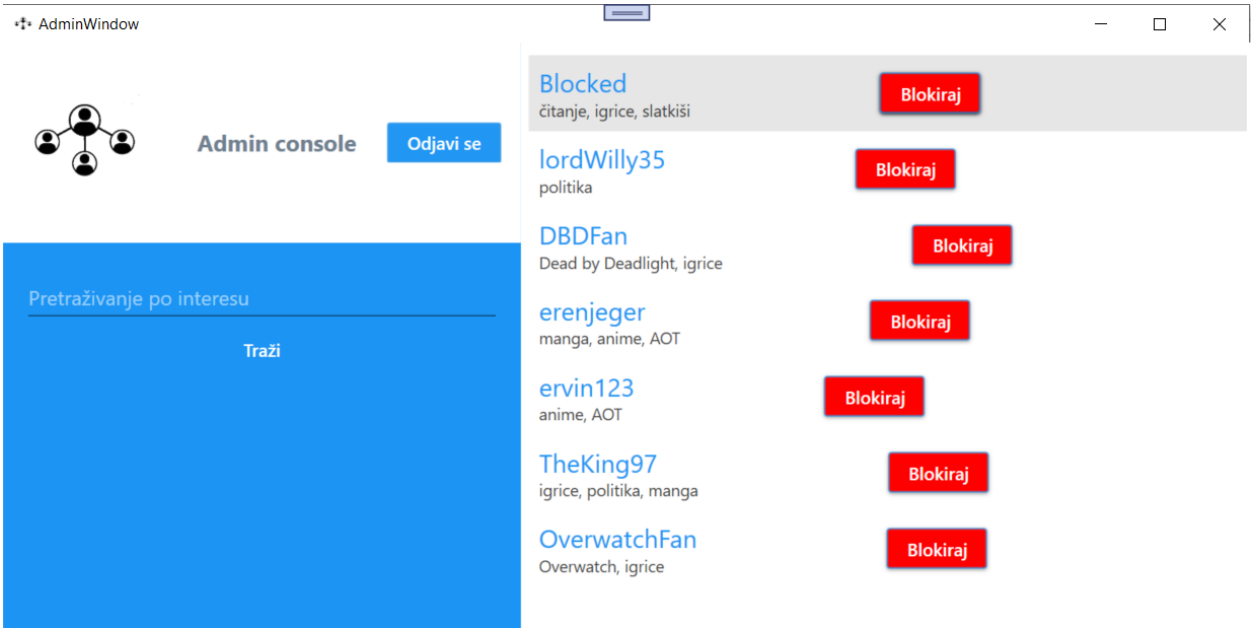
Korisnik odgovara na pristigli zahtjev



Kada korisnik kojem smo poslali zahtjev prihvati isti



Pregled profila i uređivanje interesa

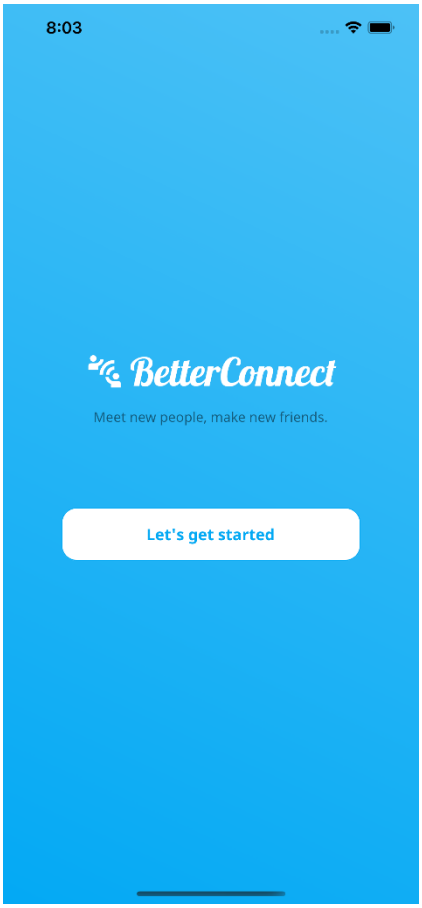


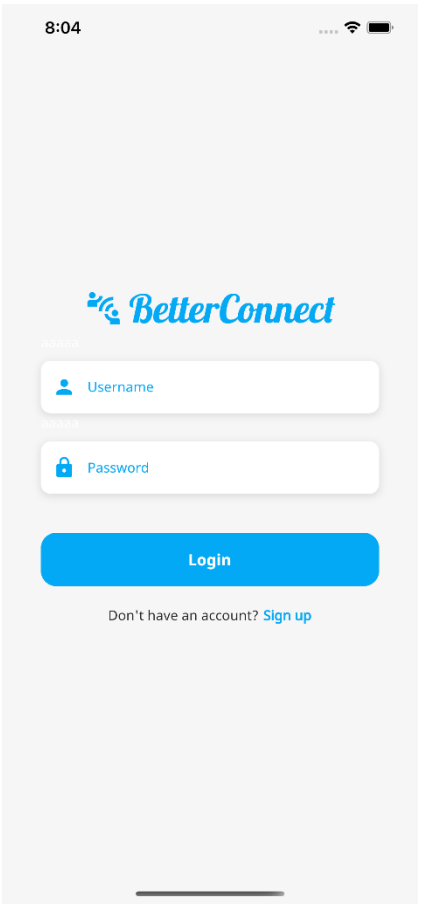
Administrator ima pregled korisnika, može ih pretraživati po odabranim interesima i u slučaju neprimjerenih interesa blokirati

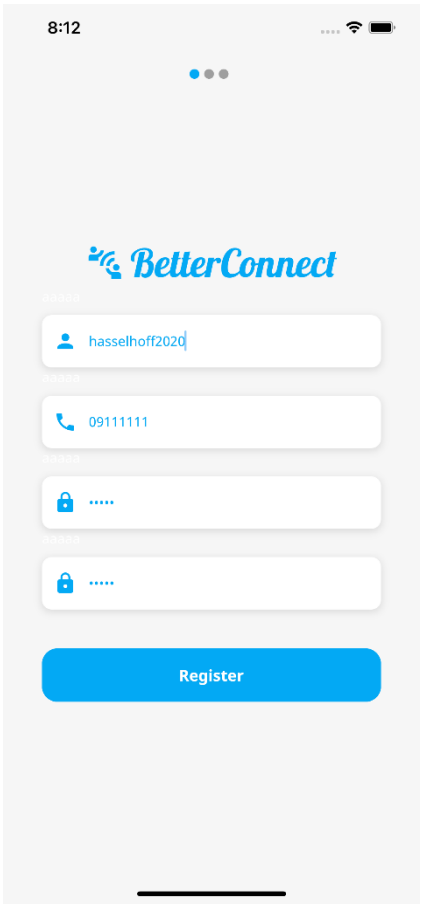
Opis izgrađene mobilne aplikacije

Aplikacija je jednostavna i intuitivna za korištenje. Sastoji se od dvije veće cjeline koje su razrađene u nastavku.

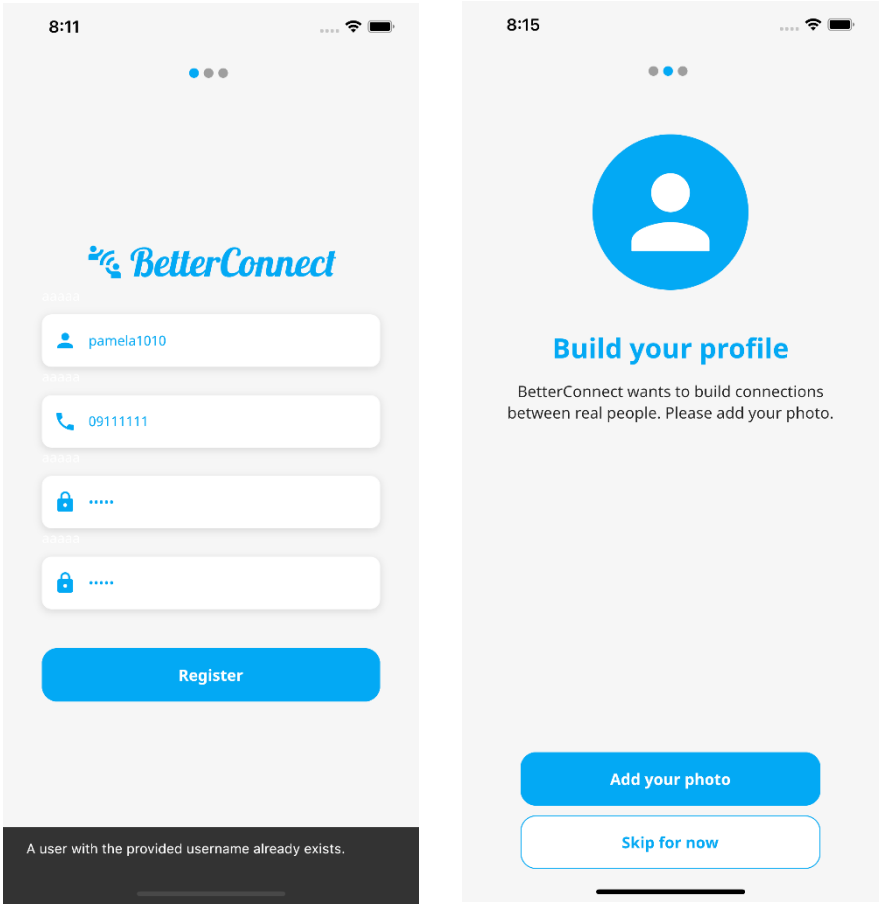
Prilikom ulaska u aplikaciju korisnika dočeka zaslon s porukom dobrodošlice. Pritiskom na gumb, korisnik biva proslijeđen na zaslon za prijavu gdje može odabrati želi li se prijaviti ili registrirati. Pritiskom na gumb za registraciju prikazuje se zaslon s registracijskom formom.



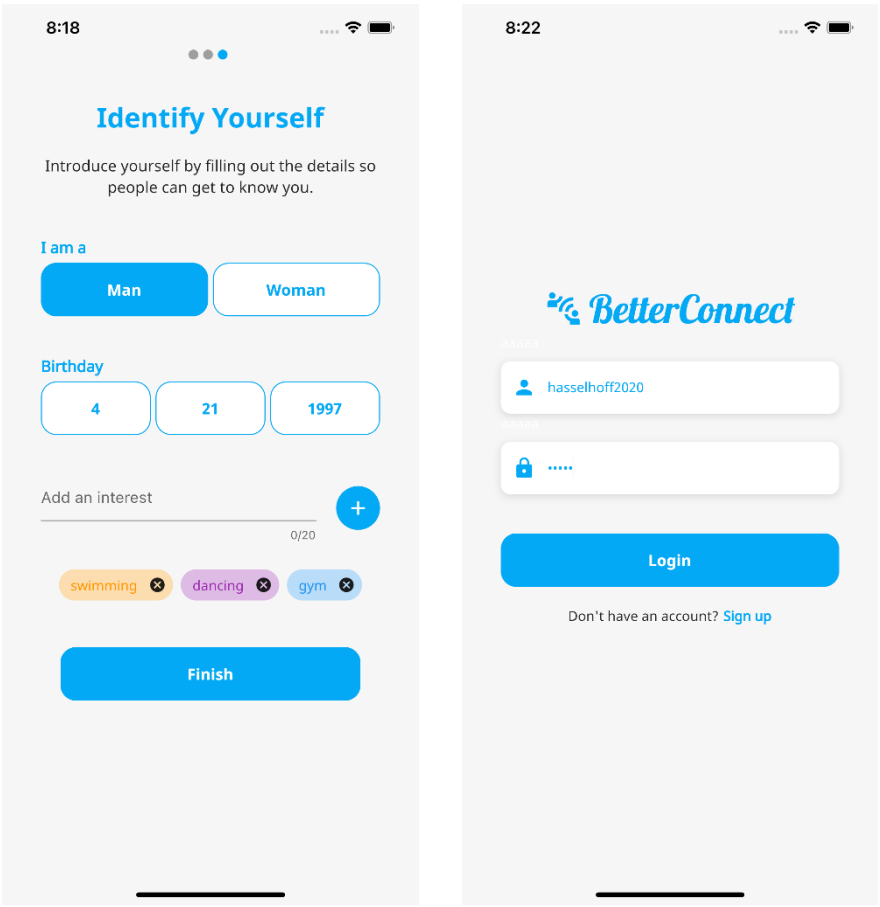




Ako korisnik unese nadimak ili telefonski broj koji već postoji, prikazuje mu se adekvatna poruka. Također, ako lozinka nije pravilno potvrđena ispisuje se adekvatna poruka. Ako su sve informacije pravilno unesene slijedi zaslon gdje korisnik može učitati svoju fotografiju iz galerije ili s kamere ako to želi.



Slijedi zaslon za unošenje osobnih podataka, gdje korisnik može odabrati spol, datum rođenja i navesti svoje interese. Ako korisnik već ima račun može se jednostavno prijaviti putem već viđenog obrasca.



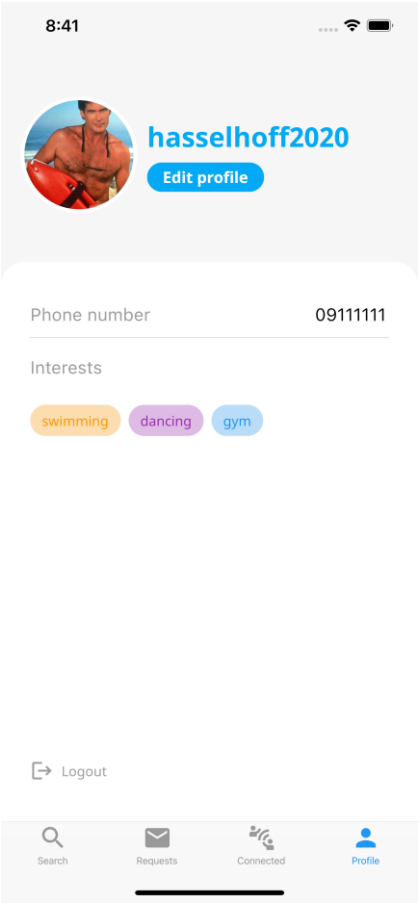
Glavni dio aplikacije sastoji se od četiri kartice: pretraživanje, zahtjevi, kontakti i profil. Na prvoj kartici korisnik može pretraživati ostale korisnike aplikacije po interesima. Ako želi stupiti u kontakt s tom osobom mora pritisnuti gumb “*Send request*”. Na kartici sa zahtjevima su prikazani svi pristigli zahtjevi koje korisnik može odbiti ili potvrditi. Ako korisnik potvrdi pristigli zahtjev ili neki drugi korisnik potvrdi njegov/njezin, korisnik će biti vidljiv u kontakt kartici zajedno s telefonskim brojem korisnika.

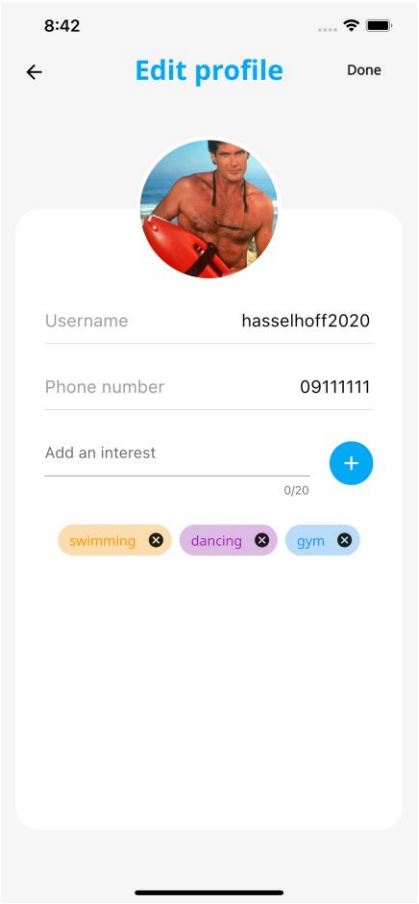

8:27
Search interests
pamela1010
baywatching
swimming
Send request
Search
Requests
Connected
Profile


8:35
Requests from 1 users
hasselhoff2020
swimming
dancing
gym
Search
Requests
Connected
Profile


8:37
Connected with 1 users
pamela1010
Phone: +385913942722
baywatching
swimming
Search
Requests
Connected
Profile

Na zadnjoj kartici profila, korisniku je omogućen pregled profila, izmjena fotografije te uređivanje interesa.





Integracija mobilne aplikacije s ostatkom sustava

Aplikacija je povezana s ostatkom sustava preko zajedničkog REST API-a. Unutar aplikacije napisani su razredi koji su odgovorni za slanje HTTP zahtjeva, serijalizaciju i deserijalizaciju pristiglih i poslanih objekata ili JSON-a. Primjer jednog takvog razreda prikazan je na slici ispod. IDioProvider je sučelje koje definira metodu za dohvat baznog URL-a, te dohvat Dio objekta koji omogućuje lako slanje HTTP zahtjeva. Konkretna implementacija IDioProvidera je u svakom sličnom razredu “injectana” pomoću dependency injectiona. Nad Dio objektom moguće je pozvati request metodu s parametrima za bazni URL, HTTP metodu, Body itd. Ako dođe do greške prilikom izvođenja HTTP zahtjeva, Dio baca DioError. Kada request metoda vrati rezultat, on biva deserijaliziran kako bi se s njime lakše radilo unutar aplikacije.

```dart
@LazySingleton(as: IPeopleDataSource)
class PeopleDataSource implements IPeopleDataSource {
  final IDioProvider _dioProvider;

  PeopleDataSource(this._dioProvider);

  @override
  Future<List<User>> getAllUsers() async {
    final result = await _dioProvider.dio.request<List<dynamic>>(
      '/auth/getAllUsers',
      options: RequestOptions(
        method: 'GET',
        baseUrl: _dioProvider.baseUrl,
      ),
    );
    final users =
        result.data.map((apiUser) => User.fromJson(apiUser as Map<String, dynamic>)).toList();
    return users;
  }
}
```

Primjer sa slanjem serijaliziranog objekta unutar Bodya. Sve se izvodi na sličan način osim što je metoda ovoga puta POST, te se šalje objekt tipa ConnectRequest čija je implementacija dana u nastavku.

```dart
@LazySingleton(as: IRequestDataSource)
class RequestDataSource implements IRequestDataSource {
  final IDioProvider _dioProvider;

  RequestDataSource(this._dioProvider);

  @override
  Future<Request> sendRequest(ConnectRequest request) async
  {
    final result = await _dioProvider.dio.request<Map<String, dynamic>>(
      '/requests/sendRequest',
      options: RequestOptions(
        method: 'POST',
        baseUrl: _dioProvider.baseUrl,
      ),
      data: request.toJson,
    );
    return Request.fromJson(result.data);
  }
```

```dart
class ConnectRequest extends Equatable {
  final int senderId;
  final int receiverId;

  const ConnectRequest({
    this.senderId,
    this.receiverId,
  });

  Map<String, dynamic> get toJson => {
        'senderId': senderId,
        'receiverId': receiverId,
      };

  @override
  List<Object> get props => [senderId, receiverId];
}
```