Univerzitet u Beogradu

Fakultet organizacionih nauka

Katedra za elektronsko poslovanje

Laravel

Domaći zadatak 1

|  |  |  |
| --- | --- | --- |
| Ime | Prezime | Broj indeksa |
|  |  |  |
|  |  |  |
| Link ka Github-u |  | |

Sadržaj

[1 Korisnički zahtev 3](#_Toc201143769)

[1.1 Opis aplikacije 3](#_Toc201143770)

[1.2 Slučajevi korišćenja 4](#_Toc201143771)

[SK: Kreiranje aktivnosti 7](#_Toc201143772)

[SK: Pregled aktivnosti 9](#_Toc201143773)

[SK: Ažuriranje aktivnosti 11](#_Toc201143774)

[SK: Brisanje aktivnosti 13](#_Toc201143775)

[2 Implementacija 16](#_Toc201143776)

[2.1 Opis modela 16](#_Toc201143777)

[2.2 Opis migracija 16](#_Toc201143778)

[2.3 Opis kontrolera 21](#_Toc201143779)

[2.4 Resource 29](#_Toc201143780)

[2.5 Policy 30](#_Toc201143781)

[2.6 Form Request 31](#_Toc201143782)

[3 Rutiranje 33](#_Toc201143783)

[4 Notifikacije 34](#_Toc201143784)

[5 REST API 38](#_Toc201143785)

[6 Korisničko uputstvo 43](#_Toc201143786)

# Korisnički zahtev

## Opis aplikacije

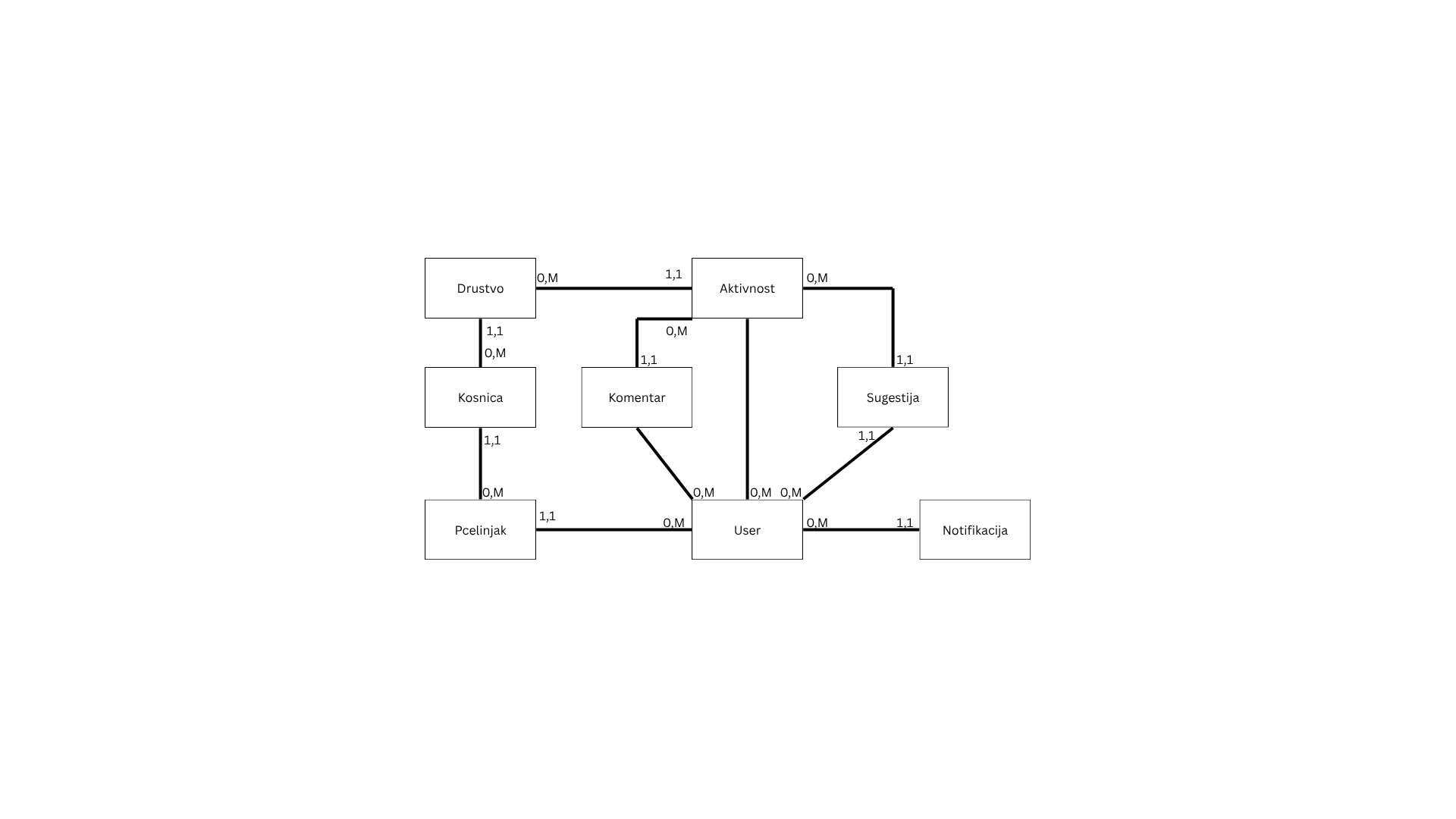
Ova aplikacija predstavlja sveobuhvatan alat za upravljanje pčelinjacima, košnicama i pčelinjim društvima. Korisnici, odnosno pčelari, mogu lako pratiti svoje pčelinjake i detaljno evidentirati svaku košnicu i društvo koje u njima živi. Aplikacija omogućava beleženje ključnih podataka kao što su starost matice, jačina društva i datum formiranja društva, što pomaže u praćenju zdravlja i produktivnosti pčela.

Kroz sistem aktivnosti korisnici planiraju, prate i dokumentuju različite radove i intervencije u pčelinjacima — kao što su pregledi, hranjenje, tretmani protiv bolesti i druge važne zadatke. Uz to, korisnici mogu dodavati komentare i sugestije vezane za svaku aktivnost, što olakšava komunikaciju i praćenje zapažanja i preporuka tokom vremena.

Aplikacija takođe šalje notifikacije i podsetnike kako bi pčelari bili pravovremeno obavešteni o važnim događajima ili obavezama. Na taj način se povećava efikasnost upravljanja i smanjuje mogućnost propusta u negovanju pčelinjih društava.

Ovaj sistem je namenjen da pomogne pčelarima svih nivoa da na jednostavan i organizovan način vode evidenciju i unaprede brigu o svojim pčelama, što može doprineti boljoj produktivnosti i zdravlju pčelinjaka.

Za potrebe kreiranja ove aplikacije fomulisan se konceptualni model, odnosno prošireni model objekti-veze, prikazan na slici ispod.



U nastavnu su objašnjeni svi koncepti modela:

1. User – predtsvalja korisnika sistema, tj. vlasnika pčelinjaka;
2. Pčelinjak – mesto gde se nalaze košnice, odnosno fizička lokacija koju poseduje korisnik;
3. Košnica – pojedinačna kutija, odnosno stanište u pčelinjaku u kojoj živi pčelinje društvo;
4. Društvo – odnosi se na pčelinje društvo, tj. zajednicu pčela koje nastanjuju košnicu, pri čemu svaka košnica može imati jedno ili više društava;
5. Aktivnost – predstavlja aktivnost vezanu za pčelinje drušvo, kao što je pregled košnice, hranjenje pčela, tretman protiv bolesti i slično;
6. Komentar – komentar koje korisnik ostavlja vezano za neku aktivnost, kao što su zapažanja, beleške i slično;
7. Sugestija – predlog, ideja ili preporuka vezana za neku aktivnost;
8. Notifikacija – obaveštenja koja korisnik može da prima, kao što su podsetnici za aktivnosti i slično.

Na osnovu konceptualnog modela moguće je kreirati relacioni model koji će kasnije poslužiti za projektovanje baze podataka. U nastavku je dat relacioni model.

* User(id, name, email, password)
* Pcelinjak(id, naziv, lokacija, user\_id)
* Kosnica(id, oznaka, tip, status, pcelinjak\_id)
* Drustvo(id, kosnica\_id, matica\_starost, jacina, datum\_formiranja)
* Aktivnost(id, naziv, opis, tip, pocetak, kraj, status, drustvo\_id, user\_id, notifikacija\_poslata)
* Komentar(id, sadrzaj, datum, aktivnost\_id, user\_id)
* Sugestija(id, poruka, datum\_kreiranja, user\_id, aktivnost\_id)
* Notifikacija(id, type, notifiable\_type, notifiable\_id , data, read\_at)

## Slučajevi korišćenja

Prilikom izrade aplikacije identifikovani slučajevi korišćenja prikazani u tabeli.

|  |  |  |  |
| --- | --- | --- | --- |
| Redni broj | Naziv | Preduslovi | Kriterijum pretrage |
| 1 | Registracija korisnika |  |  |
| 2 | Pregled korisnika | Ulogovani korisnik može pregledati svoje podatke. |  |
| 3 | Prijava korisnika |  |  |
| 4 | Ažuriranje korisnika | Ulogovani korisnik može izmeniti svoje podatke. |  |
| 5 | Kreranje pčelinjaka | Ulogovani korisnik može izvršiti kreiranje. |  |
| 6 | Pregled pčelinjaka | Ulogovani korisnik može pregledati svoje pčelinjake. | Lokacija |
| 7 | Brisanje pčelinjaka | Ulogovani korisnik može obrisati svoje pčelinjake. |  |
| 8 | Ažuriranje pčelinjaka | Ulogovani korisnik može izmeniti svoje pčelinjake. |  |
| 9 | Kreiranje košnice | Ulogovani korisnik može izvršiti kreiranje. |  |
| 10 | Pregled košnica | Ulogovani korisnik može pregledati svoje košnice. | Tip, status, pcelinjak |
| 11 | Ažuriranje košnice | Ulogovani korisnik može izmeniti svoje košnice. |  |
| 12 | Brisanje košnice | Ulogovani korisnik može obrisati svoje košnice. |  |
| 13 | Kreiranje društva | Ulogovani korisnik može izvršiti kreiranje. |  |
| 14 | Pregled društva | Ulogovani korisnik može pregledati svoja društva. |  |
| 15 | Ažuriranje društva | Ulogovani korisnik može izmeniti svoja društva. |  |
| 16 | Brisanje društva | Ulogovani korisnik može obrisati svoja društva. |  |
| 17 | Kreiranje aktivnosti | Ulogovani korisnik može izvršiti kreiranje. |  |
| 18 | Pregled aktivnosti | Ulogovani korisnik može pregledati svoje aktivnosti. | Tip, status, pocetak, drustvo |
| 19 | Ažuriranje aktivnosti | Ulogovani korisnik može izmeniti svoje aktivnosti. |  |
| 20 | Brisanje aktivnosti | Ulogovani korisnik može obrisati svoje aktivnosti. |  |
| 21 | Kreiranje komentara | Ulogovani korisnik može izvršiti kreiranje. |  |
| 22 | Pregled komenrara | Ulogovani korisnik može pregledati svoje komentare. |  |
| 23 | Brisanje komentara | Ulogovani korisnik može obrisati svoje komentare. |  |
| 24 | Kreiranje sugestija | Ulogovani korisnik može izvršiti kreiranje. |  |
| 25 | Pregled sugestija | Ulogovani korisnik može pregledati svoje sugestije. |  |
| 26 | Ažuriranje sugestija | Ulogovani korisnik može izmeniti svoje sugestije. |  |
| 27 | Brisanje sugestija | Ulogovani korisnik može obrisati svoje sugestije. |  |

SK: Kreiranje aktivnosti

Naziv SK:  
Kreiranje aktivnosti

Aktori SK:  
Korisnik (User)

Učesnici SK:  
Korisnik i sistem

Preduslov:  
Sistem je uključen, a korisnik je uspešno ulogovan i ima pravo pristupa društvu kojem aktivnost pripada. Sistem prikazuje formu za kreiranje aktivnosti.

Osnovni scenario SK:

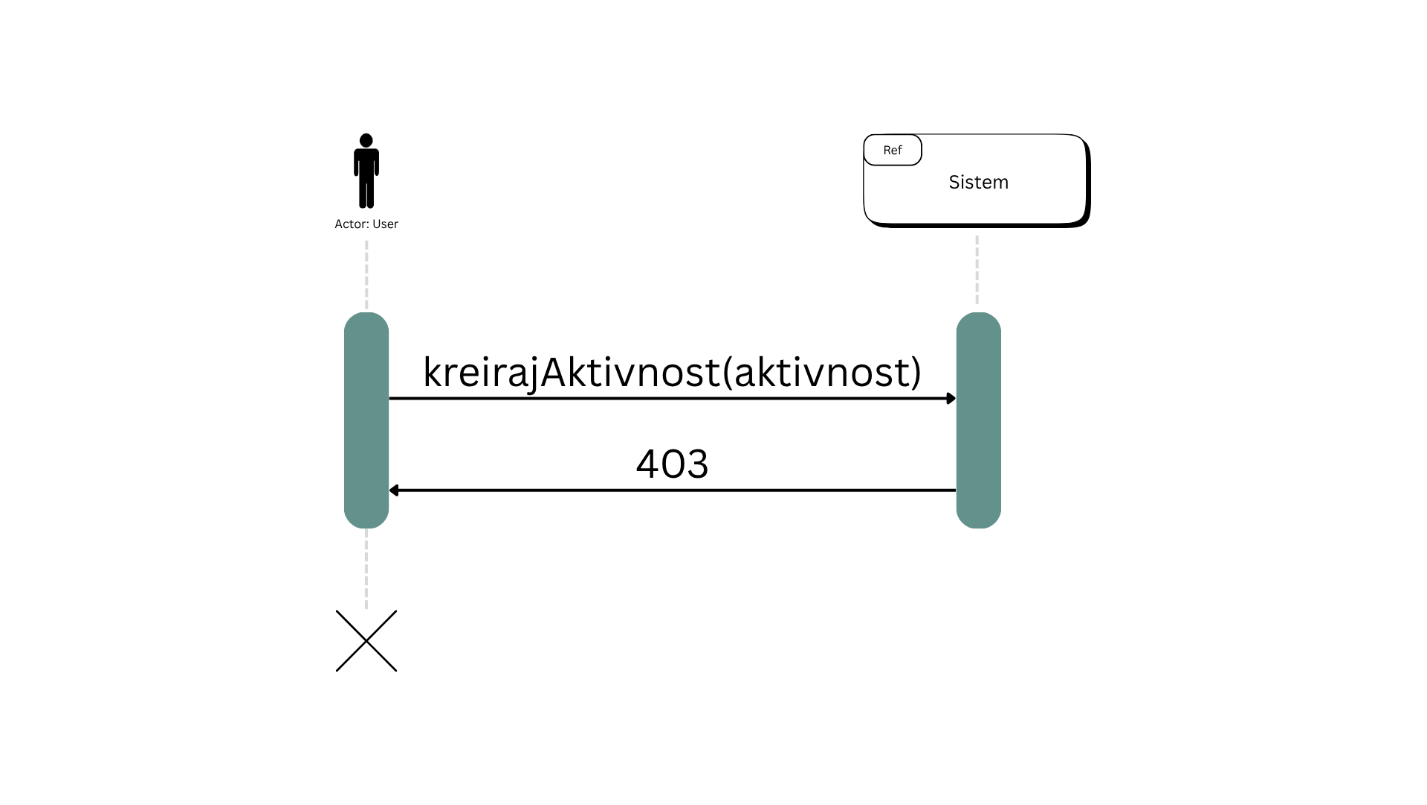
1. Korisnik unosi podatke o aktivnosti: naziv, opis, tip, početak i kraj, status i društvo kome aktivnost pripada. (APUSO)
2. Korisnik proverava da li su podaci ispravno uneti i po potrebi ih koriguje. (ANSO)
3. Korisnik šalje zahtev sistemu da sačuva novu aktivnost. (APSO)
4. Sistem validira podatke i pamti aktivnost u bazi. (SO)
5. Sistem prikazuje korisniku poruku: “Aktivnost uspešno kreirana” (IA)



Alternativni scenariji:  
3.1 Ukoliko sistem utvrdi da nisu popunjena sva obavezna polja ili da su podaci neispravni, prikazuje prilagođenu poruku. Prekida se izvršenje scenarija. (IA)



3.2 Ukoliko korisnik nema pravo da kreira aktivnost za dato društvo, sistem prikazuje 403 grešku i prekida izvršenje. (IA)



SK: Pregled aktivnosti

Naziv SK:  
Pregled aktivnost

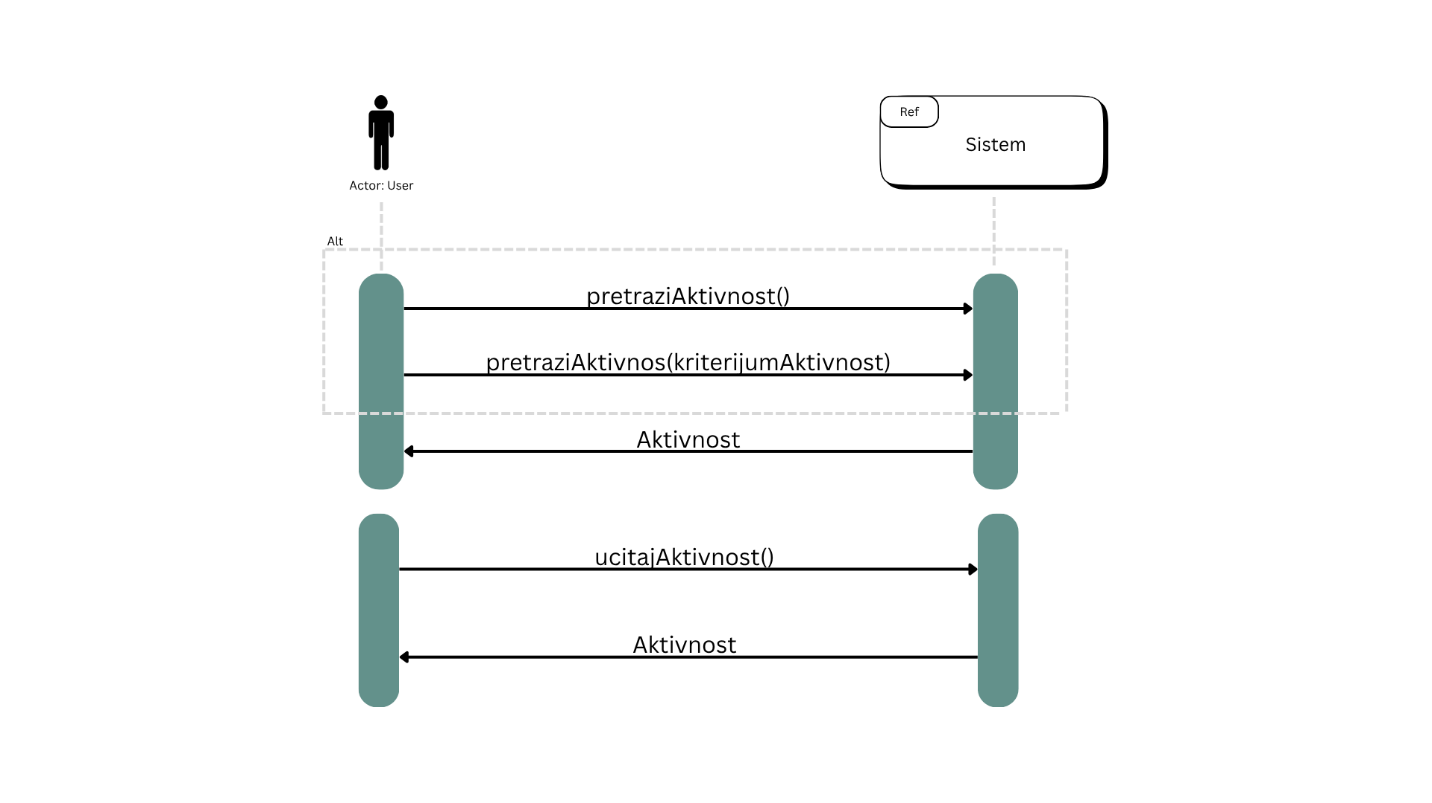
Aktori SK:  
Korisnik (User)

Učesnici SK:  
Korisnik i sistem

Preduslov:  
Sistem je uključen, korisnik je ulogovan i ima pravo pristupa aktivnostima. Sistem prikazuje formu za rad sa aktivnostima.

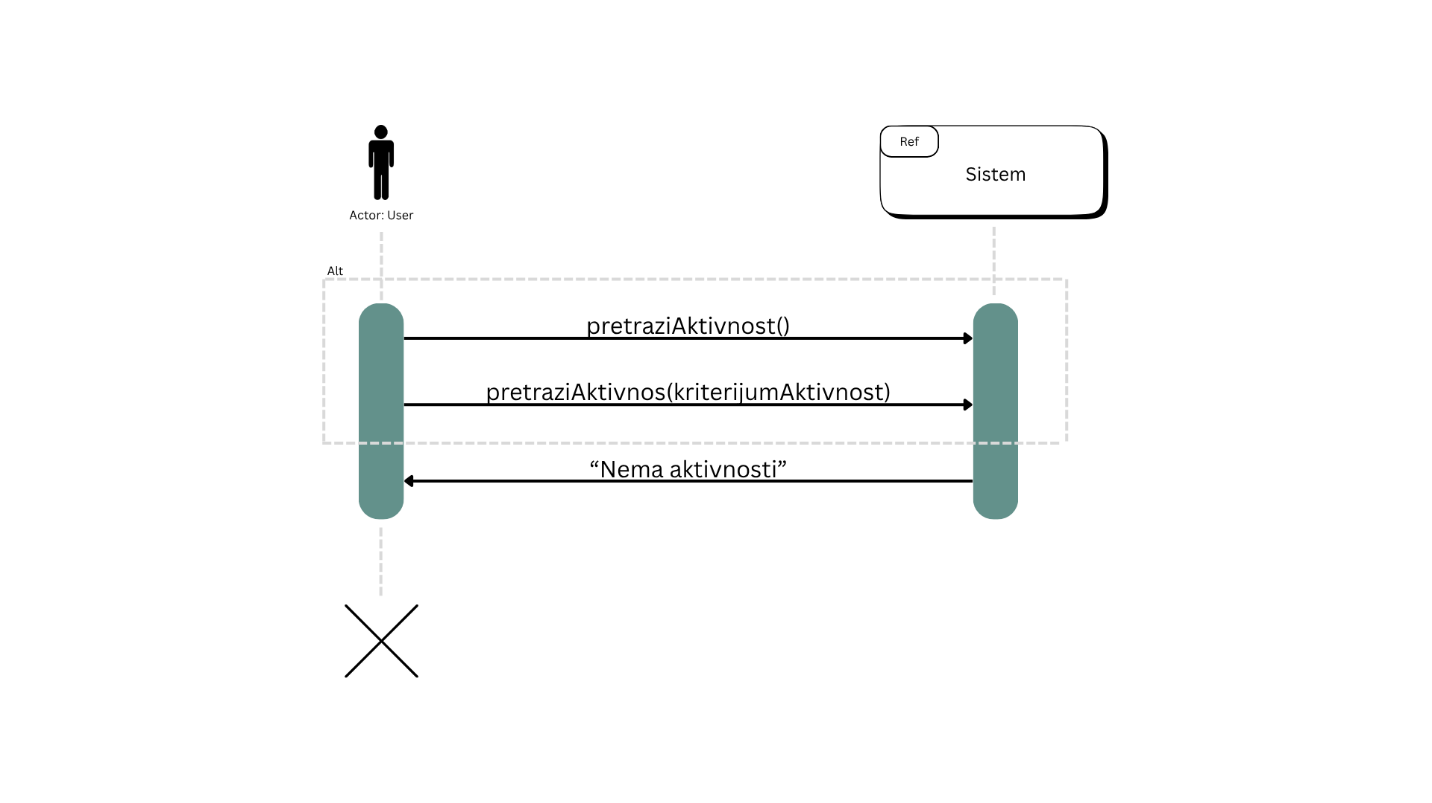
Osnovni scenario SK:

1. Korisnik unosi vrednost (kriterijum) po kojoj pretražuje aktivnosti, npr. naziv, status, datum početka ili kraja. (APUSO)
2. Korisnik poziva sistem da pronađe aktivnosti po zadatoj vrednosti. (APSO)
3. Sistem pretražuje aktivnosti u bazi na osnovu unetog kriterijuma. (SO)
4. Sistem prikazuje korisniku listu pronađenih aktivnosti. (IA)
5. Korisnik bira željenu aktivnost sa prikazane liste. (APUSO)
6. Korisnik traži od sistema da učita detalje odabrane aktivnosti. (APSO)
7. Sistem učitava podatke o odabranoj aktivnosti. (SO)
8. Sistem prikazuje korisniku detaljne podatke o aktivnosti. (IA)



Alternativni scenariji:

4.1 Ukoliko sistem ne može da pronađe aktivnosti po zadatom kriterijumu, prikazuje poruku: “Nema aktivnosti!”. Prekida se izvršenje scenarija. (IA)



SK: Ažuriranje aktivnosti

Naziv SK:  
Ažuriranje aktivnosti

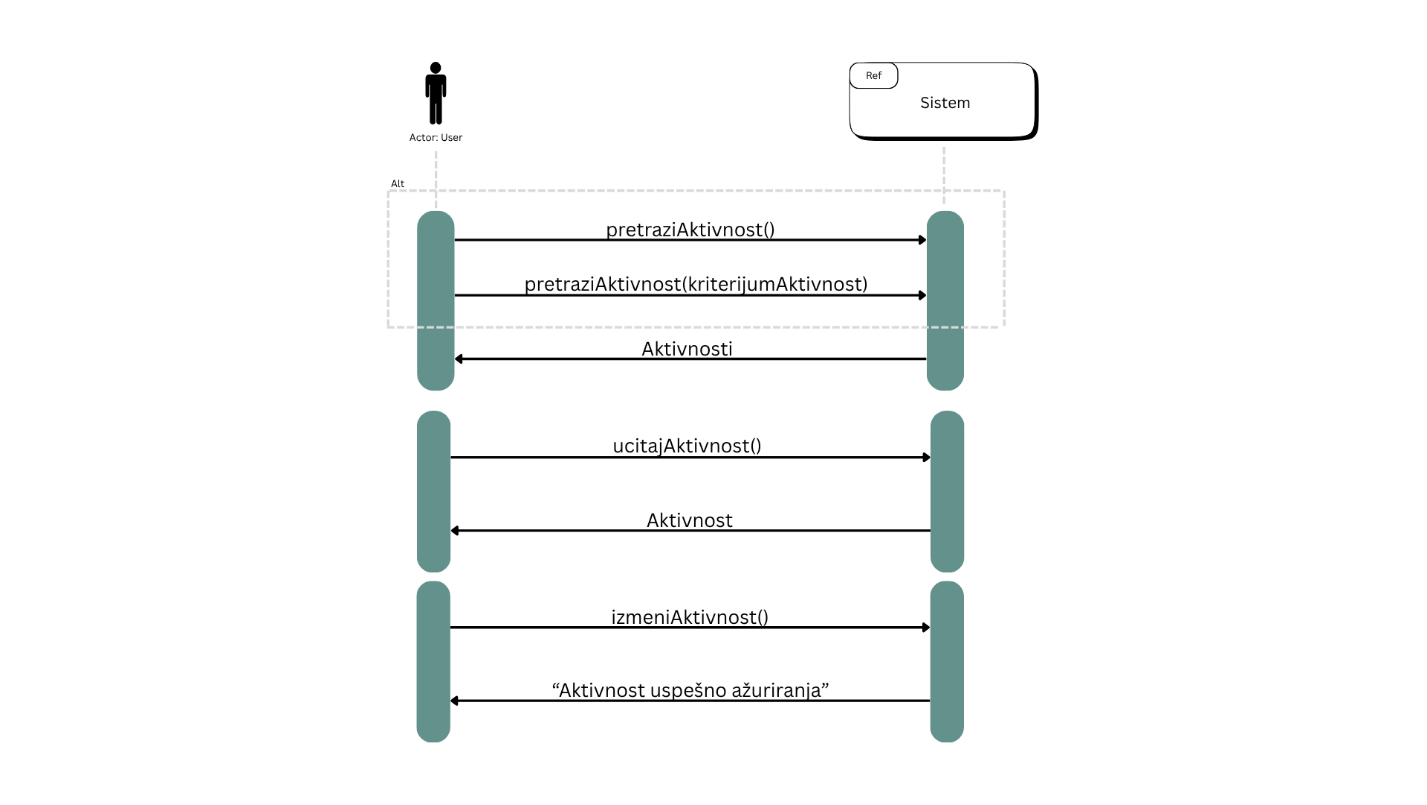
Aktori SK:  
Korisnik (User)

Učesnici SK:  
Korisnik i sistem

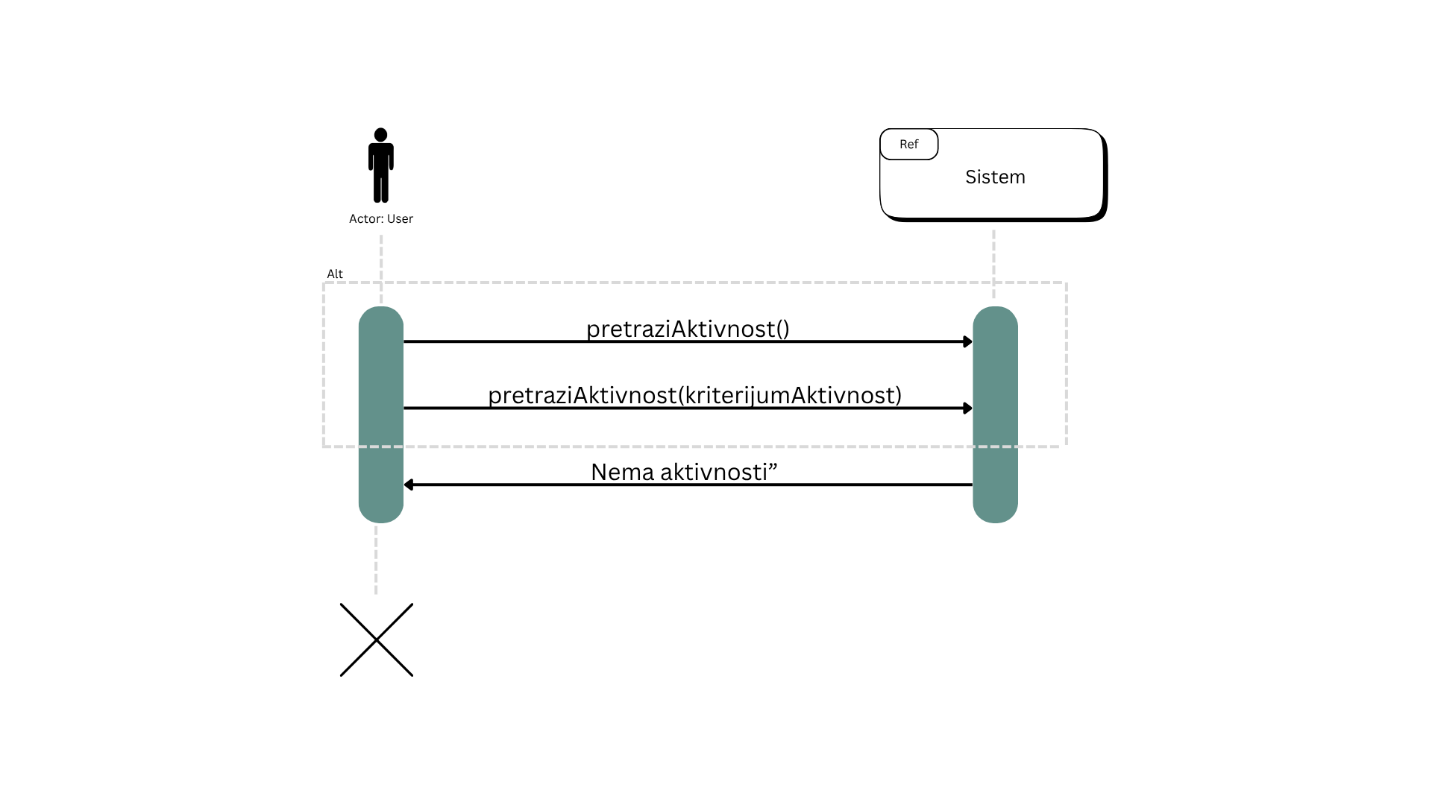
Preduslov:  
Sistem je uključen, korisnik je ulogovan i ima pravo pristupa aktivnostima. Sistem prikazuje formu za rad sa aktivnostima.

Osnovni scenario SK:

1. Korisnik unosi kriterijum za pretragu aktivnosti (npr. naziv, status, datum). (APUSO)
2. Korisnik poziva sistem da pronađe aktivnosti prema unetom kriterijumu. (APSO)
3. Sistem pretražuje bazu aktivnosti po zadatom kriterijumu. (SO)
4. Sistem prikazuje korisniku listu pronađenih aktivnosti. (IA)
5. Korisnik bira jednu aktivnost sa liste. (APUSO)
6. Korisnik traži od sistema da učita detalje odabrane aktivnosti. (APSO)
7. Sistem učitava podatke o aktivnosti iz baze. (SO)
8. Sistem prikazuje korisniku detalje o odabranoj aktivnosti. (IA)
9. Korisnik unosi izmene u podatke aktivnosti (npr. naziv, opis, status, datume). (APUSO)
10. Korisnik proverava da li su izmene ispravno unešene. (ANSO)
11. Korisnik šalje zahtev sistemu da sačuva izmene. (APSO)
12. Sistem ažurira podatke aktivnosti u bazi. (SO)
13. Sistem prikazuje korisniku poruku: “Aktivnost uspešno ažurirana!”. (IA)



4.1 Ukoliko sistem ne može da pronađe aktivnosti, prikazuje poruku: “Nema aktivnosti!”. Prekida se izvršenje scenarija. (IA)



SK: Brisanje aktivnosti

Naziv SK:  
Brisanje aktivnosti

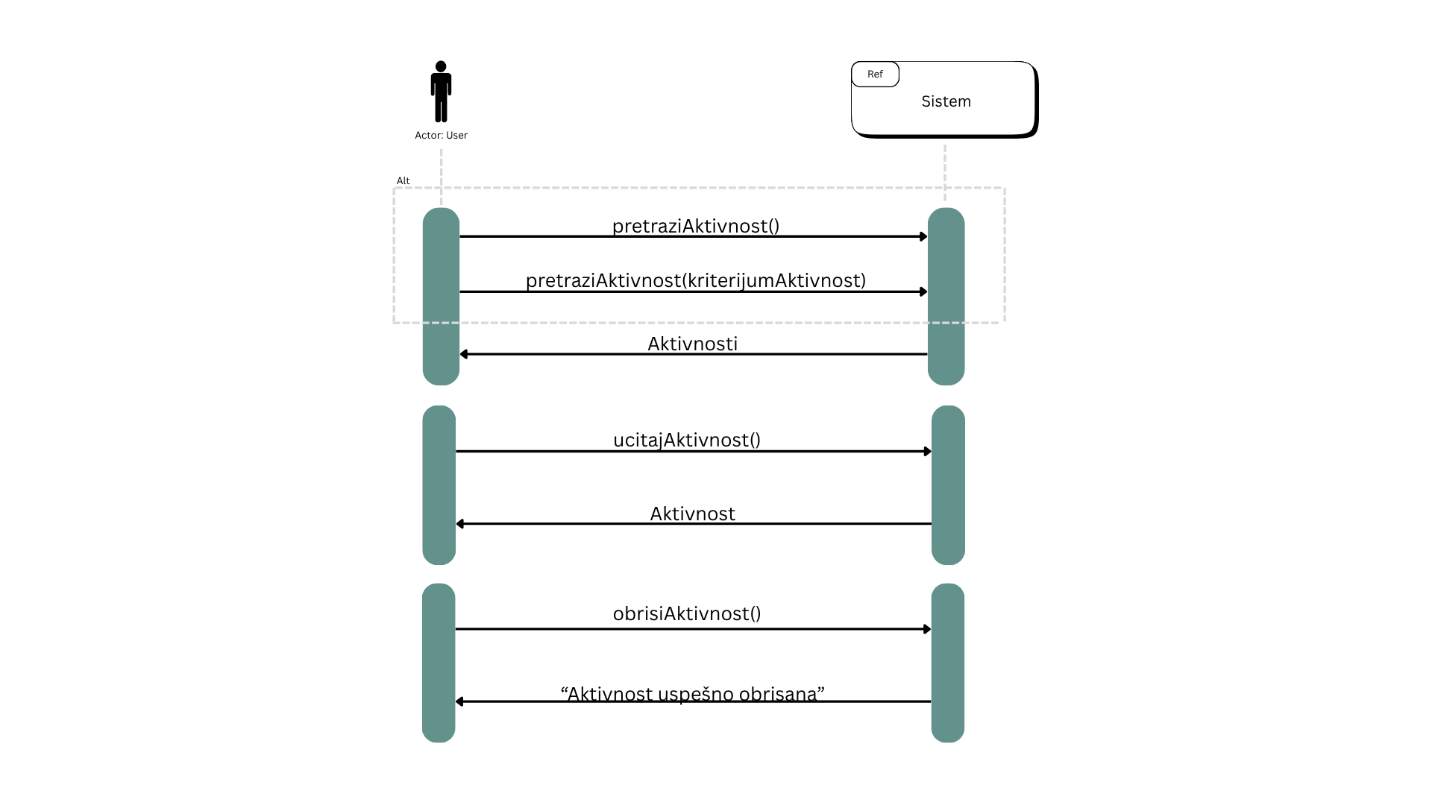
Aktori SK:  
Korisnik (User)

Učesnici SK:  
Korisnik i sistem

Preduslov:  
Sistem je uključen, korisnik je ulogovan i ima pravo pristupa aktivnostima. Sistem prikazuje formu za rad sa aktivnostima.

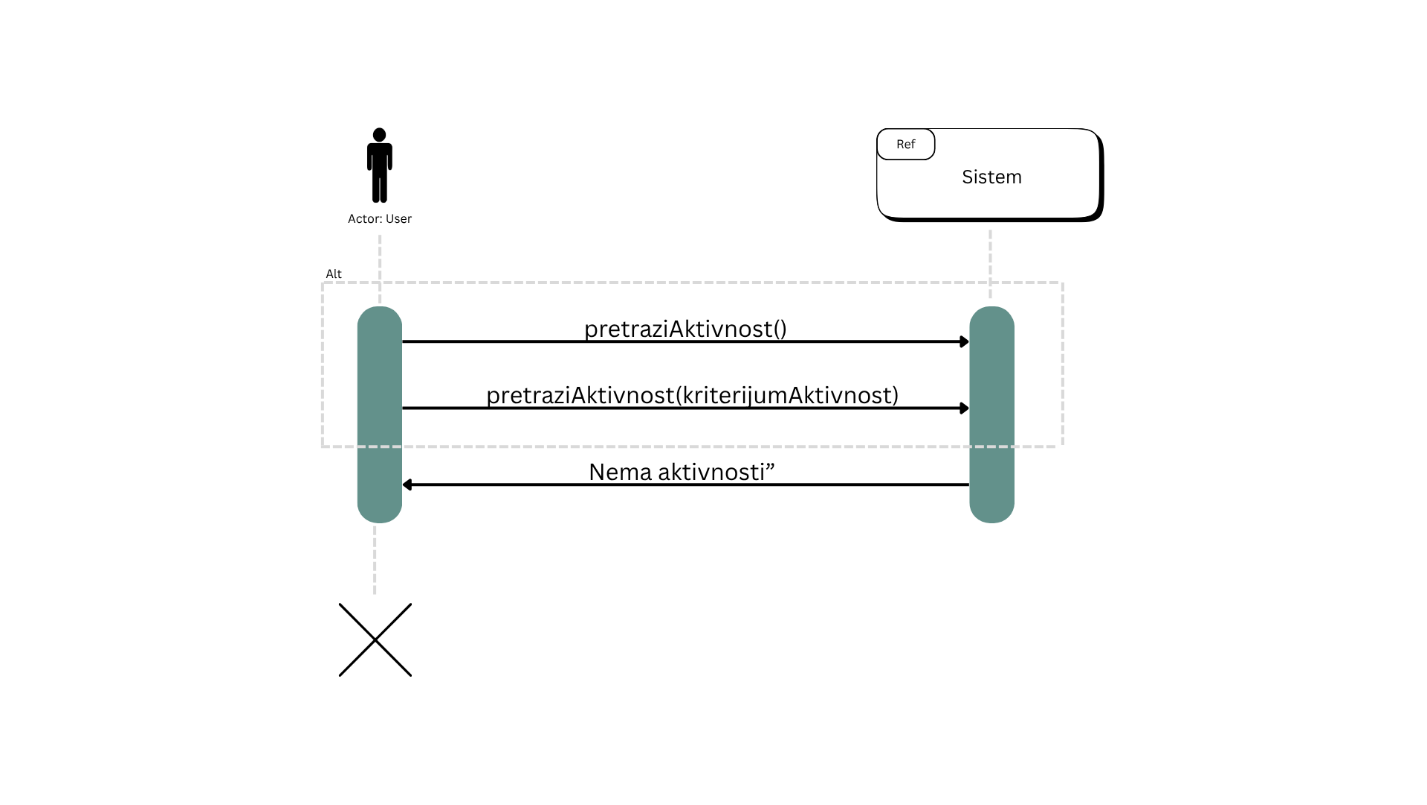
Osnovni scenario SK:

1. Korisnik bira kriterijum za pretragu aktivnosti (npr. naziv, status, datum). (APUSO)
2. Korisnik poziva sistem da pronađe aktivnosti prema unetom kriterijumu. (APSO)
3. Sistem traži aktivnosti po zadatoj vrednosti. (SO)
4. Sistem prikazuje korisniku listu pronađenih aktivnosti. (IA)
5. Korisnik bira jednu aktivnost sa liste. (APUSO)
6. Korisnik poziva sistem da učita podatke o odabranoj aktivnosti. (APSO)
7. Sistem učitava podatke o aktivnosti. (SO)
8. Sistem prikazuje korisniku podatke o aktivnosti. (IA)
9. Korisnik poziva sistem da obriše aktivnost. (APSO)
10. Sistem briše aktivnost zajedno sa povezanim komentarima i sugestijama (kaskadno). (SO)
11. Sistem prikazuje korisniku poruku: “Aktivnost uspešno obrisana”. (IA)

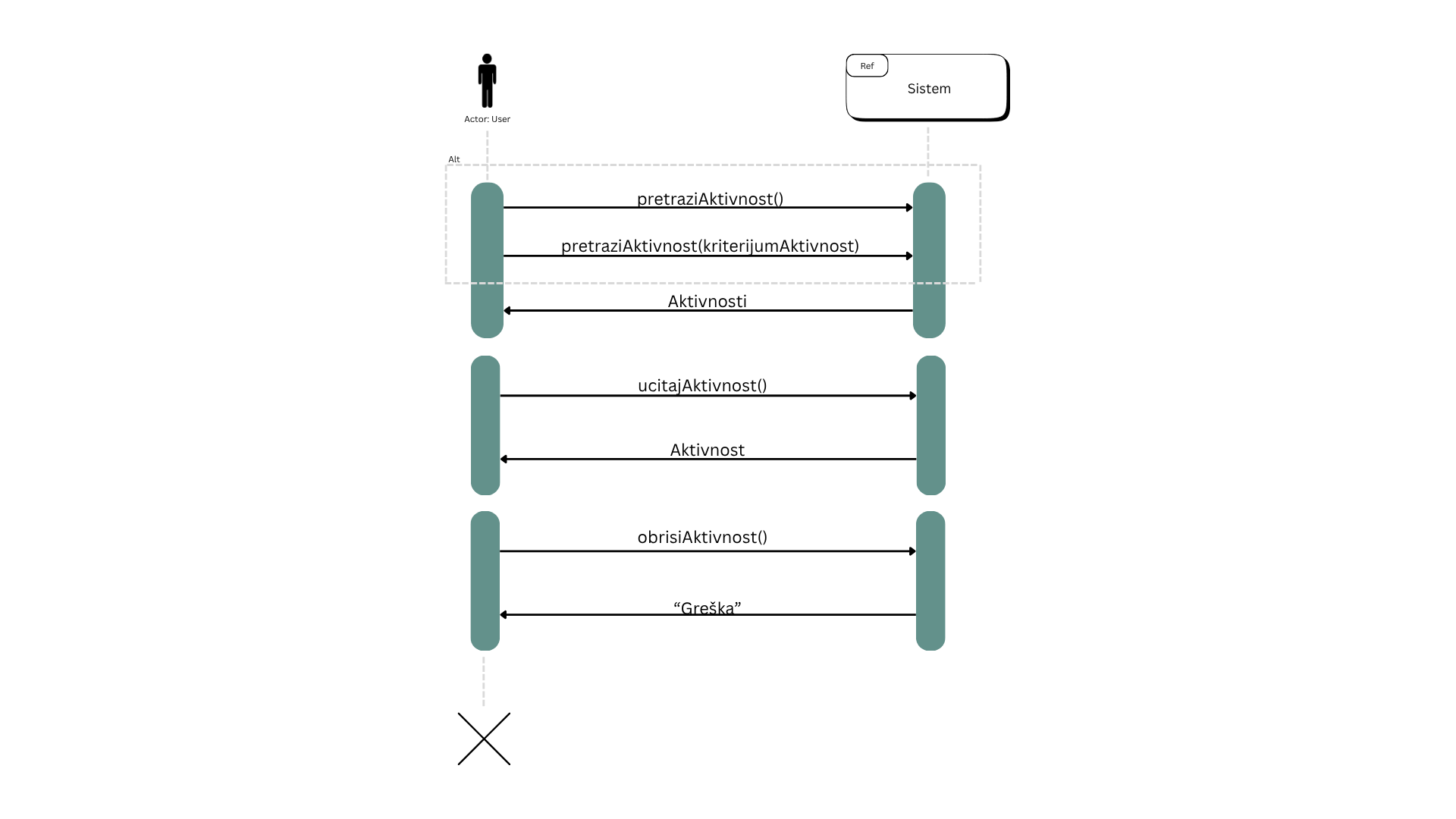


Alternativni scenariji:

4.1 Ukoliko sistem ne može da pronađe aktivnosti, prikazuje poruku: “Nema aktivnosti!”. Prekida se izvršenje scenarija. (IA)



11.1 Ukoliko sistem ne može da obriše aktivnost, prikazuje poruku: “Greška”. Prekida se izvršenje scenarija. (IA)



# Implementacija

## Opis modela

Na osnovu relacionog modela iz prethodnog poglavlja mogu se identifikovati potrebni atributi za svaki entitet, dok PMOV model omogućava uvid u odnose i njihove kardinalnosti.  
Da bismo ove veze prikazali u Laravel okruženju, koristimo Eloquent metode poput hasMany i belongsTo. Kada relacija između dva modela predstavlja odnos jedan prema više (1:0..\*), model koji pripada "više" strani sadržaće belongsTo metodu, dok će model sa "jedan" strane koristiti hasMany.  
Ovakav pristup omogućava Laravelu da automatski prepozna povezanost i strukturu odnosa između entiteta.

U daljem tekstu dati su primeri korišćenja Eloquent relacija, kao i njihova implementacija u samim modelima.

    public function drustvo(): *BelongsTo*

    {

        return *$this*->belongsTo(*Drustvo*::class);

    }

    public function komentars():*HasMany*{

        return *$this*->hasMany(*Komentar*::class);

    }

Kako svaka aktivnost pripada jednom društvu, potrebno je koristiti belongsTo relaciju, dok u slučaju predstavljanja odnosa klase aktivnost i klase komentar primenjuje se hasMany relacija kojom se označava da se za jednu pojavu instance aktivnosti može vezati više konkretnih instanci komentara.

Pomenute relacije definisane su u svim klasama oslikavajući kardinalnosti konceptualnog modela, odnosno PMOV-a.

## Opis migracija

Migracije u Laravelu predstavljaju način za verzionisano upravljanje strukturom baze podataka. Svaka migracija opisuje određenu promenu – poput kreiranja, izmene ili brisanja tabela i njihovih kolona. One omogućavaju da se promene u strukturi baze prate kroz vreme, a takođe olakšavaju timski rad olakšanim deljenjem i sinhronizacijom baze između razvojnih, testnih i produkcionih okruženja.

Kreiranje migracija realizuje se na jedan od dva načina:

1. Kreiranje migracije istovremeno sa modelom, dodavanjem *flega* -m, pri čemu je artisan komanda sledeća: php artisan make:model NazivModela -m;
2. Samostalno kreiranje migracije pozivom sledeće artisan komande: php artisan make:migration add\_deleted\_at\_to\_notifikacijas.

U daljem tekstu prikazan je kod migracije kojom se kreira jedna tabela.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::create('kosnicas', function (*Blueprint* $table) {

            $table->id();

            $table->string('oznaka');

            $table->string('tip');

            $table->enum('status', *Kosnica*::$status);

            $table->foreignIdFor(*Pcelinjak*::class)->constrained()->onDelete('cascade');

            $table->timestamps();

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::dropIfExists('kosnicas');

    }

};

Migracija kao klasa je anonimna, što se može videti na počeku koda. Svaka migracija sadrži dve metode, up i down. U metodi up navode se instrukcije koje će se izvršiti prilikom pokretanja migracija, dok su u metodi down definisane one koje će se izvršiti u slučaju vraćanja migracija. Kao parametar funckije metode up u migracijama za kreiranje tabela postoji instanca tabele ogovarajuće baze podataka. Obzirom da se ove migracije kreiraju za modele, nazivi tabela u bazi podataka nose identičan naziv kao i sami modeli (engleska množina). Nad promenljivom table pozivaju se funkcije koje najčešće nose naziv tipa vrednosti koji će kolona imati, dok su njihovi parametri nazivi kolona. Metodom enum u bazi se kreira kolona čiji će naziv biti isti kao i prvi parametar ove metode, dok će moguće vrednosti biti one koje su zadate u vidu niza kao drugi parametar. Kako bi se konzistentnost održala, definicija niza podignuta je na nivo modela. Metoda foreignIdFor prima metapodatak klase za koju se kreirani spoljni ključ, a na osnovu kog Laravel istoj koloni dodeljuje naziv model\_id (pcelinjak\_id).

Sledeći kod predstavlja migraciju change\_data\_in\_aktivnosts. Migracija u svom nazivu može sadržati naziv relacije na koju se odnosi, a na osnovu koje će je Laravel sam prepoznati.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->date('pocetak')->change();

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->dateTime('pocetak')->change();

        });

    }

};

Zbog jednostavnijeg formata, kolonu koja je prethodno bila definisana kao dateTime, menjamo u date, odnosno menjamo njen tip. Na ovom primeru jasno se vidi razlika u slučaju migracija za kreiranje tabela i migracija koje se odnose na pojedine kolone tabela. Metode up i down su kontradiktorne i zbog toga je važno na odgovarajući način implementirati i metodu down. U ovom slučaju, metoda down vraća tip vrednosti kolone pocetak na inicijalni tip dateTime.

Sledeći deo koda odnosi se na migraciju add\_column\_to\_aktivnosts.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->boolean('notifikacija\_poslata')->default(false);

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->dropColumn('notifikacija\_poslata');

        });

    }

};

Metodom up dodali smo još jednu kolonu koja će označiti da li je notifikacija za neku aktivnost poslata ili nije, kako bismo izbegli višestruke notifikacije. Metodom down novonastala kolona se uklanja.

Sledeći kod odnosi se na migraciju remove\_column\_from\_aktivnosts. Ova migracija prikazuje suprotan slučaj prethodnom.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->dropColumn('trajanje');

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->integer('trajanje');

        });

    }

};

Kolonu trajanje uklanjamo jer smo usled procesa promene pocetak kolone na tip date omogućili jednostavno računanje trajanja aktivnosti bez njenog eksplicitnog pamćenja u bazi podataka.

U sledećem kodu prikazana je migracija add\_foreign\_id\_to\_sugestijas.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::table('sugestijas', function (*Blueprint* $table) {

            $table->foreignIdFor(*Aktivnost*::class)->constrained()->onDelete('cascade');

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::table('sugestijas', function (*Blueprint* $table) {

            $table->dropForeign(['aktivnost\_id']);

            $table->dropColumn('aktivnost\_id');

        });

    }

};

Ovom mirgacijom kreiramo spoljni ključ prema komentarima. Karakteristika migracija u kojima se dodaje spoljni ključ jeste ta da u metodi down, osim običnog uklanjanja kolone spoljnog ključa (dropColumn), moramo ukloniti i vezu između tabela metodom dropForeign.

U sledećem kodu prikazana je migracija add\_constraint\_to\_aktivnosts.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->string('kraj')->nullable()->change();

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::table('aktivnosts', function (*Blueprint* $table) {

            $table->string('kolona\_name')->nullable(false)->change();

        });

    }

};

Ovom migracijom, kraj aktivnosti se označava kao nullable, odnosno kao neobavezno polje. Metodom down uklanjamo ovo ograničenje prosleđujući metodi nullable false vrednost.

## Opis kontrolera

U Laravelu, kontroleri predstavljaju centralnu komponentu unutar MVC (Model-View-Controller) arhitekture. Njihova osnovna uloga je da obrađuju korisničke zahteve, pozivaju odgovarajuće metode iz modela i prosleđuju rezultate ka prikazima (views). Na taj način, kontroleri služe kao posrednici između sloja podataka (model) i korisničkog interfejsa (view), čime doprinose boljoj organizaciji i čitljivosti koda, kao i odvajanju poslovne logike od prezentacije.

U slučaju naše aplikacije, korišćeni su api kontroleri, odnosno kontroleri koji se bave isključivo obradom poslovne logike aplikacije, bez logike prikazivanja sadržaja. API kontroleri u Laravelu idealni su za razvoj backend dela aplikacije jer ne učestvuju u renderovanju prikaza (view-a). Umesto toga, njihov zadatak je da obrađuju zahteve i vraćaju odgovore u JSON formatu, što ih čini posebno pogodnim za rad sa frontend framework-ima poput React-a, kao i za testiranje putem alata kao što je Postman.

Za kreiranje api kontrolera koristi se naredba php artisan make:controller api/NazivModelaController –api. Pri izvršenju ove komande u okviru App\Http\Controllers\api kreira se NazivModelaController.php.

Metode api kontrolera orijentisane su prema CRUD operacijama nad modelima aplikacije. Kako bi se smanjila otperećenost kontrolera, obzirom na važnost njihove uloge u sistemu, korišćene su sledeće klase:

1. Resource – klase koje predtsvljaju sloj između JSON-a i modela, odnosno koje formatiraju odgovor koji server šalje prema klijentu;
2. Policy – klase koje obrađuju logiku autorizacije korisnika;
3. Form Request – klase koje pozivaju pojedinačne metode autorizacije, vrše validaciju i definišu specifične odgovore u slučaju narušavanja postavljenih pravila za validaciju.

U nastavku je prikazan jedan od API kontrolera uz kratka objašnjenja metoda.

class AktivnostController extends *Controller*

{

    public function index()

    {

        $user = *Auth*::user();

        $filteri = request()->only('tip', 'status', 'pocetak', 'drustvo');

        $aktivnosti = *Aktivnost*::with(['user', 'drustvo', 'komentars', 'sugestijas'])

            ->filter($filteri, $user)

            ->latest()

            ->paginate();

        return *AktivnostResource*::collection($aktivnosti);

    }

    public function store(*AktivnostRequest* $request)

    {

        $user = *Auth*::user();

        $data = $request->validated();

        $data['user\_id'] = $user->id;

        $aktivnost = *Aktivnost*::create($data);

        return response()->json([

            'message' => 'Aktivnost uspešno kreirana',

            'model' => new *AktivnostResource*($aktivnost)

        ]);

    }

    public function show(*Aktivnost* $aktivnosti)

    {

        if (*Gate*::allows('view', $aktivnosti)) {

            $aktivnosti->load(['drustvo', 'user', 'komentars', 'sugestijas']);

            return new *AktivnostResource*($aktivnosti);

        } else {

            return response()->json([

                'status' => 'error',

                'message' => 'Nemate dozvolu da pregledate ovu aktivnost.',

            ], 403);

        }

    }

    public function update(*AktivnostRequest* $request, *Aktivnost* $aktivnosti)

    {

        $data = $request->validated();

        $aktivnosti->update($data);

        return response()->json([

            'message' => 'Aktivnost uspešno ažurirana',

            'model' => new *AktivnostResource*($aktivnosti)

        ]);

    }

    public function destroy(*Aktivnost* $aktivnosti)

    {

        if (*Gate*::allows('delete', $aktivnosti)) {

            $aktivnosti->delete();

            return response()->json([

                'message' => 'Aktivnost uspešno obrisana.',

            ]);

        } else {

            return response()->json([

                'status' => 'error',

                'message' => 'Nemate dozvolu da obrišete ovu aktivnost.',

            ], 403);

        }

    }

}

Metoda index kontrolera za aktivnost namenjena je slanju većeg broja aktivnosti koje opciono mogu biti filtrirane. Pored toga, za autentifikovanog korisnika prikazuju se samo aktivnosti koje mu pripadaju. Kako bi se aktivnosti učitale sa potrebnim relacijama, bitno je iste relacije uključiti metodom with koja se poziva nad klasom Aktivnost. Navedeno je posledica Laravelovog mehanizma lazy loadinga, što podrazumeva da Laravel ne učitava relacije modela bez ekplicitnog zahteva (metode). Aktivnosti se sortiraju prema koloni created\_at koja je kreirana metodom timestamp u migracijama i zatim se vrši paginacija. Paginacija je proces podele niza objekata (koji nastaju kao odgovor servera), na nekoliko stranica. Broj odgovora po starni može se navesti kao parametar metode paginate, dok ukoliko se ne navede broj elemenata po strani biće 15. Metodom paginate server šalje i podatke o strani na kojoj su poslati elementi, kao i ukupan broj strana. Ukoliko želimo da vidimo konkretnu stranu, kao parametar zahteva, u URL delu, šaljemo upit page=n, gde je n broj željene strane. Ukoliko ovaj parametar izostane, podrazumevano je da će se prikazati elementi prve strane. Aktivnosti se mogu filtrirati prema parametrima tip, status, pocetak i društvo, koje se navode u URL delu zahteva. Metoda kojom se filtriranje vrši implementirana je u samom modelu, kako bi se održala čitljivost i jasnoća koda. U nastavku je dat deo modela Aktivnost koji obuhvata metodu za filtriranje.

public function scopeFilter(*Builder*|*QueryBuilder* $query, array $filteri, *User* $user): *Builder*|*QueryBuilder*{

        return $query

        ->where('user\_id', $user->id)

        ->when($filteri['pocetak'] ?? null, function ($query, $pocetak){

            $query->whereMonth('pocetak', $pocetak);

        })->when($filteri['tip'] ?? null, function ($query, $tip){

            $query->where('tip', $tip);

        })->when($filteri['status'] ?? null, function ($query, $status){

            $query->where('status', $status);

        })->when($filteri['drustvo'] ?? null, function ($query, $drustvo){

            $query->where('drustvo\_id', $drustvo);

        });

    }

Metoda je Laravelov scope. Scope ili opsezi, su metode koje se koriste da enkapsuliraju logiku upita. Prilikom njihovog definisanja, kao ime metode navodi se scope na početku, a zatim na koji upit se odnosi. Kada se metoda poziva scope se može izostaviti iz naziva metode.

Prednost korišćenja opsega je u tome što parametar query ne mora programer definisati, već ga Laravel na osnovu klase nad kojom se poziva sam kreira. Svakako taj parametar mora postojati u definiciji opsega. U našem slučaju pomenuti query parametar je tipa Builder (Eloquent) ili QueryBuilder (Query). Ključna razlika ovih tipova leži u kompleksnosti upita koje podržavaju, odnosno, dok QueryBuilder podržava samo jednostavne upite, Builder se koristi za takozvane nested upite, ondosno upite za koje je karakterisčno *džojnovanje* (join). Povratna vrednost opsega je ista kao i vrednost prvog parametra.

Drugi parametar opsega jeste niz koji dobijamo iz kontrolera, konkretno iz metode only poznave nad request. Metodom only definišu se ključevi parametra zahteva koji če se uzimati u obzir za filtriranje i njihove vrednosti, dok se samom request metodom pristupa zahtevu. Dakle u kontroleru, promenljiva filteri biće asocijativno niz sa ključevima zadatim u only metodi i vrednostima zadatim u samom URL-u zahteva.

Treći parametar ove metode je korisnik kog dobijamo iz kontrolera pomoću fasade Auth, koja poziva metodu user, čija je povratna vrednost autentifikovan korisnik. Fasade u Laravelu najpribližnije su interfejsima pomoću kojih se mogu pozivati statičke metode, bez potrebe kreiranja instanci konkretnih klasa.

Sama metoda implementirana je tako da se na počeku pomoću metode where query nadograđuje upitom kojim se vrši filtriranje prema tome da li aktivnosti pripadaju prosleđenom korisniku. Nakon navedenog vrši se filtriranje prema pojedinačnim kriterijumima. Metoda when karakteristična je po tomešto na osnovu vrednosti (ili definisanosti) prvof parametra izvršava closure prosleđen kao drugi parametar. Closure u PHP-u su zapravo anonimne fukcije. Osnovna karakteristika ovih funcija jeste što mogu jednostavno koristiti varijable dostupne u njenom okruženju. Na početku svake metode when ispituje se definisanost uređenog para asocijativnog niza filteri čiji je ključ naveden (poceta, tip, status, drustvo). Ukoliko par ne postoji, prvi parametar dovija vrednost null i closure se ne izvršava. U suprotnom uzima se vrednost tog para i poziva closure.

Closure uzima dva parametra svog okruženja, prvo sam query, a zatim vrednost para asocijativnog niza. Redosled ovih parametara je jako bitan. Prvi parametar closure će tražiti iz okruženja, a drugi zatim iz when metode. Iako parametre ne prosleđujemo direktno, closure ih sam *hvata*. Svaki od closur-a nadograđuje query metodom where, zadajući mu vrednsot kolone koja se filtrira i vrednost kojom se fitrira (drugi parametar). Ukoliko se operator poređenja ne navede, podrazumevano je jednakost (=).

Pored toga što vrši paginaciju, metoda paginate, takođe i izvršava upit nad bazom podataka, vraćajući set dobijenih podataka.

Metoda store namenjena je kreiranju jedne instance aktivnosti i njenom beleženju u bazu podataka. Na početku metode inicijalizujemo korisnika koji je ulogovan. U promenljivu data upisujemo rezultat izvršenja metode validated koja se poziva nad parametrom request. Zbog korišćenja Form Request klasa, logika validacije podataka i autorizacije korisnika izmeštena je iz kontrolera. Upsešnom validacijom i autorizacijom, na dobijeni asocijativni niz data, konkatenira (nadovezuje) se još jedan uređeni par sa ključem user\_id i vrednosšću id-a prijavljenog korisnika. Ovakav asocijativni niz sadrži sve podatke potreben za kreirnje instance aktivnosti i njeno pamćenje u bazi. Pomenuto se vrši metodom create koja se poziva nad samom klasom. Ukoliko je metoda uspešno izvršena, server vraća json odgovor, koji sadrži message i model, pri čemu je model prosleđen kao instanca resource klase. O klasama resource i form request biće više reči kasnije.

Metoda show prikazuje aktivnost čiji je id zadat kao parametar zahteva. Kako bismo dodatno pojednostavili ovu metodu, type hint-ovali smo parametar metode. Umesto string $id, prisutan je Aktivnost $aktivnost. Ovaj proces poznat je kao route-model-binding. Naime, kada u parametru zahteva prosledimo id, a primenili smo mehanizam route-model-binding-a Laravel će promenljivoj aktivnost sam dodeliti vrednost koju dobija pozivom upita nad bazom podataka na osnovu primarnog ključa. Ukoliko ovaj mehanizam ne bi bio korišćen, metoda show zahtevala bi još jednu liniju koda u kojoj bismo morali ručno da kreiramo i pozivamo upit nad bazom kojim bismo dobili instancu samog modela.

U metodi se pojavljuje još jedna fasada – Gate. Gate je fasada koja je komplementarna klasi policy o kojoj će biti reči kasnije. Ova fasada poziva metodu allows koja proverava da li prijavljeni korisnik ima privilegije da izvrši pregled konkretne instance. Zbog navedenog, kao parametri se šalju naziv metode za koju se utvrđuje autorizacija i model ili instanca modela na osnovu koje se utvrđuje autorizacija. Korisnik koji je ulogovan biće prosleđen Laravelovim mehanizmima. Ukoliko metoda allows vrati true, za odabranu aktivnosti će se učitati potrebne relacije i ona će se prikazati pomoću resource. U suprotnom, korisnik će biti obavešten da niej autorizovan za izvršenje akcije pregledanja aktivnosti.

U metodi update takođe je implementiran mehanizam route-model-binding-a. Nakon validacije podataka i autorizacije korisnika metodom validated, pomoću nestatičke metode update nad instancom modela vrši se ažuriranje koje odmah biva zapamćeno u bazi podataka.

Metodom destroy vrši se brisanje n-torke iz baze podataka. I u ovoj metodi uočljiv je route-model-binding. Brisanje se vrši pomoću metode delete nad instancom objekta.

Pored api kontrolera postoje i obični kontroleri i resource kontroleri. U našoj aplikaciji implementiran je pored api i običan kontroler. Uloga ovog kontrolera jeste autentifikacija korisnika. Sledeći kod je kod AuthController-a.

class AuthController extends *Controller*

{

    public function login(*Request* $request)

    {

        $request->validate([

            'email' => 'required|email',

            'password' => 'required'

        ]);

        $user = *\App\Models\User*::where('email', $request->email)->first();

        if (!$user) {

            throw *ValidationException*::withMessages([

                'email' => ['The provided credentials are incorrect.']

            ]);

        }

        if (!*Hash*::check($request->password, $user->password)) {

            throw *ValidationException*::withMessages([

                'email' => ['The provided credentials are incorrect.']

            ]);

        }

        $token = $user->createToken('api-token')->plainTextToken;

        return response()->json([

            'id' => $user->id,

            'token' => $token,

            'name' => $user->name,

            'email' => $user->email,

        ]);

    }

    public function logout(*Request* $request)

    {

        $request->user()->tokens()->delete();

        return response()->json([

            'message' => 'Logged out successfully'

        ]);

    }

    public function register(*Request* $request)

    {

        $validatedUser = $request->validate([

            'name' => 'required|string|max:20',

            'email' => 'required|email|unique:users',

            'password' => 'required|string|min:8'

        ]);

        $user = *User*::create($validatedUser);

        return response()->json([

            'message' => 'Account created',

            'user' => new *UserResource*($user)

        ]);

    }

}

Registracija korisnika podrazumeva samo njegovo pamćenje u bazi pdoataka.

Metoda za prijavu korisnika definiše logiku prijave na sistem. Za samo autentifiakciju korišćen je paket Sanctum. Nakon njegove instalacije u modelu User uključen je trait HasApiTokens koji se omogućava kreiranje api tokena, a zatim i pokrenute migracije kojima se kreira tabela personal\_access\_tokens kojom se evidentira prijava korisnika. Za prijavu korisnika neophodni su kredencijali. Prosleđeni kredencijali se validiraju metodom validate koja prima asocijativni niz čiji su ključevi isti kao i ključevi zahteva, dok su vrednosti validaciona pravila odvojena uspravnim crtama koja moraju biti zadovoljena za svaku vrednost poslatog zahteva. Ukoliko su validaciona pravila ispoštovana, pronalazi se korisnik sa prosleđenom email adresom u bazi podataka. Ukoliko takav korisnik ne postoji vrednost varijable user biće null. Ako je prosleđena adresa nepostojeća, dolazi do ValidationException izuzetka. U suprotnom se za pronašenog korisnika proverava lozinka. Ukoliko lozinka nije ispravna dolazi do istog izuzetka, dok u suprotnom za korisnika se kreira api-token metodom createToken, koja je pristuna u uključenom trait-u User modela. Kao odgovor server šalje nekoliko podataka o korisniku, ali i generisani token koji će se zatim koristiti za autentifikaciju korsinika.

Metodom za odjavljivanje korisnika, kreirani token u tabeli personal\_access\_tokens se uklanja.

Laravel utvrđuje autentifikovanost kroisnika pronalazeći prosleđen api-token u autorizacionom delu zahteva u personal\_access\_tokens tabeli. Ukoliko takav token postoji korisnik je autentifikovan i dalje se mogu vršiti operacije ukoliko je autorizovan, u suprotnom Laravel obaveštava o tome da korisnik nije autentifikovan.

Na osnovu zahteva specifičnih za temu aplikacije kreirali smo još dva kontrolera. Jedan koji se odnosi na notifikacije, a biće objašnjen u delu o notifikacijama, i jedan za preuzimanje komentara u vidu word dokumenta. Kontroler za preuzimanje dokumenta dat je u nastavku.

class KomentarDownload extends *Controller*

{

    public function pogled()

    {

        return view('components.button');

    }

    public function generateKomentarDoc($id)

    {

        $komentar = *Komentar*::with(['user'])->findOrFail($id);

        $phpWord = new *PhpWord*();

        $section = $phpWord->addSection();

        $section->addText('Datum: ' . $komentar->datum, [

            'name' => 'Arial',

            'size' => 12,

        ]);

        $section->addTextBreak(1);

        $section->addText('Autor: ' . $komentar->user->name, [

            'name' => 'Arial',

            'size' => 12,

            'bold' => true,

        ]);

        $section->addTextBreak(1);

        $section->addText($komentar->sadrzaj, [

            'name' => 'Arial',

            'size' => 12,

        ]);

        $filename = 'Komentar\_' . $komentar->id . '.docx';

        $filepath = storage\_path($filename);

        $phpWord->save($filepath);

        return response()->download($filepath, $filename)->deleteFileAfterSend(true);

    }

}

Ovaj kontroler, iako nije resource, u svrhu testiranja obradiće i jedan pogled. Pogled je jednostavan i sadrži samo a tag kojim se dokuemtn preuzima. Kako bismo kreirali pogled, odnosno samo komponentu koja nam je potrebna koristimo naredbu php artisan make:component Button –view. Flaf view koristimo kako bismo izbegli kreiranje kontrolera za sam pogled. U nastavku je dat kod komponente button.

<div>

    <a *href*="/komentari/1/preuzmi" *target*="\_blank">

        Preuzmi komentar

    </a>

</div>

U komponenti se može primetiti da je id komentara koji se preuzima zakucan, tj. ne može se promeniti iz pogleda. Ovo koristimo samo kako bismo izvršili testiranje.

Metoda koja obrađuje samu logiku formiranja dokumenta i njegovog preuzimanja jeste generateKomentarDoc. U ovoj metodi se na početku prvo pronalazi koemntar za prosleđen identifikator. Nakon toga kreira se instanca Phpword klase. Kako bi ova klasa i njene metode bile dostupne, neophodno je preuzeti phpword (phpoffice) paket Laravela. Nad instancom Phpword klase kreiraju se sekcije, na kojima se ispisuje tekst koji se ujedno i formatira. Nakon dodavanja željenih sekcija definiše se naziv fajla koji će biti zapamćen, preuzima podatak o mestu čuvanaj fajla i na kraju sam fajl čuva na računaru korisnika.

## Resource

Resource klase u Laravelu koriste za formatiranje odgovora koje server šalje klijentu. Najčešće se klase prave za odgovoarajuće modele i u tom smislu predstavljaju sloj između JSON-a i samih modela. Za kreiranje ovih klase korisitmo narednu php artisan make:resource NazivResource. U nastavku je prikazan kod iz klase AktivnotResource.

class AktivnostResource extends *JsonResource*

{

    /\*\*

     \* Transform the resource into an array.

     \*

     \* @return array<string, mixed>

     \*/

    public function toArray(*Request* $request): array

    {

        return [

            'id' => *$this*->id,

            'naziv' => *$this*->naziv,

            'opis' => *$this*->opis,

            'tip' => *$this*->tip,

            'pocetak' => *$this*->pocetak,

            'kraj' => *$this*->kraj,

            'status' => *$this*->status,

            'drustvo' => new *DrustvoResource*(*$this*->whenLoaded('drustvo')),

            'user' => new *UserResource*(*$this*->whenLoaded('user')),

            'komentars' => *KomentarResource*::collection(*$this*->whenLoaded('komentars')),

            'sugestijas' => *SugestijaResource*::collection(*$this*->whenLoaded('sugestijas'))

        ];

    }

}

Pomoću ovih klasa moguće je regulisati koji podaci nekog modela će biti prikazani, a koji ne i u kom formatu. Resource klase imaju samo jednu metodu čija je povratna vrednost asocijativni niz. Ključevi asocijativnog niza predstavljaju ključeve odgovora servera, dok su vrednosti, one vrednosti koje će se pojaviti u paru sa ključem. Nazivi ključeva ne moraju biti isti kao i nazivi kolona modela. Vrednosti za svaki ključ postavljaju se pozivom atributa nad instancom same klase. Klasa Resource sadrži podatak o modelu jer se isti proleđuje u konstruktoru klasa prilikom njihove inicijalizacije, kao što smo videli u kontrolerima. Zbog navedenog, jako je važno da nazivi atributa čije vrednosti koristimo butu ispravni. Pored jednostavnih atributa moguće je na osnovu relacija učitati i resource drugih modela. Pa tako ključu društvo dodeljujemo vrednost resource klase modela Drustvo, koja će se prikazati onda kada se relacije učitaju (whenLoaded). Zahvaljujući tome što smo definisali relaciju prema modelu društvo, sada je ovde možemo iskoristiti. Ukoliko je potrebno da sa instancom modela učitamo i instance modela koji je sa njom povezan, koristimo metodu collection klase resource. Ova metoda, takođe kada su relacije učitane, ima povratnu vrednost niza nizova, odnosno vratiće niz instanci povezanog modela, koje su same po sebi asocijativni niz.

## Policy

Policy klase u Laravelu koriste se za određivanje autorizacije korisnika. Ove klase uvek se kreiraju za konkretan model. Kako bismo kreirali ove klase koristimo naredbu php artisan make:policy ModelPolicy –model=Model. Flagom model policy povezujemo sa odgovarajućim modelom. U nastavku je prikazana AktivnostPolicy klasa.

class AktivnostPolicy

{

    public function view(*User* $user, *Aktivnost* $aktivnost): bool

    {

        return $user->id === $aktivnost->user\_id;

    }

    public function create(*User* $user): bool

    {

        return true;

    }

    public function update(*User* $user, *Aktivnost* $aktivnost): bool

    {

        return $user->id === $aktivnost->user\_id;

    }

    public function delete(*User* $user, *Aktivnost* $aktivnost): bool

    {

        return $user->id === $aktivnost->user\_id;

    }

    public function restore(*User* $user, *Aktivnost* $aktivnost): bool

    {

        return false;

    }

Svaka klasa inicijalno ima više metoda, pri čemu je svaka metoda orijentisana prema nekoj od CRUD operacija. Povratna vrednost svake metode je tipa bool, odnosno true ili false. Metode viewAny (koja ovde nije implementirana) i view determinišu da li korisnik koji je ulogovan može pregledati više aktivnosti, odnosno jednu kontretnu aktivnost. Metode create, update i delete determinišu da li prijavljeni korisnik može kreirati, ažurirati i obrisati instancu modela Aktivnost. Metoda restore koristi se u slučaju implementacije mehanizma soft deletes, odnosno mekog brisanja, koji kod nas za sada nije implementiran.

Metoda create implementirana je tako da svakom korisniku koji je autentifikovan, odnosno prijavljen dozvoljava da kreira jednu aktivnost. Logika koja stoji iza zabrane kreiranja aktivnosti za drugog korisnika implementira se u samom kontroleru. Značaj ove metode više se vidi u sistemima kod kojih postoji hijerarshijska struktura korisnika, odnosno kada postoje korisnici sa različitim privilegijama. Ukoliko korisnik nije autentifikovan, on neće biti ni autorizovan, odnosno činjenica da metoda create vraća true, ne znači da svako može da kreira aktivnost, već bilo koji korisnik koji je prethodno ulogovan.

Metode view, update i delete implementiraju istu logiku, odnosno, prijavljeni korisnik može da pogleda jednu aktivnost, da je ažurira ili obriše samo ukoliko je ta aktivnost njegova. Utvrđivanje pripadnosti aktivnosti vrši se poređenjem identifikatora prijavljenog korisnika i identifikatora korisnika u okviru aktivnosti nad kojom se vrši neka od pomenutih operacija.

## Form Request

Form Request klase u Laravelu koriste se za odvajanje logike autorizacije, odnosno primene policy-a, i validacije od kontrolera, odnosno bazične poslovne logike aplikacije. Ove kalse kreiraju se komantom php artisan make:request NazivRequest. Za request klase nije obavezno da se kreiraju za konkrente modele. Ono što je sa druge strane praksa jeste da se kreiraju posebni request-i za update i create metode. Međutim, obzirom na veličinu aplikacije, kreiraćemo samo jednu request klasu za svaki model, pri čemu će se detefminacija operacije vršiti samo u delu autorizacije. U nasatvku je prikazana request kalsa AktivnostRequest.

class AktivnostRequest extends *FormRequest*

{

    public function authorize(): bool

    {

        if (*$this*->isMethod('post')) {

            return *$this*->user()->can('create', *Aktivnost*::class);

        }

        if (*$this*->isMethod('put') || *$this*->isMethod('patch')) {

            $model = *$this*->route('aktivnosti');

            return $model && *$this*->user()->can('update', $model);

        }

        return false;

    }

    public function rules()

    {

        return [

            'naziv' => 'required|string|max:255',

            'opis' => 'nullable|string',

            'tip' => 'required|string|max:100',

            'pocetak' => 'required|date',

            'kraj' => 'required|date|after\_or\_equal:pocetak',

            'status' => 'required',

            'drustvo\_id' => 'required|exists:drustvos,id',

        ];

    }

    public function messages()

    {

        return [

            'naziv.required' => 'Naziv je obavezan.',

            'naziv.string' => 'Naziv mora biti tekst.',

            'naziv.max' => 'Naziv ne sme biti duži od 255 karaktera.',

            'opis.string' => 'Opis mora biti tekst.',

            'tip.required' => 'Tip je obavezan.',

            'tip.string' => 'Tip mora biti tekst.',

            'tip.max' => 'Tip ne sme biti duži od 100 karaktera.',

            'pocetak.required' => 'Datum početka je obavezan.',

            'pocetak.date' => 'Datum početka mora biti validan datum.',

            'kraj.required' => 'Datum završetka je obavezan.',

            'kraj.date' => 'Datum završetka mora biti validan datum.',

            'kraj.after\_or\_equal' => 'Datum završetka mora biti isti ili kasniji od početka.',

            'status.required' => 'Status je obavezan.',

            'drustvo\_id.required' => 'Društvo je obavezno.',

            'drustvo\_id.exists' => 'Izabrano društvo ne postoji.',

        ];

    }

    protected function failedAuthorization()

    {

        throw new *AuthorizationException*('Nemate dozvolu za ovu akciju.');

    }

}

Inicijalno, kreiranjem ove klase dobijaju se dve metode, metoda authorize i metoda rules. Metoda rules zadužena je za uspostavljanje validacionih pravila nad svakom vrednošću zadatog naziva ključa, u delu ključa asocijativnog niza njene povratne vrednosti.

Metoda authorize proverava autorizovanost korisnika. Zbog specifičnosti samog načina korišćenja ovih klasa, u metodi authorize prvo utvrđujemo koja od operacija se vrši na osnovu glagola zahteva (post, put ili patch), a zatim autorizovanost korisnika da obavi željenu operaciju. Iz zahteva metodom user dolazimo do prijavljenog korisnika, a zatim metodom can, utvrđujemo da li on može da izvrši operaciju, koja je prosleđena kao prvi parametar metode, nad modelom, čija se instanca ili metapodatak šalje kao drugi parametar metode. Na osnovu metapodatka, koji je prisutan i u instanci modela svakako, Laravel poziva definisanu metodu nad ogovarajućim policy-em, a sve zahvaljujući jegovoj povezanosti sa konkretnim modelom.

Metode messages i failedAuthorization kastomizuju greške koje se javljaju u slučaju narušavanja konkretnih validacinih pravila nad svakim od atributa, odnosno one koja se javlja u slučaju da metoda authorize vrati false.

# Rutiranje

Rute u Laravelu mogu biti definisane na više načina. Prvi način odnosi se na definisanje ruta pomoću apiRResource. ApiResource su rute koje se mogu jednostavno povezati sa api kontrolerima i samim tim bez eksplicitnog definisanja ruta za svaku metodu kontrolera, kreirati ih automatski. Sa druge strane, za kontrolere koji nisu api ili resource, mogu se definisati rute eksplicitnim navođenjem glagola zahteva, ali i metoda koje će se u kontrolerima na osnovu njih pozivati.

Za potrebe naše aplikacije fromirane su api rute u api.php i dve rute u web.php koje su korišćene samo u svrhe testiranja. U nastvaku je dat deo koda api.php fajla.

*Route*::middleware(['auth:sanctum'])->group(function () {

*Route*::apiResource('aktivnosti', *AktivnostController*::class);

*Route*::apiResource('drustva', *DrustvoController*::class);

*Route*::apiResource('komentari', *KomentarController*::class);

*Route*::apiResource('kosnice', *KosnicaController*::class);

*Route*::apiResource('pcelinjaci', *PcelinjakController*::class);

*Route*::apiResource('sugestije', *SugestijaController*::class);

*Route*::apiResource('user', *UserController*::class);

*Route*::post('/logout', [*AuthController*::class, 'logout']);

*Route*::post('/notifikacije/{id}', [*NotifikacijaController*::class, 'read']);

*Route*::get('/notifikacije', [*NotifikacijaController*::class, 'showAll']);

});

*Route*::get('/proizvodi', function () {

    $response = *Http*::get('https://dummyjson.com/products');

    $proizvodi = collect($response->json()['products'])->map(function ($p) {

        return [

            'naziv' => $p['title'],

            'cena' => $p['price'],

            'kategorija' => $p['category'],

        ];

    });

    return response()->json($proizvodi);

});

*Route*::post('/login', [*AuthController*::class, 'login']);

*Route*::post('/register', [*AuthController*::class, 'register']);

Kako bi rute bile zaštićene neophodno je koristiti middleware u kombinaciji sa sanctum paketom koji smo preuzeli i koji je namenjen za autentifikaciju. U slučaju kada rute imaju atribute koji se ponavljaju kao što je to slučaj kod ruta koje su zaštićene, moguće je koristiti groupe funkciju, koja navedeni atribut primenjuje na sve rute, u našem slučaju middleware zaštitu.

Rute koje nisu zaštićene nalaze se izvan groupe funcije, a to su rute za registraciju i prijavu, obzirom da svaki korisnik može da proba da se prijavi, odnosno da može da se registruje na sistem.

Za potrebe testiranja kreirali smo i rute koje su za sad u web.php fajlu koji je dat u nastavku.

*Route*::get('/comm', [*KomentarDownload*::class, 'pogled']);

*Route*::get('/komentari/{id}/preuzmi', [*KomentarDownload*::class, 'generateKomentarDoc']);

Kako je za ove rute, tj. za prvu rutu iz koje se na neki način poziva i druga, neophodno je bilo prikazati određen pogled. Obzirom da je testiranje rađeno u postmanu, koji nije najbolji alat za testiranje prikaza, ove rute smeštene su u drugi fajl.

# Notifikacije

Još jedan od specifičnih zahteva ovog projekta jeste kreiranje notifiakcija, odnosno obaveštenja. Iako je u konceptualnom modelu notifikacija predstavljena kao jedan od koncepata, odnosno klasa, Laravel poznaje mehanizam obaveštenja i pojednostavljuje proces njegove implementacije. Za početak potrebno je kreirati notifikaciju pozivanjem naredbe php artisan make:notifiaction Notifikacija, a zatim i php artisan table:notifications, kojom se kreira migracija za kreiranje tabele notifikacija. U nastavku je prikazana klasa notifikacija Notifikacija.

class Notifikacija extends *Notification*

{

    use *Queueable*;

    protected $aktivnost;

    /\*\*

     \* Create a new notification instance.

     \*/

    public function \_\_construct(*Aktivnost* $aktivnost)

    {

*$this*->aktivnost = $aktivnost;

    }

    /\*\*

     \* Get the notification's delivery channels.

     \*

     \* @return array<int, string>

     \*/

    public function via(object $notifiable): array

    {

        return ['database'];

    }

    /\*\*

     \* Get the mail representation of the notification.

     \*/

    public function toMail(object $notifiable): *MailMessage*

    {

        return (new *MailMessage*)

            ->line('The introduction to the notification.')

            ->action('Notification Action', url('/'))

            ->line('Thank you for using our application!');

    }

    /\*\*

     \* Get the array representation of the notification.

     \*

     \* @return array<string, mixed>

     \*/

    public function toArray(object $notifiable): array

    {

        return [

            'naziv' => *$this*->aktivnost->naziv,

            'opis' => *$this*->aktivnost->opis,

            'tip' => *$this*->aktivnost->tip,

            'pocetak' => *$this*->aktivnost->pocetak,

            'kraj' => *$this*->aktivnost->kraj,

            'status' => *$this*->aktivnost->status,

            'drustvo\_id' => *$this*->aktivnost->drustvo\_id,

            'user\_id' => *$this*->aktivnost->user\_id,

            'poruka' => 'Podsetnik: aktivnost "' . *$this*->aktivnost->naziv . '" počinje uskoro.',

        ];

    }

}

Obzirom da će se notifikacije slati za neke aktivnosti potrebno je da u samoj klasi postoji varijabla tipa Aktivnost koja se inicijalizuje pomoću konstruktora. Notifiakcije u Laravelu mogu se slati interno, odnsno kroz bazu podataka, putem mail-a ili slack-a. Zbog navedenog u metodi via navodimo database, obzirom da ćemo notifikacije slati samo interno, odnsono podsredstvom baze podataka.

Metoda toArray odgovorna je za postavljanje sadržaja notifikacije koji će biti zapamćen u bazi podataka.

Migracija notifikacija prikazana je na slici ispod.

return new class extends *Migration*

{

    /\*\*

     \* Run the migrations.

     \*/

    public function up(): void

    {

*Schema*::create('notifications', function (*Blueprint* $table) {

            $table->uuid('id')->primary();

            $table->string('type');

            $table->morphs('notifiable');

            $table->text('data');

            $table->timestamp('read\_at')->nullable();

            $table->timestamps();

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*/

    public function down(): void

    {

*Schema*::dropIfExists('notifications');

    }

};

Za potrebe slanja notifikacija implemetiraćemo artisan komandu. Pokretanjem ove komande za sve aktivnosti za koje notifikacija nije prosleđena, a odvijaju se u narednih 24 časa, notifikacija će biti poslata. Kastom komande kreiraju se u fajlu console.php. U nastavku je prikazan izgled fajla.

*Artisan*::command('aktivnosti:posalji-notifikacije-sutra', function () {

    $sutra = now()->addDay()->toDateString();

    $aktivnosti = *\App\Models\Aktivnost*::whereDate('pocetak', $sutra)

        ->where('notifikacija\_poslata', false)

        ->get();

    if ($aktivnosti->isEmpty()) {

*$this*->warn("Nema aktivnosti za sutra.");

        return;

    }

    foreach ($aktivnosti as $aktivnost) {

        $user = $aktivnost->user ?? null;

        if ($user) {

            $user->notify(new *\App\Notifications\Notifikacija*($aktivnost));

            $aktivnost->notifikacija\_poslata = true;

            $aktivnost->save();

*$this*->info("Notifikacija poslata korisniku ID {$user->id} za aktivnost '{$aktivnost->naziv}'");

        } else {

*$this*->warn("Nema korisnika za aktivnost ID {$aktivnost->id}");

        }

    }

});

Nad klasom Artisan pozivamo metodu command za kreiranje nove komande. Kao prvi parametar prosleđujemo naziv komande koji treba da usledi nakon dela php artisan, a zatim closure, odnosno anonimnu fukciju čije će izvršavanje uslediti. Na početku utvrđujemo sutrašnji datum, a na osnovu datuma pokretanja komande, odnosno trenutka pokretanja. Zatim pronalazimo sve aktivnosti za koje notifikacija nije poslata i koje počinju sutra. Ukoliko nema takvih aktivnosti u teminalu ćemo biti obavešteni o tome. U suprotnom, za svaku aktivnost izdvojiće se korisnik, koji ako je učitan, će biti obavešten. Metodom notify, koja je deo traita Notifiable, uključenom u modelu User, kreira se n-torka u bazi podataka, a na osnovu instance notifikacije u čiji konstruktor prosleđujemo aktivnost. Usled uspešnog slanja notifikacija, u terminalu će se pojaviti odgovarajuća poruka.

# Seeder i Factory

Za brzo popunjavanje baze podataka i održavanje konzistentnosti strukture podataka u bazi koriste se seeder i factory. Seeder su klase koje se koriste za sejanje odnosno popunjavanje baze podataka, dok factory podrazumevaju klase koje se koriste za formatiranje podataka koji će biti uneti u bazu. Ove dve klase su komplementarne odnosno koriste se zajedno. Dok factory sa jedne strane kreira insatnce modela u radnoj memoriji, seeder obezbeđuje da one budu zapamćene u bazi podataka, pomoću centrane klase DatabaseSeeder.

Kako bismo kreirali factory, prilikom kreiranja modela, u fleg navodimo f, odnosno php artisan make:model Model -mf. Dok m označava migraciju, f se odnosi na fabričku klasu. Pored kreiranja samih fabrika, za svaki kreiran model uključuje se i trait Factory, kojim se omogućava korišćenje metode factory nad samom kalsom modela. U nastavku je prikazana kalsa factory modela Pcelinjak.

class PcelinjakFactory extends *Factory*

{

    /\*\*

     \* Define the model's default state.

     \*

     \* @return array<string, mixed>

     \*/

    public function definition(): array

    {

        return [

            'naziv' => *$this*->faker->company(),

            'lokacija' => *$this*->faker->address()

        ];

    }

}

Ove klase poseduju objekat faker koji ima veći broj metoda za kreiranje lažnih podataka odgovarajuće strukture. Metoda company kreiraće nazive komplanija, naravno lažne, dok će metoda address kreirati lažne adrese.

Za svaki model krerian je factory. U factory klasama nisu definisani spoljni ključevi ukoliko ih model poseduje, već kako bi se zasigurno očuvao integritet podataka, ovaj proces izmešten je u seeder-e.

Za svaki model kreiran je i seeder. Same klase ne moraju biti vezane za konretne modele, i isto tako može postojati jedna klasa za sejanje svih tabela. Ali kako bi se očuvala čistoća koda i činjenica da metode koje se pozivaju unutar neke druge metode treba da butu istog nivoa apstraktnosti, kreirali smo više seeder-a. U nastavku je data klasa seeder modela Drustvo.

class DrustvoSeeder extends *Seeder*

{

    /\*\*

     \* Run the database seeds.

     \*/

    public function run(): void

    {

        $kosnice=*\App\Models\Kosnica*::all();

        foreach($kosnice as $kosnica){

*Drustvo*::factory(2)->create([

                'kosnica\_id'=>$kosnica->id

            ]);

        }

    }

}

U metodi run, prvo dolazimo do svih košnica koje već postoje u bazi. Zatim, foreach petljom za svaku košnicu kreiramo dva društva koristeći factory, a zatim kreirane instance pamtimo u bazi. Pre samog pamćenja, na asocijativni niz fabričke klase konkateniramo, odnosno dodajemo, podatak o košnici. Na ovaj način zasigurno smo očuvali referencijalni integritet baze podataka.

Centralno mesto sedeer-a jeste klasa DatabaseSeeder koja se inicijalno nalazi u Laravel projektu. U nastavku je prikazan kod pomenute klase.

class DatabaseSeeder extends *Seeder*

{

    /\*\*

     \* Seed the application's database.

     \*/

    public function run(): void

    {

*$this*->call(*UserSeeder*::class);

*$this*->call(*PcelinjakSeeder*::class);

*$this*->call(*KosnicaSeeder*::class);

*$this*->call(*DrustvoSeeder*::class);

*$this*->call(*AktivnsotSeeder*::class);

*$this*->call(*KomentarSeeder*::class);

*$this*->call(*SugestijaSeeder*::class);

    }

}

Zahvaljujući tome što smo kreirali seeder za svaki model, centralno mesto seeder-a je pregledno i intuitvino, dok je svaka metoda istog nivoa apstraktnosti. U metodi run pozivamo run metode pojedinalnih seeder klasa. Redosled poziva je jako značajan za očuvanje referencijalnog integriteta. Kako bismo bazu posejali u terminalu unosimo narednu php artisan db:seed. Pre ove naredbe je bitno da smo pokrenuli narednu migrate kojom se inicijalno formira baza i tabele u njoj zajedno sa odgovarajućim kolonama. Ove dve naredbe mogu se pokenuti i zajedno narednom php artisan migrate –seed koja će istovremeno kreirati i posejati bazu podataka.

# REST API

|  |  |
| --- | --- |
| Opis funkcije | **Prikaz svi aktivnosti** |
| HTTP metoda | GET |
| URL | /api/aktivnosti |
| URL parametri | ?pocetak=6 |
| HTTP body parametri | Nema |
| Format HTTP body parametara | / |
| Izlazni parametri | {              "id": 17,              "naziv": "Nesto",              "opis": "Prskanje radi zaštite od varoe",              "tip": "SEZONSKA",              "pocetak": "2025-06-20",              "kraj": "2025-06-21",              "status": "PLANIRANA",              "drustvo": {                  "id": 1,                  "matica\_starost": 4,                  "jacina": "srednje",                  "datum\_formiranja": "2023-11-30"              },              "user": {                  "id": 1,                  "name": "Mrs. Noemy Littel",                  "email": "xzieme@example.org"              },              "komentars": [],              "sugestijas": []          } |
| Format izlaznih parametara | application/json |
| Opis funkcije | **Kreiranje aktivnosti** |
| HTTP metoda | POST |
| URL | /api/aktivnosti |
| URL parametri | nema |
| HTTP body parametri | {    "naziv":"Zaštita",    "opis": "Prskanje radi zaštite od varoe",    "tip": "SEZONSKA",    "pocetak": "2025-06-20",    "kraj": "2025-06-21",    "status": "PLANIRANA",    "drustvo\_id": 1  } |
| Format HTTP body parametara | (nema) |
| Izlazni parametri | "message": "Aktivnost uspešno kreirana",      "model": {          "id": 18,          "naziv": "Nesto",          "opis": "Prskanje radi zaštite od varoe",          "tip": "SEZONSKA",          "pocetak": "2025-06-20",          "kraj": "2025-06-21",          "status": "PLANIRANA"      } |
| Format izlaznih parametara | application/json |
| Opis funkcije | **Prikaz jedne aktivnosti** |
| HTTP metoda | GET |
| URL | /api/aktivnosti/{aktivnosti } (1) |
| URL parametri | nema |
| HTTP body parametri | Nema |
| Format HTTP body parametara | / |
| Izlazni parametri | "id": 1,          "naziv": "Officia laudantium ipsa atque.",          "opis": "Cumque modi quis iure nihil. Dolorum molestiae quae expedita quo velit hic. Quisquam atque itaque id eum voluptatum suscipit est.",          "tip": "NESEZONSKA",          "pocetak": "2025-07-09",          "kraj": "2025-07-10 04:38:36",          "status": "ZAVRSENA",          "drustvo": {              "id": 1,              "matica\_starost": 4,              "jacina": "srednje",              "datum\_formiranja": "2023-11-30"          },          "user": {              "id": 1,              "name": "Mrs. Noemy Littel",              "email": "xzieme@example.org"          },          "komentars": [              {                  "id": 1,                  "sadrzaj": "Aut beatae necessitatibus et dolor nesciunt atque cumque. Animi harum quia expedita totam ipsa. Sed eum impedit consequatur sint provident.",                  "datum": "2025-05-25"              },              {                  "id": 2,                  "sadrzaj": "Doloremque consequatur sed voluptates et nisi quos ad. Ullam repellat vel deserunt ea molestiae mollitia.",                  "datum": "2025-02-03"              }          ],          "sugestijas": [              {                  "id": 1,                  "poruka": "Aut ipsa illum voluptate.",                  "datum\_kreiranja": "2025-05-28"              },              {                  "id": 2,                  "poruka": "Ut saepe quaerat rerum non.",                  "datum\_kreiranja": "2025-04-06"              }          ]      } |
| Format izlaznih parametara | application/json |
| Opis funkcije | **Azuriranje aktivnosti** |
| HTTP metoda | GET |
| URL | /api/aktivnosti/{aktivnosti} (1) |
| URL parametri | nema |
| HTTP body parametri | {    "naziv": "Prskanje pčela",    "opis": "Prskanje radi zaštite od varoe",    "tip": "SEZONSKA",    "pocetak": "2025-06-20",    "kraj": "2025-06-21",    "status": "PLANIRANA",    "drustvo\_id": 2  } |
| Format HTTP body parametara | Application/json |
| Izlazni parametri | "message": "Aktivnost uspešno ažurirana",      "model": {          "id": 1,          "naziv": "Prskanje pčela",          "opis": "Prskanje radi zaštite od varoe",          "tip": "SEZONSKA",          "pocetak": "2025-06-20",          "kraj": "2025-06-21",          "status": "PLANIRANA"      } |
| Format izlaznih parametara | application/json |
| Opis funkcije | **Login** |
| HTTP metoda | Post |
| URL | /login |
| URL parametri | Nema |
| HTTP body parametri | {      "email":"travis10@example.com",      "password":"password"  } |
| Format HTTP body parametara | application/json |
| Izlazni parametri | {      "id": 2,      "token": "6|CrWEottWs3w6Og13szhpevqeJKyYmk2svX73BwK6adbe30ab",      "name": "Miss Camylle Barrows",      "email": "travis10@example.com"  } |
| Format izlaznih parametara | application/json |
| Opis funkcije | **Register** |
| HTTP metoda | Post |
| URL | /register |
| URL parametri | Nema |
| HTTP body parametri | {      "name":"Ana Pavlovic",      "email":"ana123@gmail.com",      "password":"password"  } |
| Format HTTP body parametara | application/json |
| Izlazni parametri | {      "message": "Account created",      "user": {          "id": 5,          "name": "Ana Pavlovic",          "email": "ana123@gmail.com"      }  } |
| Format izlaznih parametara | application/json |
| Opis funkcije | **Logout** |
| HTTP metoda | Post |
| URL | /logout |
| URL parametri | Nema |
| HTTP body parametri | nema |
| Format HTTP body parametara | / |
| Izlazni parametri | {      "message": "Logged out successfully"  } |
| Format izlaznih parametara | application/json |

# Korisničko uputstvo

Opis i screenshot-ovi aplikacije testirane u okviru Postman-a- (ili nekog drugog servisa za testiranje API putanja)