MVC server-side aplikacija za praćenje i administraciju nogometne lige

Bošnjak, Petar

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:278870

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-27**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

MVC SERVER-SIDE APLIKACIJA ZA PRAĆENJE I ADMINISTRACIJU NOGOMETNE LIGE

Diplomski rad

Petar Bošnjak

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 05.11.2021.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime studenta:	Petar Bošnjak
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 836 R, 08.10.2020.
OIB studenta:	33329530628
Mentor:	Izv. prof. dr. sc. . Damir Filko
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv. prof. dr. sc. Emmanuel-Karlo Nyarko
Član Povjerenstva 1:	Izv. prof. dr. sc. . Damir Filko
Član Povjerenstva 2:	Izv. prof. dr. sc. Krešimir Nenadić
Naslov diplomskog rada:	MVC server-side aplikacija za praćenje i administraciju nogometne lige
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	Napraviti aplikaciju koja bi služila svim posjetiteljima stranice za pregled rasporeda i raznih statističkih podataka nogometne lige, a administratorima, zapisničarima, klubovima i sl., interaktivnu administraciju lige, klubova i igrača. Za razvoj koristiti ASP.NET-a i Entity razvojni okvir.
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	05.11.2021.

	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 22.11.2021.

Ime i prezime studenta:	Petar Bošnjak
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D 836 R, 08.10.2020.
Turnitin podudaranje [%]:	8

Ovom izjavom izjavljujem da je rad pod nazivom**: MVC server-side aplikacija za praćenje i administraciju nogometne lige**

izrađen pod vodstvom mentora Izv. prof. dr. sc. . Damir Filko

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1 UVOD 1
2 SLIČNA RJEŠENJA 2
3 KORIŠTENI ALATI I TEHNIKE 4
3.1 ASP.NET Core MVC 3.1 4
3.2 Višeslojna arhitektura Onion 5
3.3 Entity Framework Core 6
3.4 PostgreSQL 6
3.5 Database First ili Model First pristup 7
3.6 Automapper 7
3.7 Autofac dependency injection 8
4 RAZVOJ APLIKACIJE ZA ADMINISTRACIJU NOGOMETNE LIGE 9
4.1 Baza podataka 9
4.2 Data Access Layer sloj 10
4.3 Repository sloj 11
4.4 Servis sloj 12
4.5 Web sloj 14
4.6 Ostali slojevi 19
5 PRIMJENA APLIKACIJE ZA ADMINISTRACIJU LIGE 20
5.1 Javni dio aplikacije 21
5.2 Administrativni dio aplikacije 26
6 ZAKLJUČAK 34
LITERATURA 35
SAŽETAK 37
ABSTRACT 38
ŽIVOTOPIS 39

1 UVOD

Potreba za digitalizacijom administracije i što bržim dohvaćanjem informacija veća je nego ikad. Aplikacija koja je opisana u ovome radu je internet aplikacija koja za cilj ima poslužiti svim posjetiteljima stranice za pregled rasporeda i raznih statističkih podataka nogometne lige, a administratorima, zapisničarima, klubovima i slično, interaktivnu administraciju lige, klubova i igrača.

Ovisno u ulozi korisnika aplikacija nudi različite mogućnosti. Gost stranice moći će pregledavati rezultate utakmica, sve relevantne tablice i statističke podatke, međusobne omjere te postave klubova u nadolazećim utakmicama. Klubovi će moći obavljati administraciju vlastite momčadi, podnositi zahtjev za registracijom novih igrača i osoblja, prijaviti momčadi za nadolazeće utakmice, te podnositi pritužbe na utakmice, suce, zapisničara i slično. Zapisničari, suci i igrači moći će pregledavati svoj profil, vidjeti prikaz svojeg rasporeda, te podnositi različite zahtjeve i pritužbe, ovisno o ulozi, administratoru lige. Administrator će imati mogućnost uređivanja i postavljanja svega što je aktivno na stranici, te rješavanje pritužbi i zahtjeva.

U glavnom dijelu rada možemo vidjeti nekoliko različitih poglavlja. U drugom poglavlju opisana su slična rješenja, u trećem poglavlju opisani su alati i tehnike koji su korišteni prilikom izrade aplikacije, u četvrtom poglavlju opisan je detaljan rad aplikacije zajedno sa programskim kodom, u petom poglavlju opisana je primjena aplikacije dok je u zadnjem poglavlju predstavljen zaključak.

2 SLIČNA RJEŠENJA

Javni dio aplikacije, u kojem se mogu vidjeti podaci o tablici lige, igračima i klubovima, kao i odigrane i nadolazeće utakmice, najsličniji je servisu *Rezultati.com* [1]. Najveće razlike između spomenutog servisa i aplikacije opisane u ovome radu jesu:

1. Servis *Rezultati.com* podržava prikaz podataka za više različitih liga i za više sportova.
2. Servis *Rezultati.com* ne posjeduje funkcionalnosti za administrativni dio, zbog čega nije moguće kao administrator kluba, trener ili zapisničar direktno upravljati nogometnom ligom.

Servis *SofaScore* [2] vrlo je sličan servisu *Rezultati.com*, što znači da je prikaz liga, sportova te statističkih podataka puno opsežniji nego aplikacija opisana u ovome radu, no nedostaje im administrativni dio. Iako servis *Rezultati.com* ima nešto opsežniju ponudu liga i sportova, servis *SofaScore* u svojoj ponudi ima listu utakmica sa besplatnim prijenosom i videozapise sa najvažnijim isječcima završenih utakmica.

SportsPlus servis [3] najsličniji je administrativnom dijelu aplikacije opisane u ovome radu. U svojoj usluzi nudi administraciju igrača, klubova, rasporeda, liga, turnira, volontera i ostalih osoba povezanih sa administracijom lige. Servis nije besplatan i nije otvorenog programskog koga, stoga su korisnici ograničeni sa postavkama koje servis nudi. Zbog manjka otvorenog programskog koda nije moguće odrediti na kojim se arhitekturama zasniva, no zbog postojanja mobilne i web aplikacije sigurno se ne radi o *MVC* arhitekturi pošto je ona isključivo *server-side*.

TeamSnap servis [4] jedan je od popularnijih servisa za administraciju igrača, klubova, rasporeda, liga i turnira. Po funkcionalnostima vrlo je sličan *SportsPlus* servisu, pogodan je manjim organizacijama, uključuje obučavanje korisnika uživo, no manjka 24/7 korisničku podršku. Najveća mana mu je što nije otvorenog programskog koda i što se mora plaćati, no nudi besplatnu uslugu administracije kluba, dok se za administraciju liga i turnira usluga mora plaćati.

Od aplikacija otvorenog koda najsličnija aplikaciji koja je opisana u ovome radu jest *Football-League-Management-System* dostupna na *GitHub* repozitoriju [5]. Autor ove aplikacije opisuje ju kao sustav upravljanja razvijen kao individualni projekt za *ASP.NET MVC* tečaj na *Telerik* akademiji. Nudi pregled tablice, upravljanje ligom, sezonama, timovima, igračima i utakmicama. Najveća razlika između navedene aplikacije i aplikacije opisane u ovome radu jesu korišteni alati, poput alata za razvoj korisničkog sučelja *KendoUI*, alat koji nije besplatan i dio je *Telerik* paketa za kreiranje web sadržaja. Zatim arhitektura aplikacije, u ovom radu daje se više naglaska

na *Onion* arhitekturu gdje se sva poslovna logika stavlja u poseban servis sloj, a sav programski sloj za komunikaciju sa bazom podataka u *repository* sloj. U konačnici, u ovome radu, sama izvedba aplikacije po ulogama u sustavu jest drugačija i opsežnija.

3 KORIŠTENI ALATI I TEHNIKE

Kao glavna tehnologija za izradu aplikacije korišten je *ASP.NET Core MVC 3.1*, za komunikaciju sa bazom korišten je *Entity Core Framework*, korištena baza je *PostgeSQL*, dok je za integrirano razvojno okruženje (engl. *Integrated Development Environment*) korišten *Visual Studio Community 2019*. Aplikacija je dopunjena *JavaScript* bibliotekom *jQuery* radi lakše manipulacije podacima na klijentskoj strani aplikacije.

3.1 ASP.NET Core MVC 3.1

Prema [6], *ASP.NET Core MVC* je softverski okvir za razvoj web aplikacija tvrtke *Microsoft* koji kombinira učinkovitost i urednost arhitekture *Model-View-Controller* (MVC), ideja i tehnika iz agilnog razvoja i najbolje dijelove .NET platforme.

Za shvaćanje njegovog načina rada potrebno je znati način funkcioniranja MVC arhitekture. MVC arhitektura dijeli programski kod u tri dijela:

- Modeli
- Pogledi
- Kontrolori

U modele se najčešće stavlja sva poslovna logika (engl. *business logic*) aplikacije uključujući strukturiranje podataka, no pošto se u ovoj aplikaciji koristi skupa sa višeslojnom arhitekturom *Onion,* modeli služe isključivo za strukturiranje podataka koji se prikazuju na pojedinim pogledima. Pogledi služe za prikaz podataka, najčešće u *HTML* formatu. Uobičajena praksa je ne koristiti nikakvu manipulaciju podacima ni poslovnu logiku unutar pogleda, kako ne bi dolazilo do konflikata u većim projektima i zbog čistoće programskog koda. Kontrolori su spona između pogleda i modela, primaju različite ulazne jedinice (engl. *input*) koje koriste za upravljanje modelima i prosljeđivanje podataka pogledima.

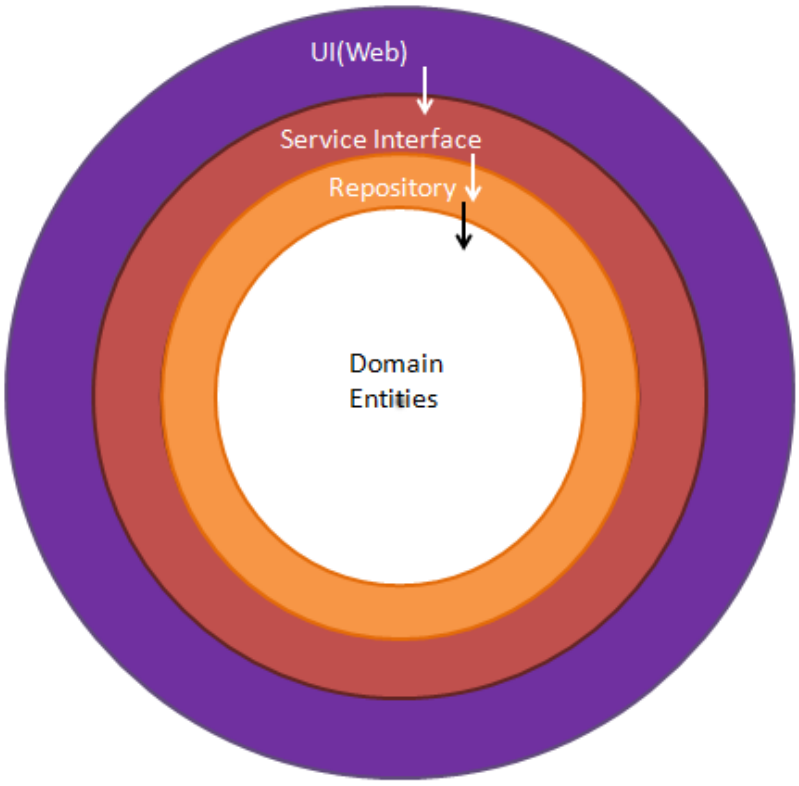
Prema [7], razlog dijeljenja programskog koda jest učinkovita upotreba koda, u smislu definiranja zamjenjivih cjelina koje se mogu zamijeniti ili promijeniti bez većeg utjecaja na ostatak koda, te podrške za paralelni razvoj među više ljudi, primjerice, omogućuje *frontend* i *backend* programeru paralelni rad na aplikaciji bez stvaranja konflikata u kodu.

.NET Core softverski okvir, za razliku od *.NET Framework-a*, podržava sve *Linux* i *macOS* platforme, uključujući i *Microsoft-ove*, što znači da se može koristiti na većini servera. Koristi

C# programski jezik i sve njegove prednosti, uključujući asinkrono programiranje, *extension* metode, *lambda* izraze, *anonymous* i *dynamic* tipove podataka, *dependency injection*, *Language Integrated Query* (*LINQ*) itd [6].

3.2 Višeslojna arhitektura Onion

Izraz *Onion* arhitektura predložio je Jeffrey Palermo 2008 godine. Ova arhitektura pruža bolji način za izradu aplikacija za bolju provjerljivost, održivost i pouzdanost u infrastrukturama poput baza podataka i servisa [8]. Ideja te višeslojne arhitekture je podijeliti aplikaciju u nekoliko slabo spojenih (engl. *Loose Coupling*) smislenih cjelina spojenih u stog, pri čemu određeni sloj zna za postojanje samo sloja koji se nalazi direktno ispod njega, uz neke iznimke, najčešće generičkog koda kojeg je potrebno koristiti kroz čitavi projekt. Iako ne postoji službeni broj slojeva koji se mogu koristiti, najčešće korišteni slojevi prikazani su na slici 3.1.



Sl. 3.1. Višeslojna arhitektura *Onion*

U ovoj aplikaciji koriste se 4 glavna sloja:

- *Database Access Layer* (DAL), kao sloj na dnu stoga zaslužan je za svu komunikaciju sa bazom podataka.

- *Repository* sloj koji se koristi za kreiranje svih filtriranih upita za CRUD (*Create, Read, Update and Delete*) operacije nad bazom podataka.
- *Service* sloj u kojem se nalazi sva poslovna logika.
- *Web* sloj koji sadržava MVC dio aplikacije zadužen za prikaz podataka korisnicima.

Uz ta 4 glavna sloja koriste se još dva sloja sa dijeljenim kodom kojima mogu pristupiti sva 4 glavna sloja, *Model* sloj koji sadržava sve domenske modele korištene u aplikaciji i *Common* sloj koji sadržava sav kod koji se proteže kroz cijelu aplikaciju.

3.3 Entity Framework Core

Entity Framework Core je lakša verzija *Microsoft-ovog Entity Framework-a*, tehnologije za pristup podacima i uređivanje baze podataka [9]. Kao i *.NET Core*, *EF Core* je *cross-platform*, što znači da podržava *Linux* i *macOS* platforme.

U ovoj aplikaciji služi kao *object-relational mapper* (*ORM*) čija je svrha eliminacija potrebe za korištenjem *SQL* koda za manipuliranje bazama podataka. Umjesto toga koristi *LINQ* za slanje upita na bazu podataka.

EF Core podržava sve veće tipove baza podataka, poput *PostgreSQL*, *MySQL* i *MSSQL*. U ovoj aplikaciji korištena je *PostgreSQL* lokalna baza podataka zbog svoje licence otvorenog koda.

3.4 PostgreSQL

PostgreSQL je moćan sustav objektno-relacijskih baza podataka s otvorenim izvorom koji koristi i proširuje *SQL* jezik u kombinaciji s mnogim značajkama koje sigurno pohranjuju i skaliraju najsloženija radna opterećenja podataka [10].

Ponajviše je korišten zbog svoje licence otvorenog koda koja omogućuje besplatno korištenje u privatne i komercijalne svrhe, no velika mu je prednost poprilična podržanost od strane različitih servera, operacijskih sustava i alata za uređivanje baza podataka. Dobro je napomenuti i neke od tipova podataka koje baza podržava, poput *jsonb*, koji služi za pohranu i manipulaciju podataka u formatu *JSON*, čime se *PostgreSQL* smatra hibridnim jezikom između *SQL* i *NoSQL* jezika. Isto tako, jedan od posebno korisnih tipova podataka je *citext*, ili *case-insensitive text* koji uvelike olakšava pretraživanje polja kod kojih razlikovanje malih i velikih slova nije potrebno.

3.5 Database First ili Model First pristup

Pri kreiranju aplikacije sa *Onion* arhitekturom moguće je razlikovati dva glavna pristupa za kreiranje baze podataka i *DAL* sloja: *Database First* i *Model First* pristup.

Kod *Model First* pristupa u samom kodu prvo se kreiraju *entity* modeli i pripadajuća klasa *DbContext* koja, pri pokretanju aplikacije i određenih naredbi *Entity Framework-a,* kreira bazu podataka i sve tablice koje odgovaraju našim *entity* modelima. Prednost ovakvog pristupa je ta što programer ne mora nužno znati programski jezik za manipulaciju bazama podataka i generalno je brži način kreiranja baze i tablica.

Kod *Database First* pristupa prvo se kreira baza podataka i sve potrebne tablice, te se zatim, pomoću *scaffold* naredbi *Entity Framework-a,* kreiraju svi *entity* modeli i pripadajuća klasa *DbContext*. Prednost ovog pristupa je veća kontrola nad strukturom baze, tipovima podataka itd.

Moguće je koristiti i oba pristupa istovremeno, pogotovo ukoliko se pokušava integrirati postojeća baza podataka u novu aplikaciju, pri čemu se prvo koristi *Database First* pristup za kreiranje *DAL* sloja, a u budućnosti se koristi *Model First* pristup za dodavanje novih tablica i polja.

3.6 Automapper

Kod kompleksnijih aplikacija najčešće se koriste tri vrste modela. *Entity* modeli, koji se nalaze u *DAL* sloju i svojom strukturom u potpunosti moraju odgovarati tablicama u bazi podataka. Domenski modeli koji se koriste za prijenos podataka iz jednog sloja u drugi kroz cijelu aplikaciju. Svojom strukturom mogu biti identični e*ntity* modelima, no mogu imati i zasebna svojstva (engl. *property*), a možemo koristiti i modele koji nisu povezani sa bazom, te se koriste samo za poslovnu logiku ili prijenos parametara. Kao zadnja vrsta modela su modeli koji se koriste za prijenos podataka krajnjem korisniku, ovisno o strukturi aplikacije mogu se zvati *ViewModel, REST model* ili *Data transfer object (DTO).* Oni se najčešće koriste zato što arhitekture kao *MVC* zahtijevaju samo jedan model po pogledu pa se stoga moraju svi domenski modeli spojiti u jedan, ili zato što ne želimo baš svako svojstvo od domenskih modela prikazati krajnjem korisniku.

Pošto sve tri vrste modela često imaju ista svojstva, kako bi izbjegli nepotrebno dupliciranje koda koji preslikava svojstva jednog modela u drugi [11], možemo koristiti vrlo koristan alat

poput *Automapper-a*, koji služi upravo tome da automatski preslikava svojstva jedne vrste modela u svojstva druge vrste modela.

3.7 Autofac dependency injection

Pojam *dependency injection* jedan je od osnovnih dijelova alata poput *.NET Core-a*. U suštini, *dependency injection* je zaslužan za komunikaciju između slojeva u *Onion* arhitekturi. Na primjer, ako postoji servis koji je zadužen za računanje koliko bodova ima svaka nogometna momčad u određenom trenutku, u taj servis moramo ubaciti (engl. *inject*) repozitorij koji nam služi za dohvaćanje svih utakmica, repozitorij koji nam služi za dohvaćanje svih klubova i slično. Alati za *dependency injection* nam omogućuju da pri kreiranju samog servisa, u konstruktoru, bez dodatnog koda ubacimo sve potrebne repozitorije u servis bez razmišljanja o kojim svojstvima određeni repozitorij ovisi.

Iako *.NET Core* ima ugrađene alate za *dependency injection*, za ovu aplikaciju je korišten zaseban alat imena *Autofac* [12] zbog svog lakšeg prilagođavanja i dodatnih mogućnosti.

4 RAZVOJ APLIKACIJE ZA ADMINISTRACIJU NOGOMETNE LIGE

Kroz sljedeće poglavlje, uz uvodni opis baze podataka, bit će opisani svi slojevi arhitekture aplikacije. Bit će dani primjeri implementacije svakog sloja uz potrebne naredbe, alate i skripte te primjeri programskog koda pomoću kojega su realizirani slojevi u aplikaciji.

4.1 Baza podataka

Kao što je već spomenuto u ovom radu, za bazu podataka korišten je *PostgreSQL*. Kod samog kreiranja baze korišten je *Database First* pristup što znači da je potrebno prvo napisati sve skripte za kreiranje i popunjavanje tablica. Kod tablica potrebno je razlikovati dvije različite vrste tablica, *relation* i *lookup* tablice. *Lookup* tablica je specifična po tome što se u njoj nalaze podaci koji se ne mijenjaju često i po tome što ona nema nikakvih relacijskih polja, što znači da nije ovisna ni o jednoj drugoj tablici, dok *relation* tablice su upravo suprotno. Primjer kreiranja *lookup* tablice *Role* može se vidjeti u sljedećoj skripti.

```
create table "Role" (
	"Id" uuid not null,
	"DateCreated" timestamp not null,
	"DateUpdated" timestamp not null,
	"Name" text not null,
	"Abrv" text not null,
	constraint "PK_Role" primary key ("Id")
);
```

Za polja *Name i Abrv* korišten je tip podatka *text* bez limita znakova, koji u *PostgreSQL-u*, za razliku od nekih drugih jezika, nema značajan negativan utjecaj na performanse čitanja i pisanja podataka [13].

Primjer kreiranja relacijske tablice *UserRole* koja spaja korisnike i njihove uloge u sustavu prikazan je u sljedećoj skripti.

```
create table "UserRole" (
	"Id" uuid not null,
	"DateCreated" timestamp not null,
	"DateUpdated" timestamp not null,
	"UserId" uuid not null,
	"RoleId" uuid not null,
	constraint "PK_UserRole" primary key ("Id"),
	constraint "FK_UserRole_User" foreign key ("UserId") references "User" ("Id")
	match simple on update no action on delete cascade,
	constraint "FK_UserRole_Role" foreign key ("RoleId") references "Role" ("Id")
	match simple on update no action on delete cascade
);
```

Važno je spomenuti relacijska polja *UserId i RoleId,* koja služe za referenciranje određenih korisnika i uloga, koja u ovom slučaju koriste tip podatka *uuid* (*Universally unique identifier*),

znan kao i *guid*. Taj tip podatka koristi se za kreiranje jedinstvene identifikacijske oznake za svaki unos u bazu podataka [14].

Primjer skripte za popunjavanje tablica je sljedeći:

```
insert into "Role" ("Id", "DateCreated", "DateUpdated" ,"Name",     "Abrv") values
(newid(), now(), now(), 'Player', 'player'),
(newid(), now(), now(), 'Coach', 'coach'),
(newid(), now(), now(), 'Referee', 'referee'),
(newid(), now(), now(), 'Official', 'official'),
(newid(), now(), now(), 'Administrator', 'admin'),
(newid(), now(), now(), 'Team Administrator', 'teamadmin');
```

Funkcija *now()* ugrađena je funkcija jezika *PostgreSQL* koja nam daje trenutni datum i vrijeme, dok je funkcija *newid()* ručno kreirana funkcija koja generira novu identifikacijsku oznaku po sljedećoj funkciji:

SELECT CAST(md5(current_database() || user || current_timestamp || random()) as uuid)

4.2 Data Access Layer sloj

Data Access Layer (DAL) zaslužan je za svu komunikaciju između baze podataka i ostatka koda. Kod *Database First* pristupa potrebno je prvo kreirati bazu, zatim otvoriti *Command Prompt* ili *PowerShell* aplikaciju, pozicionirati se u DAL datotečnu mapu projekta i pokrenuti *scaffold* naredbu za kreiranje *entity* modela i *DbContext* klase:

dotnet ef dbcontext scaffold "Host=localhost;Database=FootballLeague;Username=<username>; Password=<password>" Npgsql.EntityFrameworkCore.PostgreSQL

Kako bi naredba mogla funkcionirati potrebno je instalirati alat naziva *dotnet-ef*. U naredbi je potrebno podesiti konfiguracijski dio sa pripadajućim pristupnim podacima. Nakon pokretanja naredbe, *dotnet-ef* alat kreira sve *entity* modele i *DbContext* klasu naziva *FootballLeagueContext* pošto nam je naziv baze *FootballLeague*.

Kako bi aplikacija uopće mogla komunicirati sa *PostgreSQL* bazom potrebno je instalirati alat naziva *Npgsql*. To je moguće učiniti unutar integriranog razvojnog okruženja pomoću *NuGet package manager-a*.

Kako bi se mogao u budućnosti koristiti *Model First* pristup treba se prvo pokrenuti naredba

dotnet ef migrations add InitialCreate

kako bi uhvatili trenutnu presliku stanja baze podataka. Nakon toga, svaku promjenu u *DbContext* klasi moguće je primijeniti na bazu sa naredbom

dotnet ef database update

Na kraju, iz *DbContext* klase potrebno je maknuti *ConnectionString* svojstvo u zasebnu *JSON* datoteku u Web sloju i nakon toga kreirati dva konstruktora za *DbContext* klasu, prikazana u sljedećem kodu, od kojih se prvi poziva pri kreiranju novih migracija, dok se drugi poziva kada se pokrene aplikacija.

```
public FootballLeagueContext()
{
    string basePath = Environment.CurrentDirectory;
    string relativePath = "../FootballLeague.Web";
    string fullPath = Path.GetFullPath(relativePath, basePath);

    var builder = new ConfigurationBuilder()
        .SetBasePath(fullPath)
        .AddJsonFile("appsettings.json")
        .AddEnvironmentVariables();

    var config = builder.Build();

    ConnectionString = config.GetConnectionString("DefaultConnection");
}


public FootballLeagueContext(DbContextOptions<FootballLeagueContext> options,
IConfigurationManager configurationManager)
    : base(options)
{
    ConnectionString = configurationManager.GetConnectionString("DefaultConnection");
}
```

Potreba za tim je ta što se prilikom kreiranja novih migracija ne pokreće *dependency injection*, te se moraju ručno dohvatiti podaci za spajanje na bazu podataka iz *JSON* datoteke.

4.3 Repository sloj

U *repository* sloju cilj je staviti sav kod koji služi za jednostavne *CRUD* (*Create, Read, Update and Delete*) operacije nad bazom. Najvažnije je kreirati generički repozitorij sa definiranim generičkim parametrima koji sadrži sve osnovne operacije nad bazom poput dohvaćanja jednog zapisa iz baze, dohvaćanja liste zapisa, brisanja zapisa, unosa i uređivanja zapisa iz baze, kao metode koje se mogu primijeniti na bilo koju tablicu u bazi i samim time smanjiti broj dupliciranog koda. Deklaracija generičkog repozitorija prikazana je u sljedećem kodu.

```
public class GenericRepository<T, K> : IGenericRepository<T, K> where T : class where K :
BaseModel
```

Nakon toga moguće je za svaku tablicu kreirati zaseban repozitorij koji nasljeđuje metode generičkog repozitorija i ostavlja mogućnost dodavanja metoda koje su posebne za pojedine tablice, ukoliko je to potrebno.

U ovom trenutku potrebno je definirati pojam *interface*, kojeg možemo definirati kao predložak pojedine klase koji sadrži sve informacije o dostupnim javnim metodama i svojstvima te klase. *Interface* je bitan sastavni dio ovakve aplikacije jer je upravo on, a ne sama klasa, vidljiv višim slojevima aplikacije. Kada se repozitoriji ubacuju u servise ili servisi u metode web sloja putem *dependency injection-a*, koriste se *interface-i*.

4.4 Servis sloj

Sva poslovna logika aplikacije nastoji se staviti u servis sloj aplikacije. Tipičan primjer u ovoj aplikaciji bio bi *TeamService* servis, čija se deklaracija i konstruktor mogu vidjeti u sljedećem kodu.

```
public class TeamService : ITeamService
{
    private readonly Sorter DefaultSort = new Sorter("Club.Name|asc");

    protected ITeamRepository Repository { get; }

    protected IGameRepository GameRepository { get; }

    public TeamService(ITeamRepository repository, IGameRepository gameRepository)
    {
        Repository = repository;
        GameRepository = gameRepository;
    }
```

U kodu za deklaraciju i konstruktor servisa *TeamService* može se vidjeti na koji način funkcionira *dependency injection*, gdje se u konstruktoru servisa ubacuju repozitoriji za tablice *Team* i *Game*, te dodjeljuju pripadajućim svojstvima servisa.

TeamService, uz standardne *CRUD* metode, sadrži i metodu koja služi za kreiranje trenutne bodovne tablice svih momčadi, koja koristi i nekoliko domenskih modela koji nisu povezani sa bazom.

Lookup servisi, koji odgovaraju *lookup* tablicama kreiraju se drugačije od standardnih servisa. Naime, zbog toga što se podaci u *lookup* tablicama ne mijenjaju često ima ih smisla spremiti u različite predmemorije, poput *Redis Cache*, što nam omogućuje brže dohvaćanje podataka bez potrebe za opetovanim dohvaćanjem istih iz baze podataka.

Implementacija *Redis Cache* klijenta jednostavna je u *.Net Core-u*. Sve što je potrebno napraviti jest instalirati *NuGet* paket *StackExchange.Redis* i kreirati klasu prikazanu u sljedećem kodu.

```
public class CacheProvider : ICacheProvider
{
```

```
    private IDatabase Database { get; }

    public CacheProvider(IConfigurationManager configurationManager)
    {
        var host = configurationManager.GetConnectionString("RedisHost");
        var database = configurationManager.GetConnectionString("RedisDatabase");

        var connectionMultiplexer = ConnectionMultiplexer.Connect(host);
        Database = connectionMultiplexer.GetDatabase(Convert.ToInt32(database));
    }


    public async Task<T> GetOrAddAsync<T>(string key, Func<string, Task<T>> valueFactory)
    {
        var cacheItem = await Database.StringGetAsync(key);
        if (cacheItem.HasValue)
        {
            return JsonSerializer.Deserialize<T>(cacheItem);
        }
        else
        {
            var value = await valueFactory(key);
            var redisValue = new RedisValue(JsonSerializer.Serialize(value));
            await Database.StringSetAsync(key, redisValue);

            return value;
        }
    }
}
```

U konstruktoru potrebno je dohvatiti podatke za spajanje na *Redis* server i spojiti se. Metoda *GetOrAddAsync()* će nam, pri pozivanju, dohvatiti određeni *lookup* iz predmemorije ukoliko postoji ili dohvatiti *lookup* direktno iz baze podataka.

Kod kreiranja *lookup* servisa potrebno je kreirati svojstva koja odgovaraju zapisima iz tablice, te ih u konstruktoru *lazy load-ati*, kao što je prikazano u sljedećem kodu.

```
public RoleLookup(IRoleRepository repository, ICacheProvider cacheProvider)
{
    this.Repository = repository;
    this.CacheProvider = cacheProvider;

    Player = new AsyncLazy<RoleModel>(async () =>
    {
        return (await FindAllAsync()).SingleOrDefault(p => p.Abrv == PlayerAbrv);
    });

    Coach = new AsyncLazy<RoleModel>(async () =>
    {
        return (await FindAllAsync()).SingleOrDefault(p => p.Abrv == CoachAbrv);
    });
```

Lazy loading je princip odgode dohvaćanja podataka dok nam taj podatak nije potreban u kodu [15]. U protivnom bi se svi podaci odmah dohvatili iz baze ili predmemorije prilikom kreiranja

servisa što bi imalo negativan utjecaj na brzinu sustava. *AsyncLazy* klasa je jednostavna asinkrona adaptacija *lazy loading* funkcionalnosti [16].

S obzirom na to da l*ookup* servisi često nisu potrebni kroz cijeli životni ciklus drugih servisa ili kontrolora u web sloju, oni se kreiraju pomoću *LookupFactory* servisa. On se može ubaciti u bilo koji servis ili web kontrolor i pomoću njega u bilo kojem trenutku kreirati instanca bilo kojeg *lookup* servisa.

4.5 Web sloj

Struktura aplikacije koja odgovara standardnoj strukturi MVC arhitekture je sljedeća: Kontrolori, pogledi i modeli koji su smješteni u pripadajuće mape, dok se u *wwwroot* mapi nalaze statične datoteke poput vlastitih *JavaScript* skripti, *CSS* datoteke i slično. *Appsettings.json* datoteka sadrži zadane i vlastite konfiguracijske vrijednosti za aplikaciju. U njoj se mogu definirati podaci za pristup bazi podataka. *Program.cs* je ulazna točka aplikacije, koja poziva *Startup.cs* datoteku.

Startup.cs datoteka koristi se za, između ostalog, konfiguraciju korištenih servisa i deklaraciju korištenih modula. U sljedećem kodu možemo vidjeti konfiguraciju servisa za pristup bazi podataka, metodu *AddDbContext()*, koja prima parametar *options* koji uzima podatke za pristup lokalnoj bazi podataka koje dohvaća iz *Appsettings.json* datoteke.

```
services.AddDbContext<FootballLeagueContext>(options =>
options.UseNpgsql(Configuration.GetConnectionString("DefaultConnection")));
```

Microsoft Identity je servis koji uvelike olakšava kreiranje korisnika, njihovu autentifikaciju i dopuštenja za pristup određenim podacima u ovisnosti o njihovoj ulozi. *AddIdentity()* metoda služi za postavljanje *Microsoft-ovog* ugrađenog servisa za identitet. Taj servis inače kreira vlastitu bazu podataka sa tablicama za korisnike i uloge, no moguće ga je prilagoditi vlastitim tablicama. Ono što je potrebno napraviti jest kreirati vlastite klase *RoleStore i UserStore* koje moraju sadržavati sve metode određene pripadajućim *interface-ima*, te ih povezati sa postojećim *Microsoft Identity* servisom kao što je prikazano u sljedećem kodu.

```
services.AddIdentity<UserModel, RoleModel>().AddDefaultTokenProviders();
services.ConfigureApplicationCookie(options =>
{
    options.Cookie.HttpOnly = true;
    options.ExpireTimeSpan = TimeSpan.FromMinutes(300);
    options.LoginPath = "/login";
    options.LogoutPath = "/logout";
    options.SlidingExpiration = true;
});
```

Kako bi *Automapper* znao preslikavati jednu vrstu modela u drugi potrebno je kreirati klase koje nasljeđuju *Automapper* klasu *Profile* i u njima definirati pravila za preslikavanje, kao što je prikazano u sljedećem kodu.

```
public class ModelProfile : Profile
{
    public ModelProfile()
    {
        CreateMap<UserModel, User>()
            .ForMember(p => p.Coach, opt => opt.Ignore())
            .ForMember(p => p.Player, opt => opt.Ignore())
            .ForMember(p => p.UserRole, opt => opt.Ignore())
            .ForMember(p => p.MinuteBook, opt => opt.Ignore());
        CreateMap<User, UserModel>();

        CreateMap<UserRoleModel, UserRole>()
            .ForMember(p => p.User, opt => opt.Ignore())
            .ForMember(p => p.Role, opt => opt.Ignore());
        CreateMap<UserRole, UserRoleModel>();

        CreateMap<PlayerModel, Player>()
            .ForMember(p => p.Team, opt => opt.Ignore())
            .ForMember(p => p.Position, opt => opt.Ignore())
            .ForMember(p => p.User, opt => opt.Ignore());
        CreateMap<Player, PlayerModel>();
```

Nakon toga potrebno je sve nove *ModelProfile* klase registrirati u *Startup.cs* klasi:

```
var mappingConfig = new MapperConfiguration(mc =>
{
    mc.AddProfile(new Repository.ModelProfile());
    mc.AddProfile(new Web.Infrastructure.ModelProfile());
});
IMapper mapper = mappingConfig.CreateMapper();
services.AddSingleton(mapper);
```

Kako bi nam uspješno funkcionirao *dependency injection*, u svakom sloju potrebno je definirati *DIModule* kontejner i u njemu registrirati sve parove klasa i *interface-a*, što se može vidjeti u sljedećem kodu.

```
public class DIModule : Module
{
    #region Methods

    protected override void Load(ContainerBuilder builder)
    {
        // Factories
        builder.RegisterType<LookupFactory>().As<ILookupFactory>();

        // Singletons
        builder.RegisterType<CacheProvider>().As<ICacheProvider>().SingleInstance();

        // Services
        builder.RegisterType<UserService>().As<IUserService>();
        builder.RegisterType<PlayerService>().As<IPlayerService>();
        builder.RegisterType<TeamService>().As<ITeamService>();
```

```
        builder.RegisterType<SeasonService>().As<ISeasonService>();
        builder.RegisterType<FixtureService>().As<IFixtureService>();
        builder.RegisterType<GameService>().As<IGameService>();
        builder.RegisterType<MinuteBookService>().As<IMinuteBookService>();
        builder.RegisterType<CoachService>().As<ICoachService>();
        builder.RegisterType<PlayerGameService>().As<IPlayerGameService>();

        // Lookups
        builder.RegisterType<RoleLookup>().As<IRoleLookup>();
        builder.RegisterType<ClubLookup>().As<IClubLookup>();
        builder.RegisterType<PositionLookup>().As<IPositionLookup>();
        builder.RegisterType<MinuteBookEntryTypeLookup>()
                            .As<IMinuteBookEntryTypeLookup>();
        builder.RegisterType<PlayerGameStatusLookup>().As<IPlayerGameStatusLookup>();
    }

    #endregion Methods
}
```

Nakon toga u *Startup.cs* dodati *ConfigureContainer()* metodu koja će sve kontejnere u aplikaciji registrirati prilikom pokretanja aplikacije:

```
public void ConfigureContainer(ContainerBuilder builder)
{
    builder.RegisterModule(new Infrastructure.DIModule());
    builder.RegisterModule(new Service.DIModule());
    builder.RegisterModule(new Repository.DIModule());
    builder.RegisterModule(new Common.DIModule());
}
```

Jedna od važnijih komponenata *ASP.NET* aplikacije jest definiranje putanja (engl. *routes*) koje služe za uspješnu navigaciju kroz aplikaciju. U *Startup.cs* datoteci može se definirati standardna putanja, koja je prikazana u sljedećem kodu.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Ona spaja putanju koja se upiše u preglednik sa kontrolorom i njegovom akcijom na sljedeći način:

- Putanja /Home/Index pozvat će *Home* kontrolor i njegovu akciju *Index()*
- Putanja /Game/EditGame/{id:int} pozvat će *Game* kontrolor i predat će podatak *id* tipa *integer* njegovoj akciji *EditGame()*
- Ukoliko ništa nije upisano u putanju standardno će se pozvati *Home* kontrolor i njegova akcija *Index()*
- *{id:int?}* označava opcionalni podatak

U sljedećem kodu može se vidjeti *EditGame()* akcija *Game* kontrolora. Moguće joj je pristupiti samo preko *HTTP GET* zahtjeva. Ona kreira novi *CreateGameViewModel*, puni ga pomoću *Entity Framework-a* i *LINQ-a* tako da dohvati zadanu utakmicu iz baze, kopira ju u *ViewModel*, dohvati *Club* lookup te predaje pogledu.

```
[HttpGet]
public async Task<IActionResult> EditGame(Guid id)
{
    var poco = await GameService.GetAsync(id, "Fixture.Season");

    if (poco == null)
    {
        return RedirectToAction("Index");
    }

    var viewModel = Mapper.Map<CreateGameViewModel>(poco);
    viewModel.Fixture = Mapper.Map<RESTFixture>(poco.Fixture);
    viewModel.Clubs = (await LookupFactory.Get<IClubLookup>().FindAllAsync())
                      .ToSelectList();

    return View("CreateGame", viewModel);
}
```

U *ASP.NET-u* prati se konvencija naziva za pojedine poglede. Naziv pogleda trebao bi odgovarati nazivu akcije kontrolora i biti smješten u mapi koja ima naziv kontrolora. U tom slučaju, u kontroloru, pri pozivu metode *return View()*, ne moramo ništa dodatno predavati metodi, već će ona sama pozvati odgovarajući pogled.

U pogledima koristi se kombinacija *HTML* jezika i *Razor markup* jezika. Svaki pogled smije imati deklariran samo jedan *ViewModel* kao što je prikazano u sljedećem kodu.

```
@model CreateGameViewModel
@using FootballLeague.Common

@{
    string referer = Context.Request.Headers["Referer"].ToString();
    string gameId = Context.Request.RouteValues["id"]?.ToString() ?? string.Empty;
    bool isEdit = Context.Request.Path.Value.Contains("edit",
                      StringComparison.InvariantCultureIgnoreCase);
    string buttonText = isEdit ? "Update" : "Create";
    string title = isEdit ? "Edit Game" : "Add Game";
}

<div class="container mb-5">
    <div class="row">
        <b>
            Fixture @if (!isEdit)
            {@Html.ActionLink("Select fixture", "SelectFixture", "Game",
                              htmlAttributes: new { @class = "btn btn-primary btn-sm" })}
        </b>
    </div>
    <div class="row">Round Number: @Model.Fixture.RoundNumber</div>
    <div class="row">Season: @Model.Fixture.Season.Name</div>
</div>
```

Kod *ViewModel-a* koji se koriste za dohvaćanje brojive liste podataka potrebno je samo napomenuti generičke metode za navigaciju kroz stranice brojive liste i sortiranje podataka, kao što je prikazano u sljedećem kodu.

```
public void Initialize()
{
    if (CurrentPage == Page)
    {
        if (CurrentSort == SortBy)
        {
            Descending = !Descending;
        }
        else
        {
            Descending = false;
        }
    }

    CurrentSort = SortBy;
    CurrentPage = Page;
}

public T UpdatePage<T>(T model, int page) where T : SearchViewModel
{
    var clone = model.MemberwiseClone() as T;
    clone.Page = page;

    return clone;
}

public T UpdateSortBy<T>(T model, string sortBy) where T : SearchViewModel
{
    var clone = model.MemberwiseClone() as T;
    clone.SortBy = sortBy;

    return clone;
}
```

Primjer korištenja je prikazan u sljedećem kodu, gdje se metode koriste skupa sa *Razor* metodama za kreiranje *HTML* elemenata za *link*.

```
<thead>
    <tr>
        <th>@Html.ActionLink("First Name", "Index",
searchViewModel.UpdateSortBy<PlayerSearchViewModel>(searchViewModel, "User.FirstName"))
        </th>
        <th>@Html.ActionLink("Last Name", "Index",
searchViewModel.UpdateSortBy<PlayerSearchViewModel>(searchViewModel, "User.LastName"))
        </th>
   </tr>
</thead>
```

4.6 Ostali slojevi

Model sloj sadržava sve domenske modele koji se koriste kroz cijeli projekt. Ovaj sloj je vidljiv svim ostalim slojevima osim *Common* sloja.

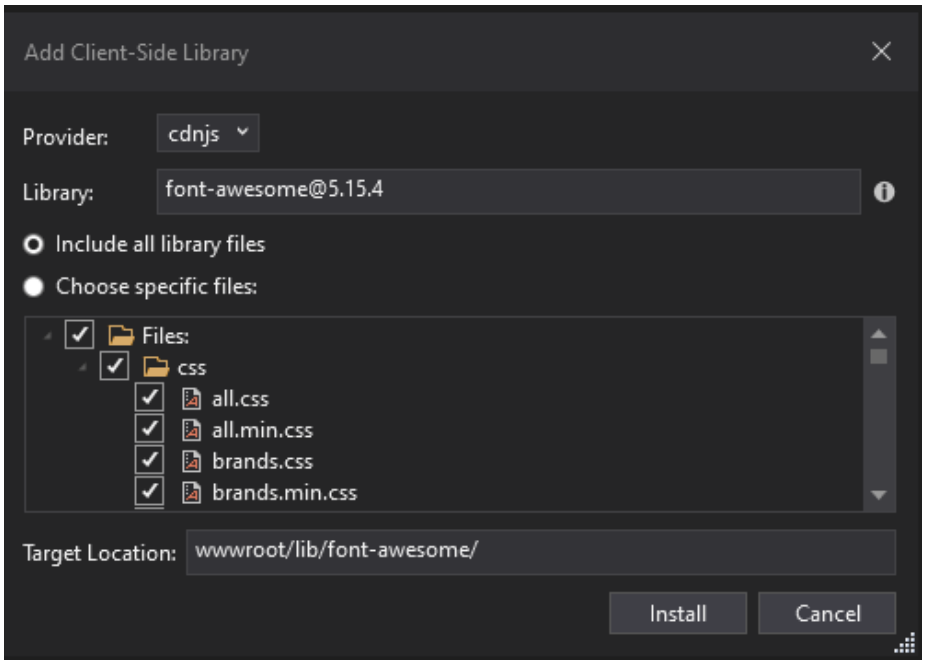
Common sloj sadržava sav kod koji se može koristiti u svim ostalim slojevima, uključujući filtere za tablice, *AsyncLazy* klase, proširenja za *DateTime* klasu i neke generičke klase poput *Pager* i *Sorter*.

5 PRIMJENA APLIKACIJE ZA ADMINISTRACIJU LIGE

Aplikacija se može podijeliti na dva glavna dijela, javni dio kojem mogu pristupiti svi korisnici aplikacije te administrativni dio kojem mogu pristupiti samo autorizirani korisnici. Kod administrativnog dijela pristup određenim funkcionalnostima i pogledima ograničen je na osnovu korisnikove uloge u aplikaciji. Te uloge mogu biti:

- Administrator
- Igrač
- Zapisničar
- Sudac
- Trener kluba
- Administrator kluba

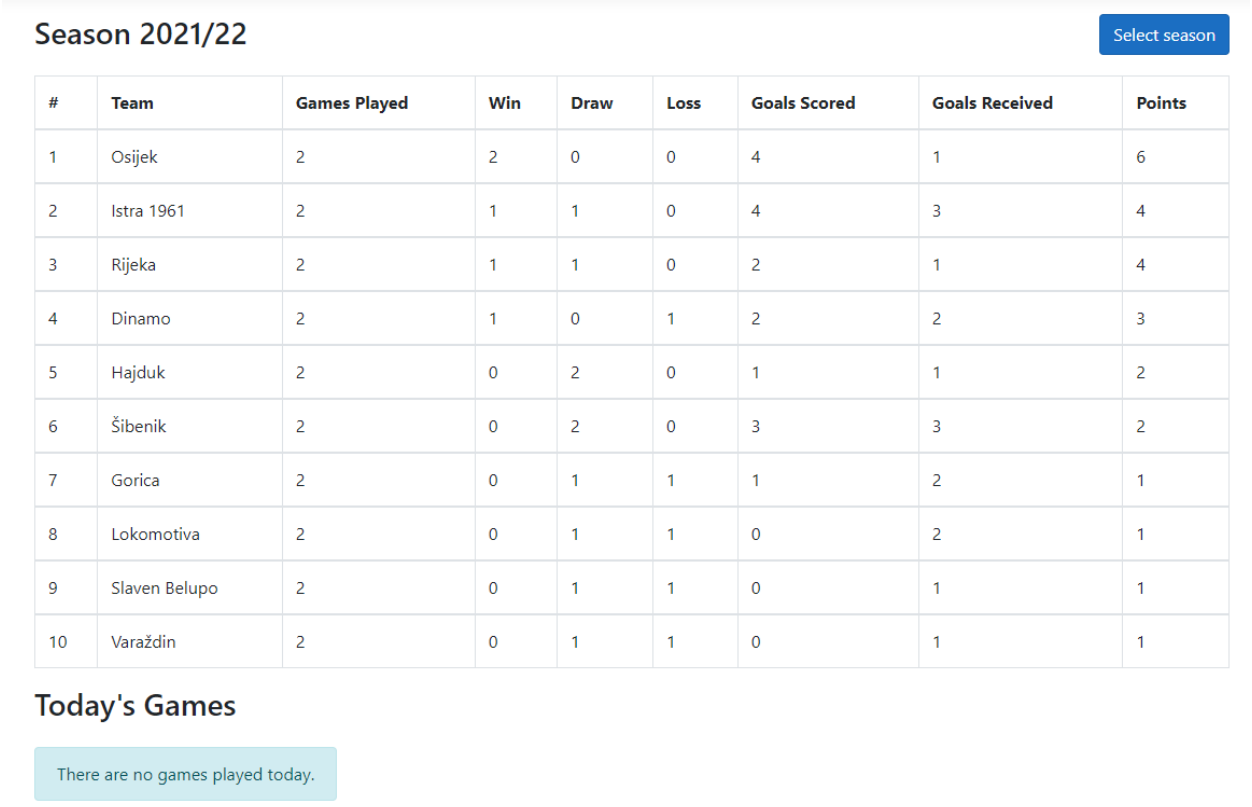
Kod dizajna aplikacije korištena je biblioteka *Bootstrap* [17], točnije verzija 4.3.1, koja dolazi u paketu sa ASP.NET Core MVC 3.1 predloškom, dok je za potrebe ikona korištena biblioteka *Font Awesome* [18]. Jedna od prednosti korištenja *Visual Studio Community 2019* razvojnog okruženja jest ugrađena funkcionalnost za dohvaćanje željenih biblioteka sa popularnih servisa, poput *cdnjs*, bez potrebe za korištenjem vanjskih alata poput *npm* ili *yarn*. Sve što je potrebno jest kliknuti desnim klikom miša na ASP.NET Core MVC projekt, odabrati *Add*, zatim *Client-Side Library* i potražiti željenu biblioteku (slika 5.1.).



Sl. 5.1. Dohvaćanje biblioteka u MVC projektu

5.1 Javni dio aplikacije

Na početnoj stranici aplikacije prikazana je tablica sa poretkom timova i njihovim rezultatima u trenutnoj sezoni, te popis utakmica koje bi se trebale održati danas (slika 5.2.).



#	Team	Games Played	Win	Draw	Loss	Goals Scored	Goals Received	Points
1	Osijek	2	2	0	0	4	1	6
2	Istra 1961	2	1	1	0	4	3	4
3	Rijeka	2	1	1	0	2	1	4
4	Dinamo	2	1	0	1	2	2	3
5	Hajduk	2	0	2	0	1	1	2
6	Šibenik	2	0	2	0	3	3	2
7	Gorica	2	0	1	1	1	2	1
8	Lokomotiva	2	0	1	1	0	2	1
9	Slaven Belupo	2	0	1	1	0	1	1
10	Varaždin	2	0	1	1	0	1	1

Sl. 5.2. Tablica trenutne sezone

Pritiskom na gumb *Select season* na slici 5.2. bilo koji korisnik može promijeniti trenutnu sezonu te vidjeti rezultate tablice za odabranu sezonu.

Dohvaćanje podataka za sezonsku tablicu jedan je od primjera u kojem se mogu vidjeti koncepti poput odvajanja poslovne logike u pripadajući servis, korištenja zasebnih domenskih modela koji nemaju sebi slične *entity* modele te korištenja *Automapper-a*.

Kada korisnik pošalje zahtjev za dobivanjem pogleda koji prikazuje tablicu lige, poziva se akcija *Index* od kontrolora *Home*. Ukoliko korisnik nije promijenio sezonu, uzima se sezona koja je u sustavu označena kao trenutna, kao što je prikazano u sljedećem kodu.

```
public async Task<IActionResult> Index(Guid? seasonId = null)
{
    SeasonModel season;
    if (seasonId.HasValue)
    {
        season = await SeasonService.GetAsync(seasonId.Value);
    }
    else
    {
        season = await SeasonService.GetCurrentSeasonAsync();
    }
```

Nakon toga kreira se *ViewModel* za pogled koji sadržava podatke o sezoni, tablici i utakmicama koje se trebaju danas održati:

```
var viewModel = new TableViewModel()
{
    Season = Mapper.Map<RESTSeason>(season),
    Table = Mapper.Map<RESTTable>(await TeamService.GetTableAsync(season.Id)),
    Games = Mapper.Map<IEnumerable<RESTGame>>(await GameService.FindAllAsync(gameFilter,
            new Sorter("StartDate|desc"), "HomeTeam.Club,AwayTeam.Club"))
};
```

Pri kreiranju *ViewModel-a* poziva se metoda za dohvaćanje podataka za popunjavanje tablice *GetTableAsync()* od *TeamService* servisa. U prvom dijelu metode dohvaćaju se svi timovi iz baze podataka zajedno sa svojim utakmicama, te se rade osnovne operacije zbrajanja postignutih i dobivenih golova, zbroj pobijeđenih, izgubljenih i izjednačenih utakmica, što je prikazano u sljedećem kodu.

```
foreach (var team in teams)
{
    var teamSeasonResult = new TeamSeasonResultModel()
    {
        Team = team
    };

    var filter = new GameFilter()
    {
        AwayTeamId = team.Id,
        GameFinished = true,
        FixtureFilter = new FixtureFilter()
        {
            SeasonId = seasonId
        }
    };

    var awayGames = await GameRepository.FindAllGamesAsync(filter);

    filter.AwayTeamId = null;
    filter.HomeTeamId = team.Id;
    var homeGames = await GameRepository.FindAllGamesAsync(filter);

    teamSeasonResult.GamesWon = awayGames.Where(p => p.AwayGoals > p.HomeGoals).Count() +
                                homeGames.Where(p => p.HomeGoals > p.AwayGoals).Count();
    teamSeasonResult.GamesDraw = awayGames.Where(p => p.AwayGoals == p.HomeGoals).Count()
                               + homeGames.Where(p => p.HomeGoals == p.AwayGoals).Count();
```

```
    teamSeasonResult.GamesLost = awayGames.Where(p => p.AwayGoals < p.HomeGoals).Count()
                               + homeGames.Where(p => p.HomeGoals < p.AwayGoals).Count();

    teamSeasonResult.GoalsScored = awayGames.Sum(p => p.AwayGoals.Value) +
                                   homeGames.Sum(p => p.HomeGoals.Value);
    teamSeasonResult.GoalsReceived = awayGames.Sum(p => p.HomeGoals.Value) +
                                     homeGames.Sum(p => p.AwayGoals.Value);

    result.TeamSeasonResult.Add(teamSeasonResult);
}
```

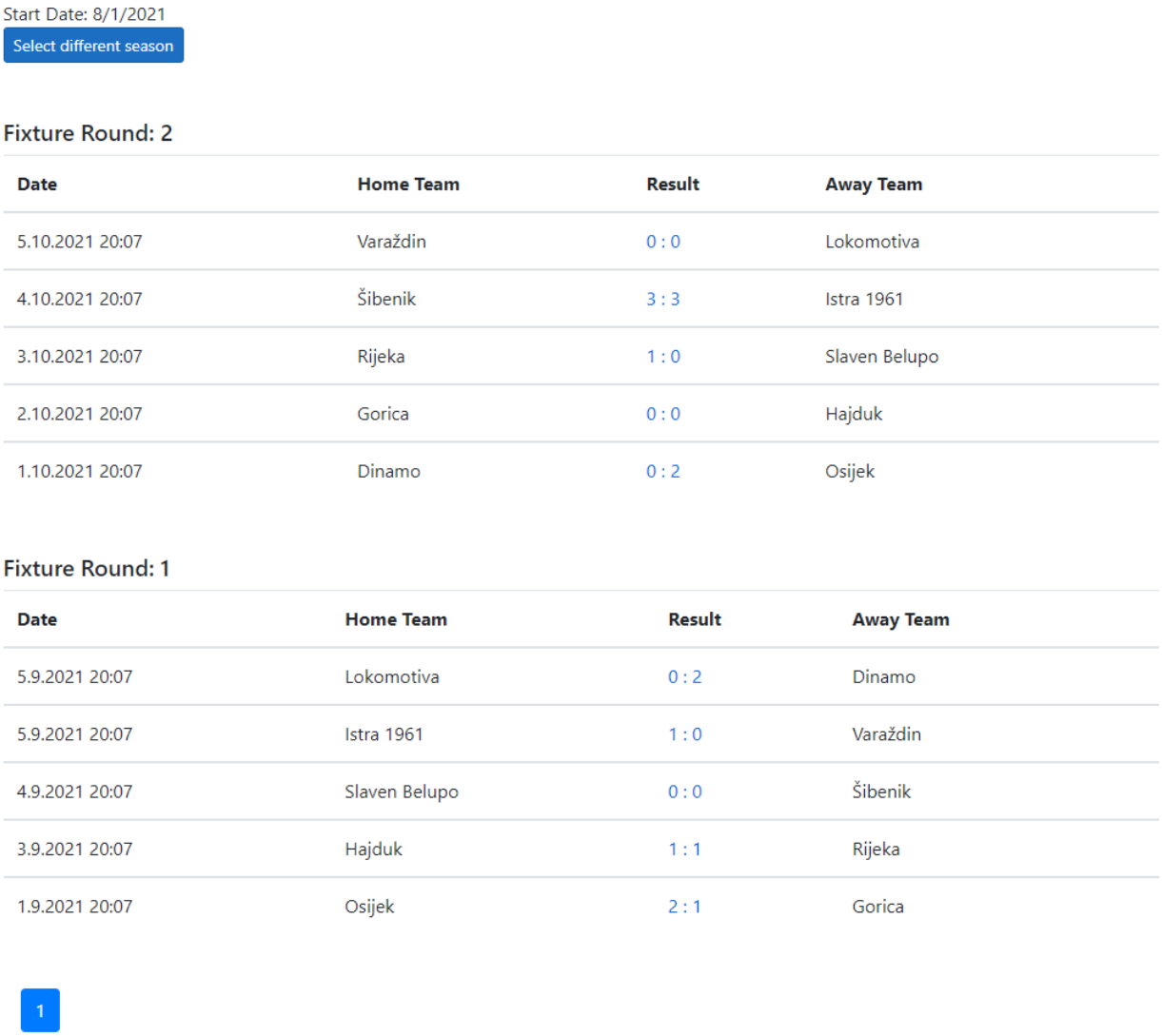
Drugi dio metode *GetTableAsync()* sastoji se od određivanja pozicije na tablici u ovisnosti o broju bodova, međusobnom omjeru pobjeda između timova sa istim brojem osvojenih bodova, zatim gol razlici, a u slučaju izjednačenosti tri kluba ili više, broju postignutih golova.

Tijek metode može se vidjeti u sljedećem kodu. Lista klubova prvo se poreda po broju postignutih bodova od najvišeg prema najmanjem, te se potom grupira po broju postignutih bodova. Zatim se *foreach* petljom prolazi kroz svaki grupirani redak i u slučaju da dva ili više klubova imaju isti broj bodova, određuje im se pozicija na već spomenuti način.

```
private async Task ProcessTeamPositionsAsync(List<TeamSeasonResultModel> list, Guid
seasonId)
{
    var position = 1;
    var groupedByPoints = list.OrderByDescending(p => p.Points).GroupBy(p => p.Points);

    foreach (var group in groupedByPoints)
    {
        if (group.Count() > 1)
        {
            var teamMatchups = new List<TeamMatchupModel>();

            foreach (var team in group)
            {
                var currentTeamId = team.Team.Id;
                var filter = new GameFilter()
                {
                    TeamMatchups = new Tuple<Guid, Guid[]>(currentTeamId, Group
                              .Where(s => s.Team.Id != currentTeamId)
                              .Select(p => p.Team.Id).ToArray()),
                    GameFinished = true,
                    FixtureFilter = new FixtureFilter()
                    {
                        SeasonId = seasonId
                    }
                };

                var games = await GameRepository.FindAllGamesAsync(filter);

                var teamMatchup = new TeamMatchupModel(currentTeamId);

                var awayGoals = games.Where(s => s.AwayTeamId == currentTeamId)
                                     .Sum(p => p.AwayGoals.GetValueOrDefault());
                var homeGoals = games.Where(s => s.HomeTeamId == currentTeamId)
                                     .Sum(p => p.HomeGoals.GetValueOrDefault());
```

```
                var goalsReceived = games.Where(s => s.AwayTeamId == currentTeamId)
                                         .Sum(p => p.HomeGoals.GetValueOrDefault()) +
                                    games.Where(s => s.HomeTeamId == currentTeamId)
                                         .Sum(p => p.AwayGoals.GetValueOrDefault());

                teamMatchup.GoalsScored = awayGoals + homeGoals;
                teamMatchup.AwayGoals = awayGoals;
                teamMatchup.GoalDifference = awayGoals + homeGoals - goalsReceived;
                teamMatchup.NumberOfWins = games.Count(p =>
                    (p.HomeTeamId == currentTeamId && p.HomeGoals > p.AwayGoals) ||
                    (p.AwayTeamId == currentTeamId && p.AwayGoals > p.HomeGoals));

                teamMatchups.Add(teamMatchup);
            }

            var orderedTeamMathchups = teamMatchups
                                           .OrderByDescending(p => p.NumberOfWins)
                                           .ThenByDescending(s => s.GoalDifference);
            if (group.Count() > 2)
            {
                orderedTeamMathchups = orderedTeamMathchups.ThenBy(p => p.GoalsScored);
            }
            else
            {
                orderedTeamMathchups = orderedTeamMathchups.ThenBy(p => p.AwayGoals);
            }

            foreach (var orderedTeam in orderedTeamMathchups)
            {
                group.First(p => p.Team.Id == orderedTeam.TeamId)
                     .TablePosition = position;
                position++;
            }

        }
        else
        {
            group.First().TablePosition = position;
            position++;
        }
    }
}
```

Za javne korisnike aplikacije pogled za prikazivanje utakmica izgleda kao na slici 5.3. Sve što javni korisnici mogu vidjeti jest popis utakmica grupiranih po kolima koji su prikazani u straničnoj listi od maksimalno dvije stavke po stranici. Na tom pogledu korisnici mogu, pritiskom na poveznicu koja se nalazi na rezultatu pojedine utakmice, otvoriti novi pogled u kojem su prikazani detalji te utakmice.



Sl. 5.3. Izgled popisa utakmica za javne posjetitelje

Otvaranjem pogleda za detalje o utakmici korisnik može saznati informacije o odabranoj utakmici, uključujući rezultat, popis igrača, golova, kartona i službeni zapisnik. Pogled je prikazan na slici 5.4.

Game: Osijek 2 : 1 Gorica

Stadium: Gradski Vrt

Official: Test1 Official

Referee: Test1 Ref

	Minute	Player	Team
⚽	25	Pero Perić	Osijek
⚽	58	Pero Perić	Osijek
⚽	90	Dino Dinić	Gorica

	Minute	Player	Team
🟨	30	Anto Antić	Gorica
🟨	40	Ivo Ivić	Osijek

Minute Book

First Half

Time	Entry	Player
11	Foul committed	Ivo Ivić
18	Handball	Dino Dinić
25	Goal scored	Pero Perić
30	Yellow card received	Anto Antić
45	Yellow card received	Ivo Ivić

Second Half

Time	Entry	Player
58	Goal scored	Pero Perić
60	Offside	Pero Perić
74	Substitution	Ivo Ivić (Andro Andrić)
84	Foul committed	Marko Markić
90	Goal scored	Dino Dinić

Sl. 5.4. Izgled pogleda za detalje utakmice

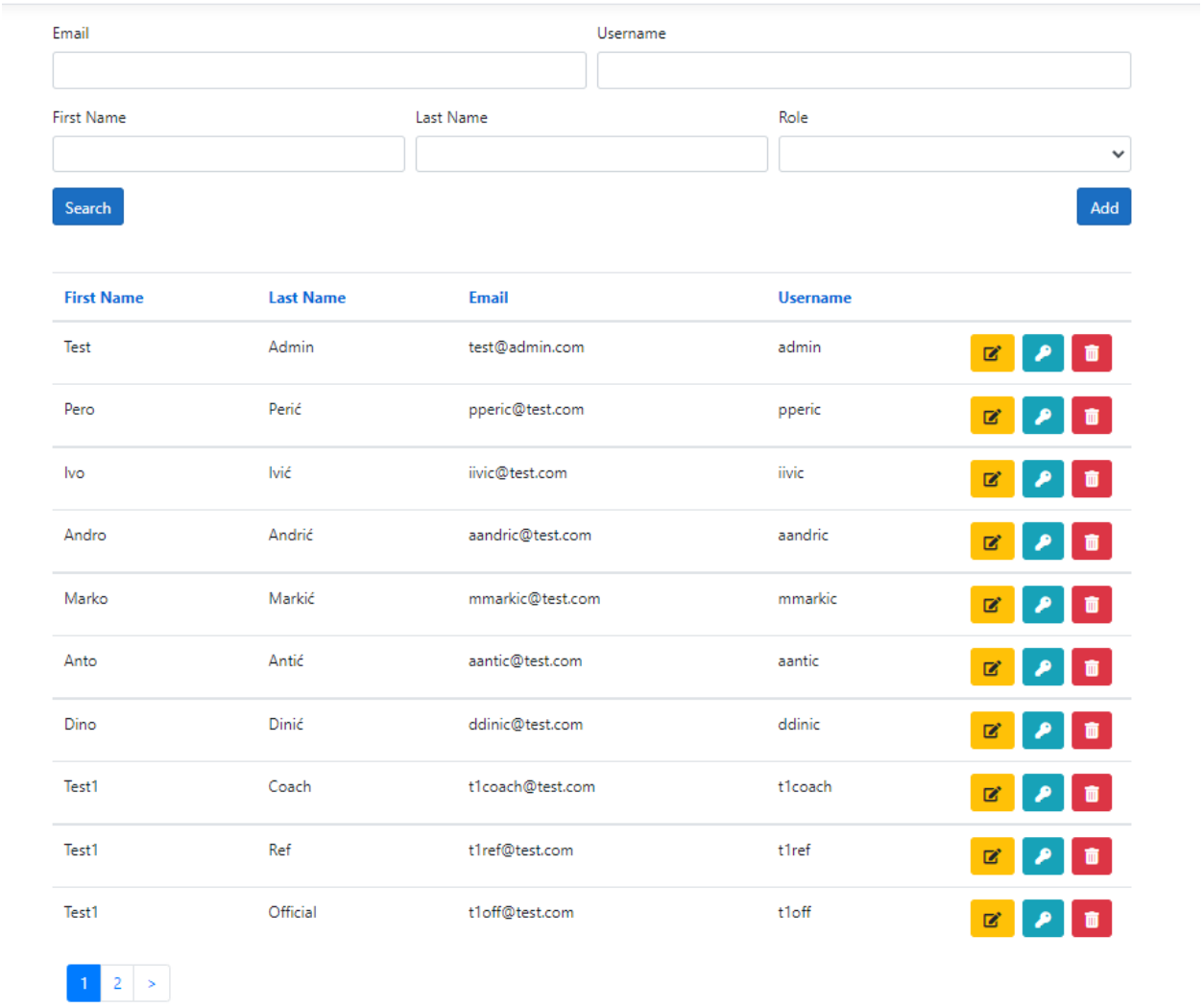
Zadnja dva pogleda koja su dostupna javnim korisnicima su pogled za profil igrača, gdje se mogu vidjeti informacije o trenutnom klubu, statistika za svaku sezonu za postignute golove i dobivene žute i crvene kartone te rezultati posljednjih 10 utakmica od trenutnog kluba, i pogled za profil kluba gdje se mogu vidjeti rezultati posljednjih 10 utakmica tog kluba, statistika za sve sezone, popis klubova protiv kojih trenutni klub ima najbolji omjer pobjeda i poraza te popis klubova protiv kojih trenutni klub ima najgori omjer pobjeda i poraza.

5.2 Administrativni dio aplikacije

Nakon uspješne autorizacije, autoriziranom korisniku prikazat će se pogled s osnovnim informacijama o korisniku te ovisno o ulozi u aplikaciji neke njemu specifične informacije. Igračima će biti prikazan trenutni valjani ugovor sa klubom, osnovna ovosezonska statistika, odigrane i nadolazeće utakmice te otvorene primjedbe za utakmicu. Sucima, zapisničarima i trenerima bit će prikazane njihove odigrane i nadolazeće utakmice te otvorene primjedbe za njihove utakmice, dok će se za administratore i administratore tima prikazivati samo osnovne osobne informacije uz mogućnost uređivanja tih informacija.

Samo administrator ima pristup pogledu gdje se mogu vidjeti, dodavati, uređivati i brisati korisnici te im mijenjati lozinka. Pogled se sastoji od standardne *HTML* forme za filtriranje

podataka, tablice u kojoj su prikazani podaci o korisnicima i kontrole za paginaciju. Podaci su prikazani u straničnoj listi od maksimalno deset stavki po stranici (slika 5.5.).



Sl. 5.5. Uređivanje korisnika aplikacije

Kod pogleda za prikaz korisnika važno je istaknuti na koji način funkcionira sortiranje dobivenih podataka i kako se može prolaziti kroz različite stranice u straničnoj listi. Kao glavni *ViewModel* vezan za ovaj pogled korišten je *UserSearchViewModel* koji nasljeđuje osnovni model *SearchViewModel* i njegova svojstva:

- *Page* – kazuje našem kodu koju stranicu iz stranične liste korisnik zahtjeva.
- *SortBy* – kazuje našem kodu prema kojem svojstvu bi se stranična lista korisnika trebala sortirati.

Uz svojstva, *UserSearchViewModel* nasljeđuje i dvije metode prikazane u sljedećem kodu, koje se koriste za ažuriranje gore navedenih svojstava.

```
public T UpdatePage<T>(T model, int page) where T : SearchViewModel
{
    var clone = model.MemberwiseClone() as T;
    clone.Page = page;

    return clone;
}

public T UpdateSortBy<T>(T model, string sortBy) where T : SearchViewModel
{
    var clone = model.MemberwiseClone() as T;
    clone.SortBy = sortBy;

    return clone;
}
```

Primjer implementacije metode za sortiranje prikazan je u sljedećem kodu.

```
<th>@Html.ActionLink("First Name", "Index",
searchViewModel.UpdateSortBy<UserSearchViewModel>(searchViewModel, "FirstName"))</th>
<th>@Html.ActionLink("Last Name", "Index",
searchViewModel.UpdateSortBy<UserSearchViewModel>(searchViewModel, "LastName"))</th>
<th>@Html.ActionLink("Email", "Index",
searchViewModel.UpdateSortBy<UserSearchViewModel>(searchViewModel, "Email"))</th>
<th>@Html.ActionLink("Username", "Index",
searchViewModel.UpdateSortBy<UserSearchViewModel>(searchViewModel, "UserName"))</th>
```

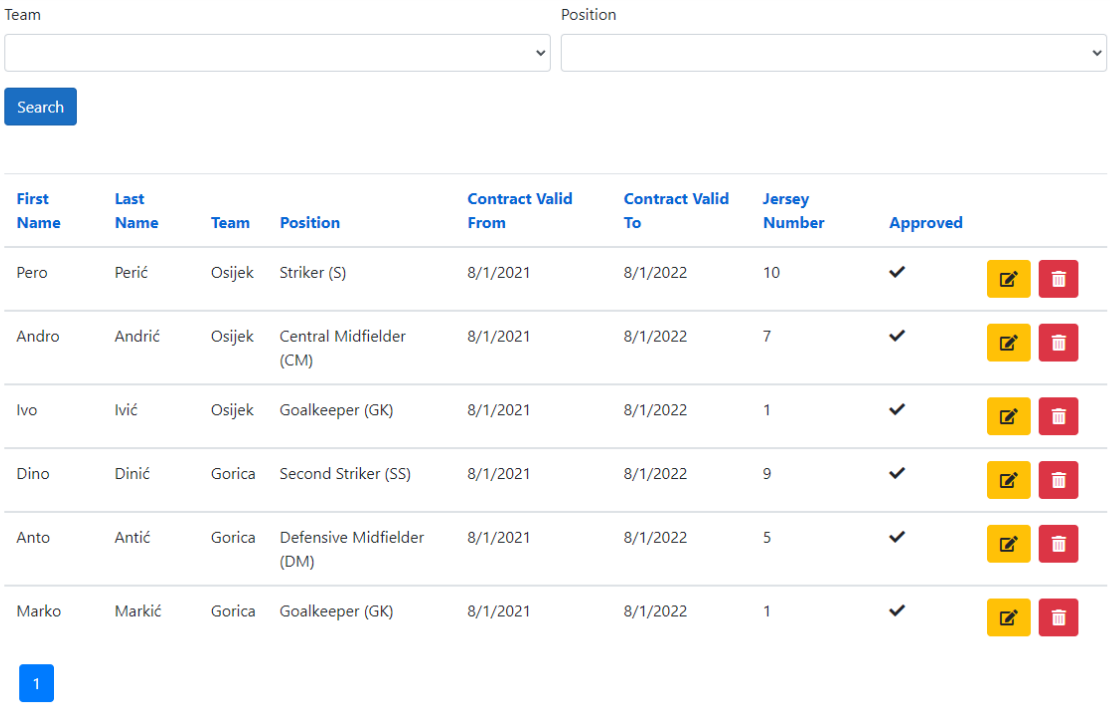
Implementacija za prolazak kroz različite stranice u straničnoj listi prikazana je u sljedećem kodu.

```
@Html.PagedListPager(Model.Users, p => Url.Action("Index",
searchViewModel.UpdatePage<UserSearchViewModel>(searchViewModel, p)), new
PagedListRenderOptions()
{
    ContainerDivClasses = new[] { "container" },
    UlElementClasses = new[] { "pagination" },
    LiElementClasses = new[] { "page-item" },
    PageClasses = new[] { "page-link" }
})
```

Kod sortiranja korištene su pomoćne *HTML* metode za kreiranje hiperlinka koji za treći parametar prima objekt sa svojstvima za putanju. Ukoliko želimo, kao u ovom primjeru, imati mogućnost sortirati sa više različitih svojstava, moramo biti sigurni da objekt koji prosljeđujemo pomoćnoj *HTML* metodi je novi objekt, a ne referenca na već postojeći objekt. Upravo zato moramo koristiti *MemberwiseClone()* metodu od *SearchViewModel*-a kako bi kreirali novu instancu *UserSearchViewModel*-a

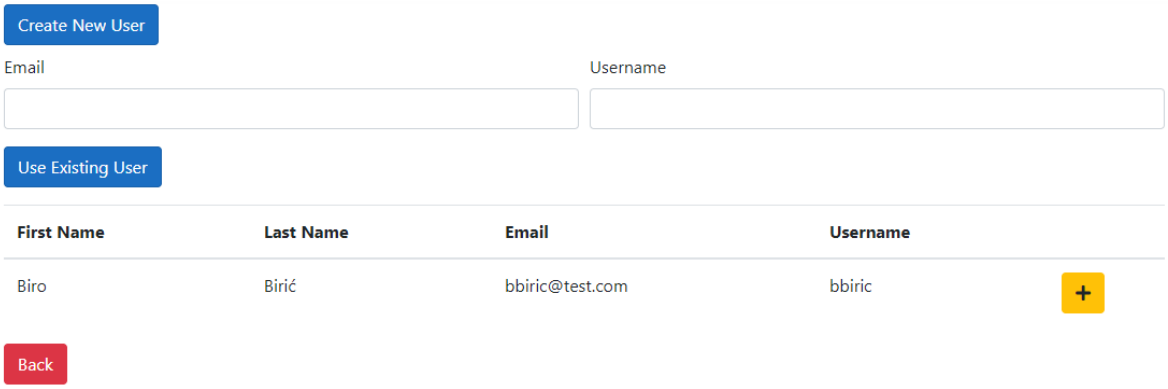
Postupak kreiranja igrača se svodi na kreiranje korisnika u bazi sa ulogom igrač, ukoliko taj korisnik već ne postoji, te kreiranja ugovora između odabranog kluba i igrača na određeni vremenski period. Cijeli odjeljak za rukovanjem igrača i njihovih ugovora podijeljen je na četiri pogleda:

- Pogled za pretraživanje već postojećih igrača i ugovora.
- Pogled za kreiranje potpuno novog igrača i njegovog prvog ugovora.
- Pogled za ponovnu registraciju ili obnovu ugovora već postojećih igrača.
- Pogled za potvrdu registracije.

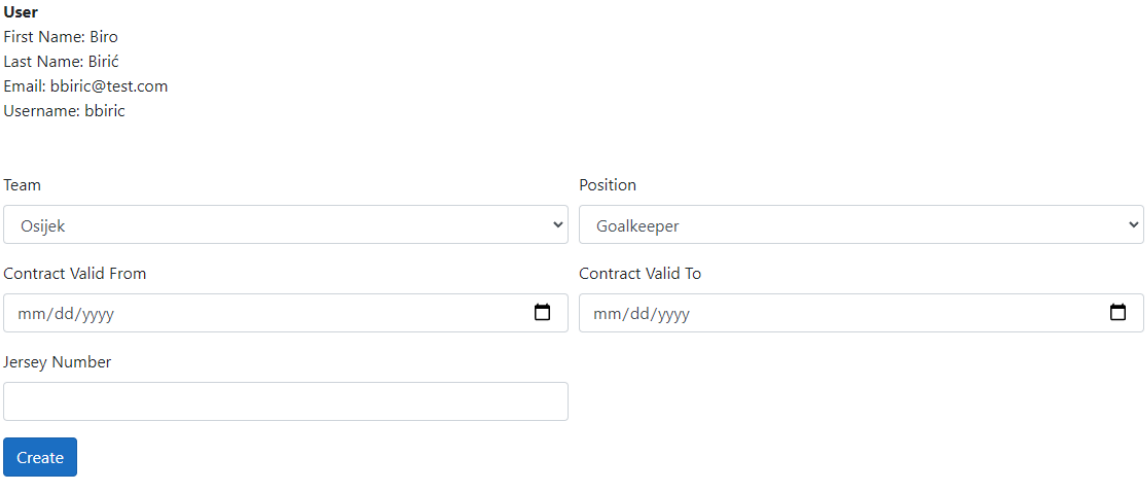


Sl. 5.6. Pogled za pretraživanje već postojećih igrača i ugovora

Na slici 5.6. prikazan je pogled za pretraživanje već postojećih igrača i ugovora. Na slikama 5.7. i 5.8. mogu se vidjeti pogledi za kreiranje novog igrača. Kod kreiranja imamo mogućnost koristiti već postojećeg korisnika ili kreirati novog. Pogled za registraciju već postojećeg igrača isti je kao na slici 5.8., najveća razlika među njima je to što se u slučaju registracije postojećeg igrača rade provjere kako se igračev ugovor ne bi preklapao sa već postojećim ugovorom. Ovim pogledima pristup imaju administratori i administratori kluba.

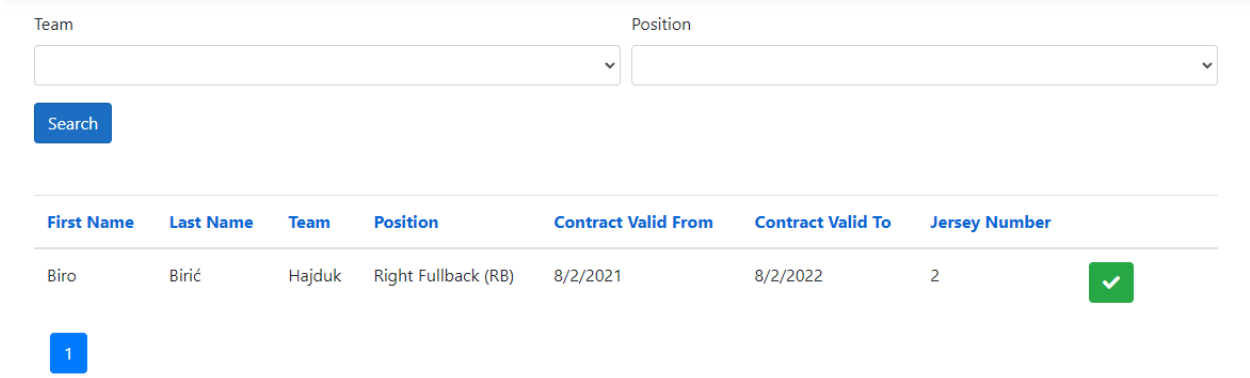


Sl. 5.7. Pogled za kreiranje novog igrača



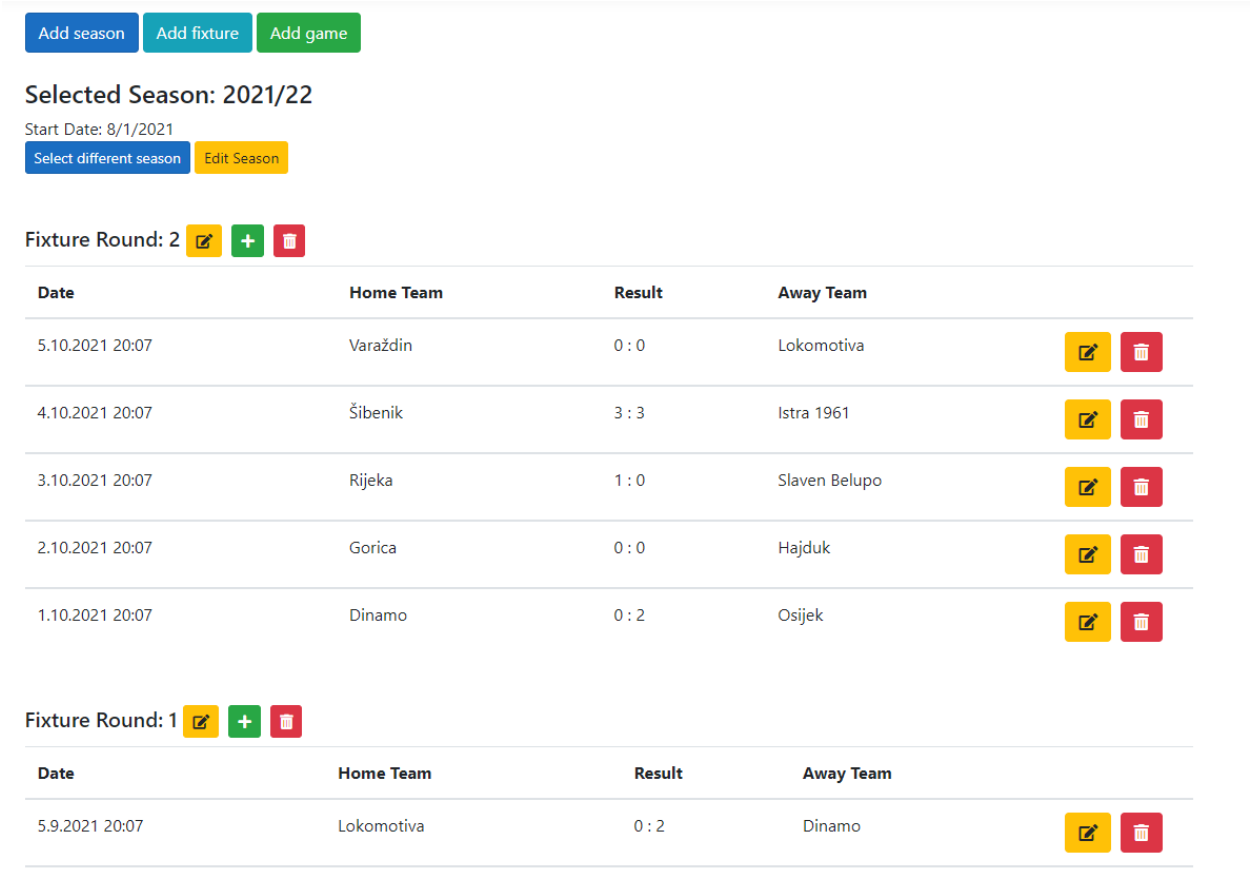
Sl. 5.8. Pogled za registraciju novog igrača

Zadnji pogled je pogled za potvrdu registracije, njemu imaju pristup samo administratori. Nakon što se registracija igrača dovrši potrebno ju je odobriti kako bi igrač bio dostupan treneru pri registraciji momčadi za pojedinu utakmicu (slika 5.9.). Postupak za kreiranje trenera za pojedinu momčad vrlo je sličan postupku kreiranja igrača.

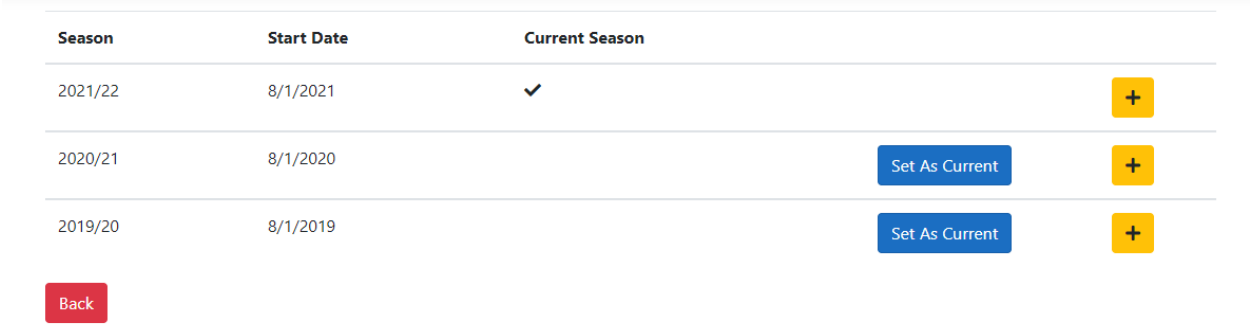


Sl. 5.9. Pogled za potvrdu registracije

Na pogledu za prikazivanje utakmica (slika 5.10.) administratorima je ponuđeno nekoliko dodatnih funkcionalnosti: dodavanje i uređivanje sezone te dodavanje, uređivanje i brisanje kola i utakmica. Još jedna od važnijih stvari jest mogućnost postavljanja trenutne sezone za cijeli sustav klikom na gumb *Select different season* te odabirom željene sezone (slika 5.11.).



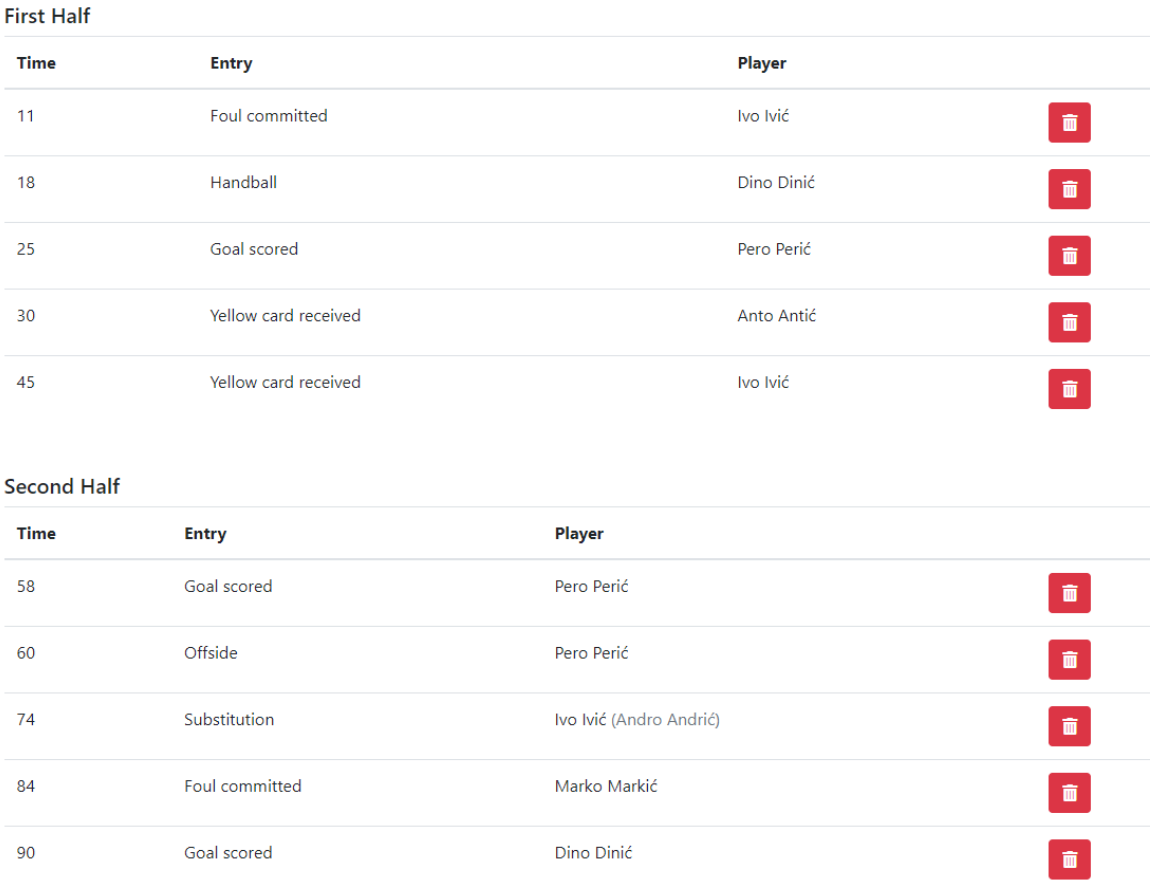
Sl. 5.10. Pogled za prikazivanje utakmica



Sl. 5.11. Pogled za postavljanje trenutne sezone za cijeli sustav

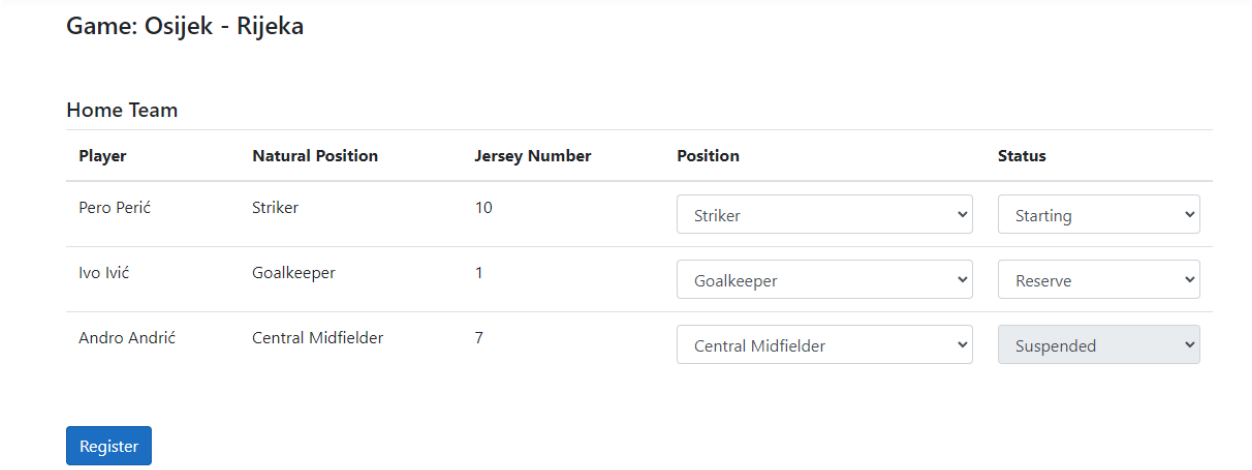
Ulaskom na pogled za detalje završene utakmice, zapisničari koji su sudjelovali u toj utakmici imat će mogućnost dodati zapisnik i njegove detalje. Na slici 5.12. imamo mogućnost vidjeti kako izgleda već popunjen zapisnik. Važno je napomenuti da će sustav prilikom spremanja

zapisnika u bazu voditi računa o broju žutih i crvenih kartona za sve igrače te zabilježiti suspenziju igrača ukoliko je to potrebno.



Sl. 5.12. Popunjeni zapisnik

Na pogledu za detalje utakmice koja još nije počela, treneri klubova koji sudjeluju u utakmici imat će priliku registrirati igrače za tu utakmicu (slika 5.13.).



Sl. 5.13. Registriranje igrača

Za svakog igrača morat će se odabrati pozicija koju će igrati tu utakmicu i status, koji može biti:

- Početna postava
- Rezerva
- Ozlijeđen
- Suspendiran
- Odsutan

Ukoliko je sustav prepoznao da je igrač zaradio suspenziju u prijašnjim utakmicama, automatski će se suspenzija primijeniti na iduću utakmicu.

Posljednji pogled namijenjen je za pisanje primjedbi za utakmicu, suđenje i službene osobe povezane sa utakmicom. Svaka registrirana osoba će moći napisati primjedbu za bilo koju utakmicu, no samo će administratori i zapisničar koji je bio prisutan na toj utakmici moći odgovoriti na primjedbe, i samo će administratori moći zatvoriti cijeli slučaj.

6 ZAKLJUČAK

Cilj ovoga rada bio je napraviti *server-side* aplikaciju na način koji omogućuje skalabilnost aplikacije u širinu i visinu, istovremeno omogućujući podjelu koda na smislene slojeve koji se mogu mijenjati po potrebi bez prevelikog utjecaja na sloj iznad njih. Višeslojna *Onion* arhitektura upravo to omogućuje.

Kod baza podataka vrlo je lako prebaciti se na neku drugu vrstu, poput *MSSQL-a,* dokle god su tablice i tipovi podatka isti. Sve što je potrebno je zamijeniti neke od *NuGet* paketa.

Entity Framework Core moguće je zamijeniti nekim drugim sofisticiranijim *ORM-om* poput *LLBLGen-a.* Ovisno o tom novom *ORM-u* i njegovom kompatibilnošću sa *LINQ-om,* kod u *repository* sloju bi se morao mijenjati, no najbitnije je da taj kod obavlja upravo ono što je bio obavljao i prije, kako ne bi imao nikakav utjecaj na poslovnu logiku u *service* sloju. Upravo ta mogućnost zamjene *ORM-a* je jedan od najvećih razloga za odvajanje poslovne logike u *service* sloj.

U ovom slučaju, s obzirom da se radi o *server-side* aplikaciji, kao najviši sloj odabrana je *MVC* arhitektura *ASP.NET Core-a*. Iako ima svojih prednosti, za aplikacije ovog tipa, koje su najviše namijenjene krajnjem korisniku i zahtijevaju dosta interakcije, nije najbolje rješenje. Bilo koja kombinacija *JavaScript* aplikacije i *Web API-ja* bi uvelike poboljšala lakoću izrade i lakšu uporabljivost ove aplikacije. Na svu sreću, kod *MVC* arhitekture *ASP.NET Core-a* vrlo je jednostavno prebaciti kontrolore i njihove akcije u *API endpoint-e Web API-ja* te kreirati zaseban *JavaScript* projekt koji će taj *Web API* koristiti.

LITERATURA

[1] Rezultati.com, »Rezultati.com,« [Mrežno]. Available: https://www.rezultati.com/.

[2] SofaScore, »SofaScore,« [Mrežno]. Available: https://www.sofascore.com/.

[3] SportsPlus, »sportsplus.app,« [Mrežno]. Available: https://sportsplus.app/.

[4] TeamSnap, »www.teamsnap.com,« [Mrežno]. Available: https://www.teamsnap.com/.

[5] RadoChervenkov, »github.com,« [Mrežno]. Available: https://github.com/RadoChervenkov/Football-League-Management-System.

[6] A. Freeman, Pro ASP.NET Core MVC 2, London: Apress, 2017.

[7] Mozilla, »MVC -MDN Web Docs Glossary: Definitions of Web-related terms,« Mozilla, [Mrežno]. Available: https://developer.mozilla.org/en-US/docs/Glossary/MVC.

[8] S. S. Shekhawat, »www.c-sharpcorner.com,« [Mrežno]. Available: https://www.c-sharpcorner.com/article/onion-architecture-in-asp-net-core-mvc/.

[9] Microsoft, »EF Core,« Microsoft, [Mrežno]. Available: https://docs.microsoft.com/en-us/ef/core/.

[10] »postgresql.org,« [Mrežno]. Available: https://www.postgresql.org/about/.

[11] »automapper.org,« [Mrežno]. Available: https://automapper.org/.

[12] »autofac.io,« [Mrežno]. Available: https://autofac.readthedocs.io/en/latest/getting-started/index.html.

[13] »depesz.com,« [Mrežno]. Available: https://www.depesz.com/2010/03/02/charx-vs-varcharx-vs-varchar-vs-text/.

[14] Microsoft, Microsoft, [Mrežno]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.guid.

[15] »imperva.com,« [Mrežno]. Available: https://www.imperva.com/learn/performance/lazy-loading/.

[16] S. Cleary, »blog.stephencleary.com,« [Mrežno]. Available: https://blog.stephencleary.com/2012/08/asynchronous-lazy-initialization.html.

[17] »getbootstrap.com,« [Mrežno]. Available: https://getbootstrap.com/.

[18] »fontawesome.com,« [Mrežno]. Available: https://fontawesome.com/.

SAŽETAK

Za kreiranje većine aplikacija koje nastaju u današnje vrijeme najčešće se koristi neka vrsta alata za Internet programiranje zbog potrebe za lakšim pristupom tim aplikacijama i povezivanjem velikog broja korisnika na različitim uređajima.

U ovom radu korišten je *ASP.NET Core* web okvir koji omogućuje rad sa različitim operativnim sustavima, bazama podataka i programskim alatima. Zbog veće potražnje za besplatnim softverom otvorenog koda objašnjen je način kako implementirati *PostgreSQL,* sustav objektno-relacijskih baza podataka, sa *ASP.NET Core* web okvirom. Kako bi aplikacija bila lakša za održavanje i nadograđivanje korištena je višeslojna arhitektura *Onion,* koja dijeli programski kod u određeni broj blisko povezanih slojeva, poput sloja za komunikaciju između servera i baze podataka, kako bi nam omogućila da u budućnosti, ukoliko želimo koristiti drugačiji alat za komunikaciju između servera i baze podataka, možemo ga zamijeniti bez prevelikog utjecaja na ostatak programskog koda poput poslovne logike.

Aplikacija je namijenjena administraciji nogometne lige i trebala bi poslužiti administraciji nižih ili amaterskih liga koje najčešće nemaju gotova softverska rješenja ili su im rješenja nedovoljno razvijena.

Ključne riječi: Administracija nogometne lige, ASP.NET Core, Entity Framework Core, MVC arhitektura, Onion arhitektura.

ABSTRACT

MVC server-side application for monitoring and administration of a football league

To create most of the applications that are being created nowadays, some kind of Internet programming tool is most often used due to the need for easier access to these applications and connecting a large number of users on different devices to them.

In this paper, the ASP.NET Core web framework is used, which enables work with various operating systems, databases and software tools. Due to the growing demand for free open source software, a way to implement PostgreSQL, an object-relational database system, with an ASP.NET Core web framework is explained. To make the application easier to maintain and upgrade, the multi-layer Onion architecture was used, which divides the program code into a number of closely related layers, such as the server-to-database communication layer, to allow us to replace them without too much impact on the rest of the program code, like business logic, if we want to use a different tool in the future.

The application is intended for the administration of a football league and should serve as an administration tool for lower or amateur leagues, which usually do not have a ready-made software solution or their solution is insufficiently developed.

Keywords: ASP.NET Core, Entity Framework Core, Football league administration, MVC architecture, Onion architecture.

ŽIVOTOPIS

Petar Bošnjak, rođen u Osijeku 31.12.1990., s prebivalištem u Osijeku, student je Fakulteta elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. Pohađao je OŠ Dobriša Cesarić i Isusovačku klasičnu gimnaziju u Osijeku. U kolovozu 2018. primljen je u tvrtku MONO d.o.o. kao programer na puno radno vrijeme preko studentskog ugovora o radu. Od siječnja 2020. godine zapošljava se u tvrtki MONO d.o.o. kao stalan zaposlenik. U slobodno vrijeme bavi se web programiranjem, te posjeduje *MCP Exam 361: Software Development Fundamentals* certifikat. Od stranih jezika koristi se engleskim jezikom u govoru i pisanju.

_____

Potpis autora