Web aplikacija za generanje uravnoteženih sportskih ekipa

Vurbić, Ilija

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

Stručni prijediplomski studij Računarstvo

WEB APLIKACIJA ZA GENERIRANJE URAVNOTEŽENIH SPORTSKIH EKIPA

Završni rad

Ilija Vurbić

Osijek, 2024.

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

Obrazac Z1S: Obrazac za ocjenu završnog rada na stručnom prijediplomskom studiju

Ocjena završnog rada na stručnom prijediplomskom studiju

Ime i prezime pristupnika:	Ilija Vurbić
Studij, smjer:	Stručni prijediplomski studij Računarstvo
Mat. br. pristupnika, god.	AR 4829, 27.07.2020.
JMBAG:	0165085367
Mentor:	Marina Peko, dipl. ing. el.
Sumentor:	
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Robert Šojo, univ. mag. ing. comp.
Član Povjerenstva 1:	Marina Peko, dipl. ing. el.
Član Povjerenstva 2:	doc. dr. sc. Ivana Hartmann Tolić
Naslov završnog rada:	Web aplikacija za generanje uravnoteženih sportskih ekipa
Znanstvena grana završnog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada:	Kreirati web aplikaciju koja će imati generator sportskih ekipa za igranje odbojke. Aplikacija ima 2 role: treneri i igrači. Zadatak je kreirati aplikaciju za generiranje uravnoteženih sportskih ekipa na temelju unesenih igrača i njima pridruženih ocjena iz zadanih kategorija. Broj igrača, broj timova, kategorije i raspon ocjena mogu se zadavati unutar aplikacije.

Datum ocjene pismenog dijela završnog rada od strane mentora:	23.09.2024.
Ocjena pismenog dijela završnog rada od strane mentora:	Vrlo dobar (4)
Datum obrane završnog rada:	14.10.2024.
Ocjena usmenog dijela završnog rada (obrane):	Vrlo dobar (4)
Ukupna ocjena završnog rada:	Vrlo dobar (4)
Datum potvrde mentora o predaji konačne verzije završnog rada čime je pristupnik završio stručni prijediplomski studij:	14.10.2024.

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

IZJAVA O IZVORNOSTI RADA

Osijek, 14.10.2024.

Ime i prezime Pristupnika:	Ilija Vurbić
Studij:	Stručni prijediplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	AR 4829, 27.07.2020.
Turnitin podudaranje [%]:	13

Ovom izjavom izjavljujem da je rad pod nazivom**: Web aplikacija za generanje uravnoteženih sportskih ekipa**

izrađen pod vodstvom mentora Marina Peko, dipl. ing. el.

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 pristupnika:

SADRŽAJ

1. UVOD

U današnjem dinamičnom i tehnološki naprednom svijetu, softverske aplikacije igraju ključnu ulogu u različitim sferama života, uključujući sport. Sportovi kao što je odbojka zahtijevaju učinkovito upravljanje igračima, timovima i utakmicama kako bi se osigurao kvalitetan i organiziran trening te natjecateljski proces. Tradicionalni načini upravljanja ovim aspektima, koji često uključuju papirnatu dokumentaciju i ručno evidentiranje, postaju zastarjeli i neučinkoviti. Stoga je potreba za digitalizacijom i automatizacijom sportskih aktivnosti postala očita.

Ovaj završni rad bavi se razvojem web aplikacije za generiranje nasumičnih timova odbojkaša koristeći moderni okvir ASP.NET Core. Cilj aplikacije je omogućiti trenerima jednostavno upravljanje igračima i njihovim statistikama, te automatizirano generiranje timova za natjecanja. Uz to, aplikacija omogućuje administratorima upravljanje utakmicama i relevantnim informacijama, čime se dodatno olakšava organizacija sportskih događanja.

Web aplikacija implementira sustav autentifikacije i autorizacije koji podržava različite korisničke uloge (trener, igrač, administrator), čime se osigurava kontroliran pristup podacima i funkcionalnostima aplikacije. Treneri mogu unositi i ažurirati podatke o igračima, administratori mogu unositi informacije o utakmicama i timovima, dok igrači imaju mogućnost pregleda svojih statistika i rasporeda utakmica.

Kroz ovaj rad, detaljno će se opisati proces razvoja aplikacije, uključujući definiranje modela podataka, implementaciju repozitorija, servisa i kontrolera, kao i integraciju sustava za autentifikaciju i autorizaciju. Poseban naglasak bit će stavljen na algoritam za generiranje nasumičnih timova, koji osigurava ravnomjernu raspodjelu igrača prema njihovim vještinama, čime se postiže balansirana konkurencija unutar timova.

Ovaj završni rad doprinosi boljem razumijevanju primjene ASP.NET Core i React okvira u razvoju web aplikacija i pruža praktičan primjer implementacije sveobuhvatnog sustava za upravljanje sportskim timovima.

2. C# PROGRAMSKI JEZIK

2.1. Uvod

C# je programski jezik koji je razvijen od strane Microsofta, pod vodstvom Andersa Hejlsberga i Scotta Wiltamutha, a prvi put je predstavljen 2000. godine zajedno s .NET platformom. Osnovni cilj ovog jezika bio je kreirati jednostavan, siguran, moderan i visokoperformansni objektno-orijentirani jezik, posebno dizajniran za rad unutar .NET okruženja. .NET platforma predstavlja sveobuhvatan razvojni okvir koji omogućuje korištenje različitih Microsoftovih tehnologija poput ASP.NET za razvoj web aplikacija, XML za rad s podacima, te objektno-orijentirani pristup dizajnu aplikacija.

C# omogućava definiranje klasa, metoda i svojstava, te koristi ključne principe objektno-orijentiranog programiranja kao što su enkapsulacija, nasljeđivanje i polimorfizam, slične onima u jezicima poput Jave i C++. Iako njegova sintaksa dijeli sličnosti s ovim jezicima, C# je prvenstveno namijenjen razvoju aplikacija unutar Microsoftovog ekosustava i nije zamišljen kao platformno neovisan jezik poput Jave. [1]

Jedna od naprednijih značajki C# jezika je podrška za rad s pokazivačima, kao i integracija XML-a, sučelja i događaja. Ove mogućnosti omogućuju izgradnju robusnih i fleksibilnih aplikacija, bilo da se radi o razvoju složenih desktop ili web aplikacija, uz korištenje suvremenih razvojnih paradigmi.

2.2. Tipovi C# podataka

C#, kao i većina programskih jezika, koristi varijable i konstante za rad s informacijama u memoriji. Varijable su elementi koji mogu pohranjivati podatke čije se vrijednosti mogu mijenjati tijekom izvođenja programa. Nasuprot tome, konstante su namijenjene za pohranu vrijednosti koje se ne mijenjaju nakon inicijalizacije. U C#, konstante se definiraju pomoću ključne riječi *const*, dok se varijable mogu deklarirati bez ove oznake, čime omogućuju dinamičko mijenjanje svojih vrijednosti tijekom izvođenja programa. Prema tome tko ih definira, tipovi se dijele na *Intrinsic* ili *built-in* (ugrađeni tipovi) te *Userdefined* (oni koje definira korisnik) [1]

Nekoliko primjera imena: a, b, i12, varijabla, drugaVarijabla.

2.3. **Blokovi, petlje i grananja**

Blok je najjednostavnija vrsta strukturiranog izraza. Namjena mu je da okupi niz naredbi zatvorenih vitičastim zagradama u jednu naredbu.

Petlje se koriste za uzastopno ponavljanje izraza. Postoji nekoliko petlji: *while, do-while, for, foreach.*

While petlja se ponavlja sve dok je zadani uvjet istinit. Kada kompajler dođe do *while* petlje, procjenjuje logički izraz koji vraća vrijednost *true* ili *false*. Ako je vrijednost *false*, preskače se ostatak petlje i nastavlja se izvršavanje programa. Ako je vrijednost *true*, izvršava se izraz unutar petlje, zatim se vraća na početak petlje i ponavlja postupak.

Do-while petlja je za slučajeve kada je potrebno da se uvjet ponavljanja testira na kraju petlje umjesto na početku. *Do-while* je zapravo *while* petlja s uvjetom ponavljanja na kraju petlje.

For petlja je slična *while* petlji, ali je za neke namjene prikladnija od *while* petlje. Inicijalizacija, uvjet nastavljanja i promjena vrijednosti su objedinjeni u prvoj liniji *for* petlje. Na taj način su svi činitelji *for* petlje na jednom mjestu što olakšava čitanje i razumijevanje. Uvjet nastavljanja mora biti logički izraz, dok inicijalizacija i promjena vrijednosti mogu biti bilo kakvi izrazi.

Foreach petlja koristi se za iteraciju kroz stavke u popisu. Ona djeluje na polja kao što su *ArrayList*.

3. ASP.NET CORE OKRUŽENJE

3.1. Osnovne karakteristike ASP.NET-a

ASP.NET je open-source web okvir koji je razvio Microsoft, s ciljem izrade dinamičkih web stranica, aplikacija i servisa. Prvobitno je predstavljen 2002. godine kao nasljednik klasičnog ASP-a i ubrzo je postao ključni dio šireg **.**NET ekosustava.

ASP.NET nudi brojne prednosti koje ga čine popularnim među web programerima. Podržava različite programerske modele, uključujući Web Forms, MVC (*Model-View-Controller*) i Blazor, koji omogućava izradu interaktivnih web aplikacija koristeći C# umjesto JavaScript-a. [2] Osim toga, ASP.NET se ističe visokom performansom zahvaljujući sposobnosti kompajliranja koda u nativni strojni kod, a napredni mehanizmi za keširanje i optimizaciju osiguravaju visoku skalabilnost. [3]

Integrirana sigurnosna rješenja kao što su autentikacija, autorizacija, zaštita od CSRF (*Cross-Site Request Forgery*) i enkripcija, čine ASP.NET odličnim izborom za razvoj sigurnih aplikacija [4]. *Visual Studio i Visual Studio Code* su popularni IDE-ovi za ASP.NET razvoj, nudeći bogat set alata za *debugging*, testiranje i implementaciju. Također, postoji veliki broj dodatnih biblioteka koje omogućavaju jednostavno proširenje funkcionalnosti aplikacija [5].

S izlaskom .NET Core, ASP.NET je postao cross-platform rješenje koje omogućava razvoj i pokretanje aplikacija na različitim operativnim sustavima, uključujući Windows, Linux i macOS [6].

3.2. Primjena

Primjena ASP.NET-a je vrlo široka i obuhvaća različite vrste web aplikacija. Često se koristi za izradu robusnih *e-commerce* platformi zbog svoje visoke skalabilnosti i naprednih sigurnosnih značajki. Mnogi sustavi za upravljanje sadržajem (CMS), poput popularnog Umbraca, razvijeni su na temelju ASP.NET-a. Osim toga, korporativne aplikacije koje zahtijevaju visoku razinu sigurnosti i performansi često se razvijaju koristeći ovaj okvir. ASP.NET Web API dodatno omogućava razvoj RESTful servisa, koji su idealni za mobilne aplikacije i druge klijentske aplikacije [4].

Primjena ASP.NET *frameworka* može se demonstrirati kroz nekoliko primjera koji ilustriraju osnovne aspekte razvoja web aplikacija koristeći ovaj moćan alat. U nastavku ćemo opisati primjer za osnovnu web aplikaciju, RESTful API, te detaljno objasniti njihov kod i funkcionalnost.

ASP.NET Core Web API omogućava razvoj RESTful servisa koji se mogu koristiti za mobilne aplikacije i druge klijentske aplikacije. Za kreiranje API aplikacije koristimo naredbu dotnet new webapi -o MyApi. Nakon kreiranja, prelazimo u direktorij aplikacije (cd MyApi).

U ovoj API aplikaciji, prvo dodajemo model. U mapi *Models*, kreiramo datoteku *TodoItem*.cs koja definira klasu *TodoItem* sa svojstvima *Id, Name i IsComplete.*

Linija ***Kod***

```
1:      Namespace MyApi.Models
2:      {
3:         public class ToDoItem
4:             {
5:                public long Id {get;set;}
6:                public string Name { get; set; }
7:                public bool IsComplete { get; set; }
8:             }
9:      }
```

Sl. 3.1. Kôd primjer

Zatim dodajemo kontroler. U mapi *Controllers*, kreiramo datoteku *TodoController*.cs. Ovaj kontroler sadrži metode za dobivanje svih stavki (*Get*), dobivanje stavke po ID-u (*Get*(id)), kreiranje nove stavke (*Post*), ažuriranje postojeće stavke (*Put*(id)), i brisanje stavke (*Delete*(id)).

Linija ***Kod***

```
1:      Namespace MyApi.Controllers
2:      {
3:         [route(„api/[controller]“)]
4:         [ApiController]
5:          public class TodoController : ControllerBase
6:                {
7:                private static readonly List<TodoItem> items = new
8:      List<TodoItem>();
9:                [HttpGet]
10:               public ActionResult<IEnumerable<TodoItem>> Get()
11:               {
12:                   return items;
13:               }
14:               [HttpGet(„{id}“)]
15:                public ActionResult<TodoItem> Get(long id)
16:                {
17:                   var item = items.FirstOrDefault(t => t.Id == id);
18:                   if (item == null)
19:                   {
20:                     return NotFound();
21:                   }
22:                   return item;
23:             }
24:             [HttpPost]
25:             public ActionResult<TodoItem> Post(TodoItem item)
```

Linija ***Kod***

```
26:             {
27:             var item = items.FirstOrDefault(t => t.Id == id);
28:             if (item == null)
29:             {
30:                return NotFound();
31:             }
32:             return item;
33:             }
34:             [HttpPost]
35:             public ActionResult<TodoItem> Post(TodoItem item)
36:             {
37:               item.Id = items.Count > 0 ? items.Max(t => t.Id) + 1 : 1;
38:               items.Add(item);
39:               return CreatedAtAction(nameof(Get), new { id = item.Id },
    item);
40:             }
41:             [HttpPut("{id}")]
42:             public IActionResult Put(long id, TodoItem item)
43:             {
44:                var existingItem = items.FirstOrDefault(t => t.Id == id);
45:                if (existingItem == null)
46:                {
47:                   return NotFound();
48:                }
49:                existingItem.Name = item.Name;
50:                existingItem.IsComplete = item.IsComplete;
51:                return NoContent();
52:                }
53:             [HttpDelete("{id}")]
54:             public IActionResult Delete(long id)
55:             {
56:                var item = items.FirstOrDefault(t => t.Id == id);
57:                if (item == null)
58:                {
59:                   return NotFound();
60:                }
61:                items.Remove(item);
62:                return NoContent();
63:             }
64:            }
65:         }
```

Sl. 3.2. Kôd primjer

Ovaj API omogućava osnovne CRUD (*Create*, *Read*, *Update*, *Delete*) operacije za upravljanje zadacima. Kada pokrenemo aplikaciju pomoću *dotnet run*, API će biti dostupan na *http://localhost:5000/api/todo*.

4. REACT OKRUŽENJE

4.1. Ključne značajke React okruženja

React je JavaScript knjižnica za izgradnju korisničkih sučelja, razvijena od strane Facebooka. Omogućuje razvoj dinamičkih web aplikacija s visokom performansom, a posebno je popularna zbog svoje jednostavnosti i učinkovitosti u izradi komponenti. [7].

Jedna od ključnih karakteristika React-a su komponente. React je temeljen na komponentama, što znači da su korisničko sučelje i njegove funkcionalnosti podijeljeni na manje, samostalne dijelove koji se mogu ponovno koristiti. Svaka komponenta može održavati vlastito stanje (state), koje se može mijenjati nezavisno od drugih komponenti, što olakšava razvoj složenih aplikacija [8].

Još jedna značajna inovacija koju React donosi je virtualni DOM (*Document Object Model*). Kada se stanje aplikacije promijeni, React ažurira virtualni DOM umjesto stvarnog DOM-a, a zatim uspoređuje promjene i minimalno ažurira stvarni DOM. Ovaj pristup značajno poboljšava performanse aplikacije [9].

Također, React implementira jednosmjernu vezu podataka (*One-way Data Binding*), što znači da se podaci prenose iz stanja (*state*) komponenti prema prikazu (*view*), ali ne i obrnuto. Ovaj princip pojednostavljuje razumijevanje toka podataka kroz aplikaciju i olakšava debugiranje [11].

4.2. Primjena React-a

U ovom primjeru, izgradit ćemo jednostavnu React aplikaciju koja će komunicirati s RESTful API-jem izrađenim u ASP.NET Core-u iz prethodnog primjera. Aplikacija će omogućiti pregled, dodavanje, uređivanje i brisanje zadataka (*Todo items*).

Prvi korak je instalacija Node.js-a i npm-a (Node Package Manager) sa službene stranice Node.js-a. Koristimo *create-react-app* alat za kreiranje novog projekta. U direktoriju projekta, instaliramo axios za jednostavno izvršavanje HTTP zahtjeva. Unutar mape src, stvaramo mapu components gdje smještamo naše React komponente.

Linija Kod

```
1:         const TodoList = () => {
2:           const [todos, setTodos] = useState([]);
3:           const [editingTodo, setEditingTodo] = useState(null);
4:
5:           useEffect(() => {
6:             fetchTodos();
7:           }, []);
8:
9:           const fetchTodos = async () => {
10:     const response = await axios.get('http://localhost:5000/api/todo');
11:           setTodos(response.data);
12:         };
13:
14:         const addTodo = async (todo) => {
15:    const response = await axios.post('http://localhost:5000/api/todo',
todo);
16:           setTodos([...todos, response.data]);
17:         };
18:
19:         const updateTodo = async (todo) => {
20:     await axios.put(`http://localhost:5000/api/todo/${todo.id}`, todo);
21:           setTodos(todos.map(t => (t.id === todo.id ? todo : t)));
22:           setEditingTodo(null);
23:         };
24:
25:         const deleteTodo = async (id) => {
26:           await axios.delete(`http://localhost:5000/api/todo/${id}`);
27:           setTodos(todos.filter(t => t.id !== id));
28:         };
```

Linija Kod

```
29:
30:         return (
31:           <div>
32:             <h1>Todo List</h1>
33:                   <TodoForm addTodo={addTodo} updateTodo={updateTodo}
editingTodo={editingTodo} />
34:             <ul>
35:               {todos.map(todo => (
36:                 <li key={todo.id}>
37:             {todo.name} - {todo.isComplete ? 'Complete' : 'Incomplete'}
38:              <button onClick={() => setEditingTodo(todo)}>Edit</button>
39:            <button onClick={() => deleteTodo(todo.id)}>Delete</button>
40:                 </li>
41:               ))}
42:             </ul>
43:           </div>
44:         );
45:       };
46:
47:       export default TodoList;
```

Sl. 4.1. Kôd primjer

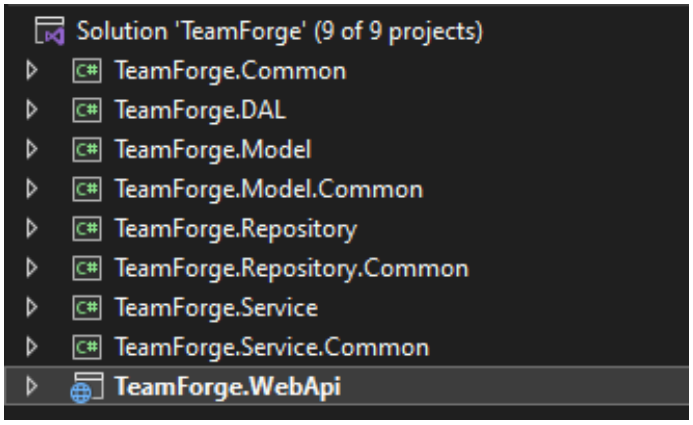
TodoList Komponenta je odgovorna za prikaz popisa zadatka, dohvaćanje podataka s RESTful API-ja pomoću Axiosa te upravljanje operacijama dodavanja, ažuriranja i brisanja zadataka.

U terminalu, unutar direktorija poziva se naredba *npm run start* te se pokreće React aplikacija. Ovaj primjer demonstrira kako se React može uspješno integrirati s backendom izgrađenim u ASP.NET Core-u, omogućujući razvoj dinamičkih web aplikacija s intuitivnim korisničkim sučeljem i efikasnim upravljanjem stanjem aplikacije.

5. IZRADA BACKEND DIJELA WEB APLIKACIJE

5.1. Struktura backend-a

Ovaj rad analizira tipičnu strukturu backend projekta u .NET okruženju, koristeći slojevitu arhitekturu kao referentni okvir. Predstavljena struktura omogućava jasno razdvajanje odgovornosti, skalabilnost i održavanje projekta. U radu se detaljno objašnjavaju funkcionalnosti i uloge svakog sloja unutar arhitekture. Slojevita arhitektura organizira kod u nekoliko slojeva, pri čemu svaki sloj ima specifične odgovornosti. Glavni slojevi uključuju sloj za pristup podacima (*DAL*), poslovnu logiku (*Service*), repozitorij (*Repository*), modele (*Model*) i web API (*WebApi*).



Sl. 5.1. Struktura projekta

TeamForge.Common projekt sadrži zajedničke komponente i funkcionalnosti koje se koriste u više slojeva ili projekata unutar rješenja. Komponente uključuju pomoćne klase, ekstenzije i zajedničke konfiguracije. Ovaj sloj poboljšava ponovno korištenje koda i smanjuje dupliciranje. *TeamForge.DAL* je sloj za pristup podacima upravlja svim operacijama vezanim za bazu podataka. Korištenjem ORM alata poput *Entity Frameworka*, DAL omogućava komunikaciju s bazom podataka putem SQL upita i procedura. Glavna odgovornost ovog sloja je pohrana i dohvat podataka. Projekt koji sadrži modele ili entitete koji predstavljaju podatkovne strukture korištene u aplikaciji je *TeamForge.Model*. Modeli su klase koje odgovaraju tablicama u bazi podataka ili poslovnim objektima, omogućavajući jasan i strukturiran prikaz podataka. *Common* projekt uključuje zajedničke modele ili entitete koji se koriste u više slojeva ili projekata. Sadrži osnovne klase ili sučelja za modele, olakšavajući ponovno korištenje i standardizaciju podataka kroz različite dijelove aplikacije. Implementirajući obrazac repozitorija, sloj *TeamForge.Repository* pruža apstrakciju pristupa podacima. Omogućava interakciju s DAL-om na način koji podržava

jednostavnu zamjenu ili promjenu implementacije pristupa podacima bez utjecaja na ostatak aplikacije. *Common* projekt sadrži zajedničke repozitorijske klase ili sučelja koji se koriste u više slojeva ili projekata. Omogućava osnovne klase ili generičke implementacije, podržavajući standardizaciju i modularnost. Sloj za poslovnu logiku *TeamForge.Service* implementira poslovna pravila i procese. Komunicira s repozitorijima kako bi dobio potrebne podatke, obradio ih i vratio rezultate klijentu ili kontroleru. Ovaj sloj osigurava da je poslovna logika centralizirana i lako održiva. *Common* projekt sadrži zajedničke servisne klase ili sučelja koji se koriste u više slojeva ili projekata. Sadrži osnovne servise ili generičke implementacije, olakšavajući ponovno korištenje poslovne logike. Web API projekt služi kao ulazna točka za vanjske zahtjeve. Sadrži kontrolere koji primaju HTTP zahtjeve, pozivaju odgovarajuće servise i vraćaju odgovore klijentima. Korištenjem ASP.NET Web API okvira, ovaj sloj omogućava izradu RESTful usluga.

5.2. Objašnjenje koda

Linija Kod

```
1:        public class CommonProperties
2:        {
3:           public          string          connectionString          =
"Server=teamforgeserver.mysql.database.azure.com;  port=3306;  UserID  =
vurbic;Password=Ilija31531;Database=teamforgedb;";
4:        }
```

Sl. 5.2. CommonProperties klasa projekta

connectionString je niz koji sadrži informacije potrebne za povezivanje s bazom podataka. U ovom slučaju, niz je konfiguriran za povezivanje s MySQL bazom podataka. Evo što svaki dio niza predstavlja:

- Server = teamforgeserver.mysql.database.azure.com: IP adresa ili naziv servera na kojem se nalazi baza podataka. U ovom slučaju, teamforgeserver.mysql.database.azure.com je adresa host-a na Azure platformi.
- *Database=TeamForgeDB*: Naziv baze podataka na koju se aplikacija želi povezati, ovdje je to *TeamForgeDB*.
- *Userid*=vurbic: Korisničko ime za pristup bazi podataka, ovdje je to vurbic.
- Pwd=Ilija31531: Lozinka za korisnički račun vurbic.
- *Port*=3306: *Port* na kojem MySQL server sluša zahtjeve. Zadani *port* za MySQL je 3306.

Linija Kod

```
1:        public class AdminRepository : IAdminRepository
2:        {
3:            CommonProperties cProperties = new CommonProperties();
4:        }
```

Sl. 5.3. klasa AdminRepository

Ova klasa implementira sučelje, osiguravajući da su sve potrebne metode definirane. Klasa koristi *CommonProperties* za pristup informacijama o povezivanju s bazom podataka.

Linija Kod

```
1:        public Admin GetAdminById(Guid adminId)
2:        {
3:         MySqlConnection conn = new
MySqlConnection(cProperties.connectionString);
4:            using (conn)
5:            {
6:               string query = "SELECT * FROM Admin WHERE Id = @AdminId";
7:       return  conn.QueryFirstOrDefault<Admin>(query,  new  {  AdminId  =
adminId});
8:            }
9:        }
```

Sl. 5.4. metoda GetAdminById

Ova metoda koristi *MySqlConnection* za povezivanje s bazom podataka i *Dapper* ORM za izvršavanje SQL upita. Metoda dohvaća admin podatke prema jedinstvenom identifikatoru (GUID).

Linija Kod

```
1:         public IEnumerable<Admin> GetAllAdmins()
2:         {
3:             MySqlConnection conn = new
MySqlConnection(cProperties.connectionString);
4:             using (conn)
5:             {
6:                 string query = "SELECT * FROM Admin";
7:                 return conn.Query<Admin>(query).ToList();
8:             }
9:         }
```

Sl. 5.5. metoda GetAllAdmins

Ova metoda vraća popis svih admin podataka iz baze podataka.

Linija Kod

```
1:         public void AddAdmin(Admin admin)
2:         {
3:            MySqlConnection conn = new
MySqlConnection(cProperties.connectionString);
4:             using (conn)
5:             {
6:    string query = @"INSERT INTO Admin (Id, Username, Email, PasswordHash)
7:                VALUES (@Id, @Username, @Email, @PasswordHash)";
8:                  conn.Execute(query, admin);
9:             }
10:        }
```

Sl. 5.6. metoda AddAdmin

Ova metoda omogućava dodavanje novog admina u bazu podataka.

Linija Kod

```
1:        public void UpdateAdmin(Admin admin)
2:        {
3:            MySqlConnection conn = new
MySqlConnection(cProperties.connectionString);
4:            using (conn)
5:            {
6:                string query = @"UPDATE Admin
7:   SET Username = @Username, Email = @Email, PasswordHash = @PasswordHash
8:                              WHERE Id = @Id";
9:                conn.Execute(query, admin);
10:           }
11:       }
```

Sl. 5.7. metoda UpdateAdmin

Ova metoda omogućava ažuriranje postojećih admin podataka.

Linija Kod

```
1:        public void DeleteAdmin(Guid adminId)
2:        {
3:           MySqlConnection conn = new
MySqlConnection(cProperties.connectionString);
4:            using (conn)
5:            {
6:                string query = "DELETE FROM Admin WHERE Id = @AdminId";
7:                conn.Execute(query, new { AdminId = adminId });
8:            }
9:        }
```

Sl. 5.8. metoda DeleteAdmin

Ova metoda omogućava brisanje admina iz baze podataka.

Linija Kod

```
1:       builder.Services.AddScoped<IAdminRepository, AdminRepository>();
2:        builder.Services.AddScoped<IAdminService, AdminService>();
```

Sl. 5.20. Dependency Injection

U softverskom inženjeringu, upravljanje zavisnostima je ključno za izgradnju skalabilnih i održivih aplikacija. *Dependency Injection* (DI) je obrazac dizajna koji omogućava injekciju zavisnosti u objekte umjesto da ih objekti sami kreiraju. Ovaj pristup povećava fleksibilnost i

olakšava testiranje koda. Dependency Injection je tehnika kojom se zavisnosti (objekti koje klasa treba da bi obavila svoj posao) injektiraju u klasu umjesto da ih klasa sama kreira. Ovo se postiže kroz konfiguraciju servisa i njihovom registracijom u IoC (Inversion of Control) kontejneru. Ovaj kod registrira *IAdminRepository* i *IAdminService* kao *scoped* servise. *Scoped* servisi kreiraju novi objekt po zahtjevu unutar pojedinačnog HTTP zahtjeva. Registrira se *IAdminRepository* sučelje sa *AdminRepository* implementacijom. Novi primjerak *AdminRepository* klase kreira se po svakom HTTP zahtjevu i koristi se za sve zavisnosti unutar tog zahtjeva. Također se registrira *IAdminService* sučelje sa *AdminService* implementacijom. Novi primjerak *AdminService* klase kreira se po svakom HTTP zahtjevu i koristi se za sve zavisnosti unutar tog zahtjeva.

6. ALGORITAM ZA GENERIRANJE BALANSIRANIH TIMOVA

6.1. Sortiranje igrača

Algoritam se sastoji od nekoliko ključnih koraka: sortiranje igrača, inicijalizacija praznih timova i ravnomjerno raspoređivanje igrača među timovima.

Linija Kod

```
1:      players.Sort((p1, p2) =>
p2.OverallRating.CompareTo(p1.OverallRating));
```

Sl. 6.1. Sortiranje igrača

Prvi korak algoritma je sortiranje igrača prema njihovim rejtingima u opadajućem redoslijedu. Ovo omogućuje da najbolji igrači budu ravnomjerno raspoređeni među timovima.

6.2. Inicijalizacija praznih timova

Nakon sortiranja igrača, inicijaliziraju se prazni timovi. Broj timova određuje se parametrom `numTeams`:

Linija Kod

```
1:          var teams = new List<List<Player>>();
2:          for (int i = 0; i < numTeams; i++)
3:          {
4:              teams.Add(new List<Player>());
5:          }
```

Sl. 6.2. Kreiranje liste

Ovdje se kreira lista timova gdje je svaki tim inicijalno prazan.

6.3. Ravnomjerno raspoređivanje igrača

Nakon inicijalizacije praznih timova, igrači se ravnomjerno raspoređuju među timovima:

Linija Kod

```
1:          int currentTeamIndex = 0;
2:          foreach (var player in players)
3:          {
4:              teams[currentTeamIndex].Add(player);
5:              currentTeamIndex = (currentTeamIndex + 1) % numTeams;
6:          }
```

Sl. 6.3. Raspoređivanje igrača

Ovdje se koristi jednostavan ciklički pristup za raspoređivanje igrača. Svaki igrač se dodaje u trenutni tim, a indeks trenutnog tima se povećava i resetira na 0 kada dosegne broj timova. Na taj način, igrači su ravnomjerno raspoređeni među svim timovima.

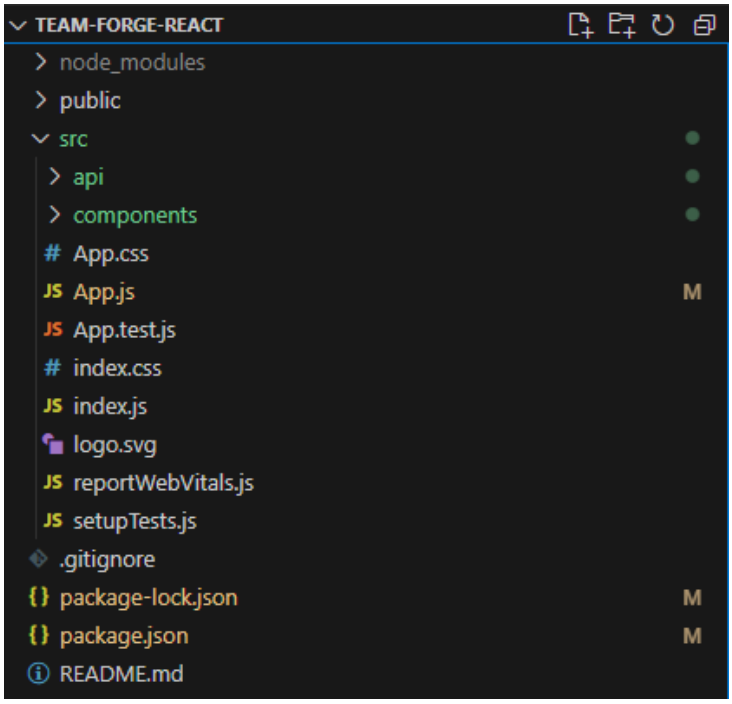
6.4. Rezultati

Rezultat algoritma je lista timova, gdje je svaki tim lista igrača. Timovi su balansirani na način da svaki tim ima približno jednaku ukupnu kvalitetu, s obzirom na rejtinge igrača. Predstavljeni algoritam je jednostavan, ali učinkovit način za generiranje balansiranih sportskih timova. Sortiranjem igrača prema rejtingu i ravnomjernim raspoređivanjem među timovima, osigurava se da svaki tim ima sličnu razinu kvalitete.

7. IZRADA FRONTEND DIJELA WEB APLIKACIJE

7.1. Struktura frontend-a

Struktura aplikacije prikazana je na slici. Svaka komponenta i mapa ima specifičnu ulogu u aplikaciji. Ovdje ćemo detaljno opisati ulogu i sadržaj svake mape i datoteke.



Sl. 7.1. Struktura React projekta

Korijenska mapa sadrži osnovne konfiguracijske datoteke i mape potrebne za rad aplikacije. Mapa *src* sadrži sav izvorni kod aplikacije. Ključne mape unutar *src* su *api* i *components*. Mapa *api* sadrži datoteke koje upravljaju komunikacijom s *backend* API-jem. Ovdje se koriste Axios ili slične biblioteke za slanje HTTP zahtjeva. Mapa *components* sadrži sve React komponente koje čine korisničko sučelje aplikacije. Svaka komponenta ima specifičnu ulogu u prikazu i upravljanju podacima unutar aplikacije.

7.2. Objašnjenje koda

API servis je implementiran kao skup asinhronih funkcija koje koriste Axios za slanje HTTP zahtjeva na *backend* server. Svaka funkcija je dizajnirana da obavi specifičan zadatak vezan za upravljanje podacima o igračima.

Prva linija koda definira osnovnu URL adresu za API:

Linija Kod

```
1:        const API_BASE_URL = 'https://localhost:7152/api';
```

Sl. 7.2. Veza sa backendom

Funkcija *fetchAllPlayers* šalje GET zahtjev na *endpoint /players* za dohvaćanje svih igrača:

Linija Kod

```
1:        export const fetchAllPlayers = async () => {
2:            try {
3:            const response = await axios.get(`${API_BASE_URL}/players`);
4:                return response.data;
5:            } catch (error) {
6:                console.error('Error fetching players:', error);
7:                throw error;
8:            }
9:        };
```

Sl. 7.2. Funkcija za dohvaćanje igrača

Ova funkcija vraća podatke o igračima ili baca grešku ako zahtjev nije uspješan. Korištenje t*ry-catch* bloka omogućava hvatanje i obradu grešaka koje se mogu dogoditi tijekom HTTP zahtjeva.

Funkcija *createPlayer* šalje POST zahtjev za kreiranje novog igrača:

Linija Kod

```
1:        export const createPlayer = async (player) => {
2:            try {
3:            const response = await axios.post(`${API_BASE_URL}/players`,
player);
4:                return response.data;
5:            } catch (error) {
6:                console.error('Error creating player:', error);
7:                throw error;
8:            }
9:        };
```

Sl. 7.3. Funkcija za kreiranje igrača

Ova funkcija prima objekt *player* koji sadrži podatke o novom igraču. Podaci se šalju na server, a odgovor servera se vraća kao rezultat funkcije.

Funkcija *updatePlayer* šalje PUT zahtjev za ažuriranje podataka o postojećem igraču:

Linija Kod

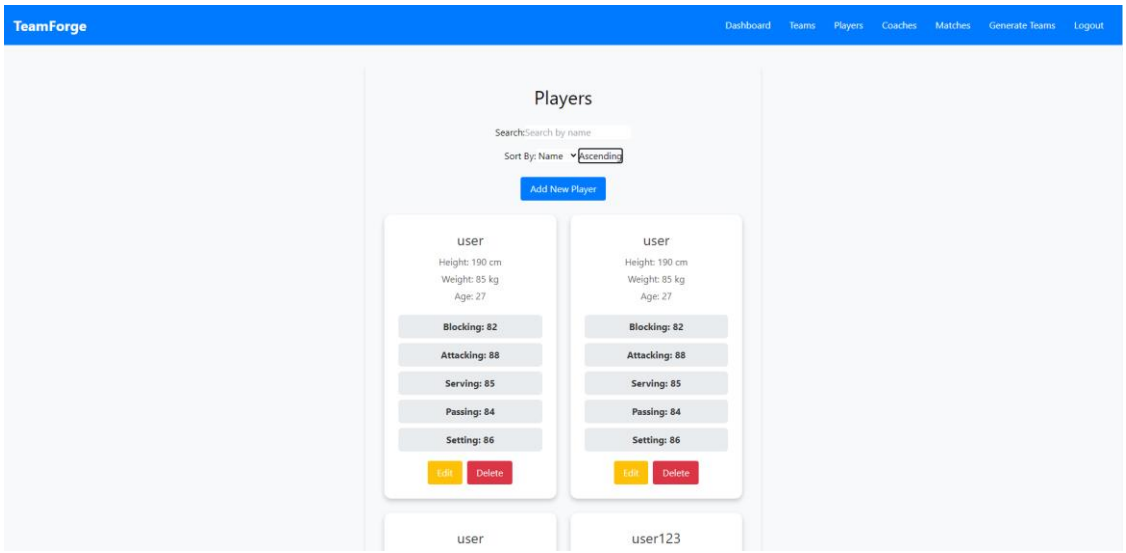
```
1:        export const updatePlayer = async (player) => {
2:            try {
3:                                          const   response   =   await
axios.put(`${API_BASE_URL}/players/${player.id}`, player);
4:                return response.data;
5:            } catch (error) {
6:                console.error('Error updating player:', error);
7:                throw error;
8:            }
9:        };
```

Sl. 7.4. Funkcija za ažuriranje igrača

Ova funkcija prima objekt *player* koji sadrži ažurirane podatke o igraču, uključujući njegov ID. Podaci se šalju na server, a odgovor servera se vraća kao rezultat funkcije.

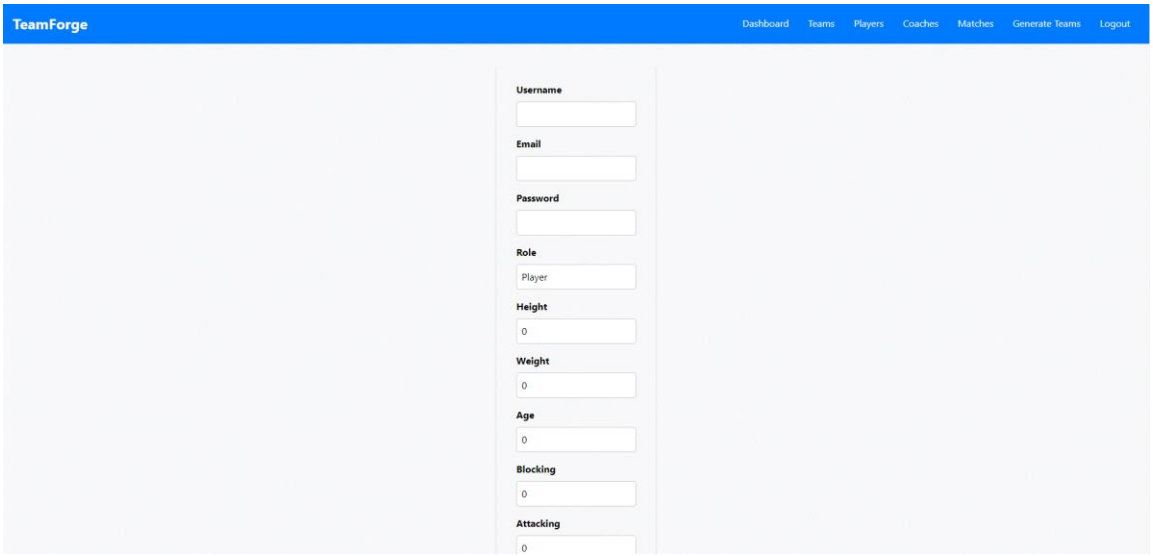
8. PRIKAZ APLIKACIJE

U web aplikaciji moguće je unositi igrače, trenere, timove, utakmice i rezultate, te u slučaju pogreške pri unosu može se naknadno urediti podatke ili izbrisati unos. Na slici 8.1 prikazani su svi igrači koji se nalaze u bazi podataka zajedno sa svojim ocjenama koji se mogu filtrirati te pretražiti.



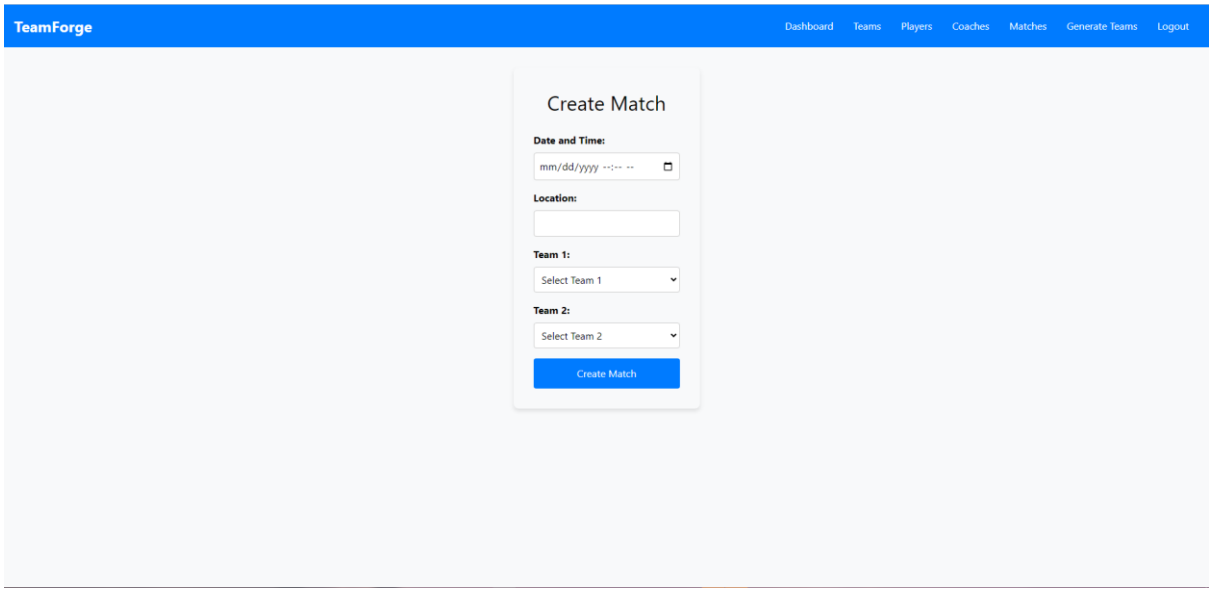
Sl. 8.1. prikaz svih igrača

Na slici 8.2. prikazan je zaslon koji se otvara nakon pritiska „*Add new player*“. Ovaj zaslon služi za unos novih igrača te njihovih ocjena te stvari kao što su korisničko ime i lozinka.



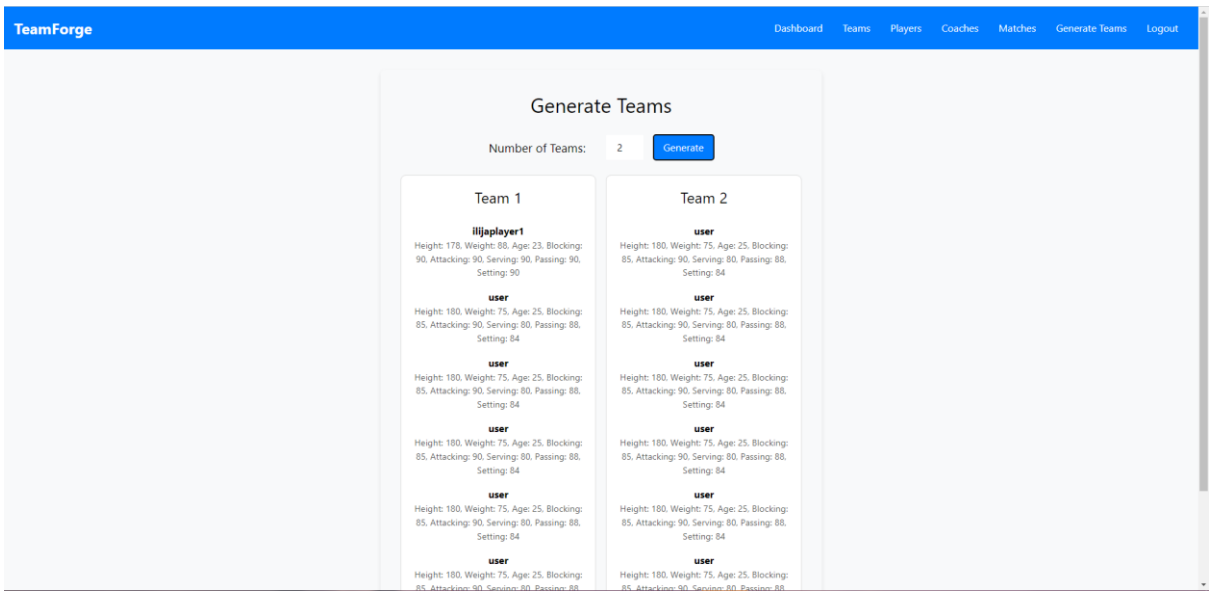
Sl. 8.2. unos igrača

Slika 8.3. prikazuje unos utakmica te informacije o utakmici kao što su vrijeme, mjesto i timovi koji sudjeluju na utakmici.



Sl. 8.3. unos utakmice

Na slici 8.4. prikazano je ravnopravno raspoređivanje igrača u timove. Može se dodavati više od 2 tima, a svaki tim generira se sa po 6 igrača. Generirani timovi zatim imaju prikazane igrače te njihove karakteristike.



Sl. 8.4. generiranje timova

9. USPOREDBA S POSTOJEĆIM APLIKACIJAMA

9.1. SportsEngine

SportsEngine je još jedna popularna aplikacija koja se koristi za upravljanje sportskim organizacijama i timovima. Njene glavne značajke uključuju upravljanje registracijama i plaćanjima, kreiranje rasporeda utakmica i treninga, te organizaciju i praćenje sportskih događaja.

Za razliku od *SportsEnginea,* koji je prvenstveno usmjeren na administraciju sportskih organizacija, aplikacija razvijena u ovom radu fokusira se na generiranje sportskih timova na temelju vještina igrača. Implementirani algoritam u ovoj aplikaciji omogućuje ravnomjernu raspodjelu igrača u timove, osiguravajući balansiranost, što *SportsEngine* ne nudi. Iako *SportsEngine* pruža širok raspon administrativnih funkcija, nedostaje mu mogućnost optimizacije timova prema vještinama igrača, što je ključna prednost aplikacije razvijene u ovom radu.

9.2. MyTeam

MyTeam je jednostavnija aplikacija za vođenje timova koja pruža osnovne funkcionalnosti za upravljanje sportskim ekipama, uključujući komunikaciju putem obavijesti i poruka.

Aplikacija razvijena u ovom radu nudi znatno naprednije funkcionalnosti, uključujući algoritam za automatsko generiranje uravnoteženih timova, dok *MyTeam* nudi samo osnovne opcije za upravljanje rasporedom i komunikaciju.

9.3. Spliddit (Fair Division Tool)

Spliddit nudi algoritamske alate za fer podjelu resursa i raspoređivanje, uključujući timove. Cilj je pravedno podijeliti resurse među sudionicima, uključujući raspodjelu igrača u sportskim timovima. Omogućuje korisnicima unos različitih parametara za generiranje timova s naglaskom na ravnopravnu raspodjelu resursa. Iako nije specifično za sport, može se koristiti za generiranje timova prema definiranim preferencijama. *Spliddit* nije sportska aplikacija, ali koristi napredne algoritme za ravnomjernu podjelu, što je slično cilju ravnomjernog rasporeda igrača prema vještinama.

10. ZAKLJUČAK

U ovom završnom radu, uspješno je razvijena web aplikacija za generiranje uravnoteženih sportskih ekipa, koja koristi moderne tehnologije poput ASP.NET Core za *backend* i React za *frontend*. Aplikacija omogućava jednostavno i efikasno upravljanje sportskim timovima, igračima i njihovim statistikama, te automatsko generiranje timova na temelju algoritma koji osigurava ravnomjerno raspoređivanje igrača prema njihovim vještinama.

Ovaj projekt pokazuje praktičnu primjenu ASP.NET Core okvira za izgradnju sigurnih i skalabilnih web aplikacija, s implementacijom sustava autentifikacije i autorizacije koji omogućuju različite korisničke uloge poput trenera, igrača i administratora. Algoritam za balansiranje timova predstavlja ključni doprinos, omogućujući pošteno raspoređivanje igrača i poboljšanje kvalitete sportskih događanja.

React okruženje na *frontend*-u omogućuje izradu intuitivnog korisničkog sučelja koje je jednostavno za korištenje i održavanje, dok integracija sa *backendom* omogućava dinamičko ažuriranje podataka i prikaz rezultata. Aplikacija pruža primjer kako tehnologije ASP.NET Core i React mogu zajedno funkcionirati u stvaranju funkcionalnih i efikasnih web rješenja.

Rad također naglašava važnost digitalizacije sportskih aktivnosti u cilju modernizacije i poboljšanja organizacije sportskih događaja, te doprinosi razumijevanju primjene suvremenih tehnologija u ovom području.

LITERATURA

[1] L. Kicoš, "Uvod u C#," dostupno na: https://dokumen.tips/documents/pdf1-uvod-u-c-etfosunioshrwwwetfosunioshrlukicoopauditornevjezbe5pdfljudi.html

[2] Esposito, D. (2020). Programming ASP.NET Core. Microsoft Press.

[3] Freeman, A. (2020). Pro ASP.NET Core 3. Apress.

[4] Lock, A. (2019). ASP.NET Core in Action. Manning Publications.

[5] Microsoft Docs. (2023). ASP.NET Documentation. Dostupno na: Microsoft Docs

[6] Skeet, J. (2019). C# in Depth. Manning Publications.

[7] Jackson, A. (2018). Learning React: A Hands-On Guide to Building Web Applications Using React and Redux. Addison-Wesley Professional.

[8] Sams, B. (2020). React - Up & Running: Building Web Applications. O'Reilly Media.

[9] Yang, E. (2019). Mastering React: A Beginner's Guide to Building Modern Web Applications. Packt Publishing.

[10] Hall, K. (2021). Pro React 16. Apress.

[11] Banks, A. (2019). Learning React, 2nd Edition: Functional Web Development with React and Redux. O'Reilly Media.

SAŽETAK

Završni rad bavi se razvojem web aplikacije za generiranje uravnoteženih sportskih ekipa, koristeći ASP.NET Core za backend i React za frontend. Aplikacija je dizajnirana kako bi trenerima omogućila jednostavno upravljanje igračima i njihovim statistikama te automatsko generiranje timova za natjecanja. Algoritam unutar aplikacije osigurava ravnomjernu raspodjelu igrača na temelju njihovih vještina, čime se postiže balansirana konkurencija među timovima. Implementirani sustavi autentifikacije i autorizacije omogućuju različite korisničke uloge (trener, igrač, administrator). Rad naglašava važnost digitalizacije u sportu i pokazuje kako tehnologije ASP.NET Core i React mogu zajedno funkcionirati za izradu funkcionalnih web aplikacija.

Ključne riječi: web aplikacija, ASP.NET Core, React, uravnoteženi sportski timovi, digitalizacija sporta, autentifikacija, autorizacija, algoritam za balansiranje, sport, upravljanje timovima.

ABSTRACT

The thesis focuses on developing a web application for generating balanced sports teams, utilizing ASP.NET Core for the backend and React for the frontend. The application is designed to allow coaches to easily manage players and their statistics, as well as automatically generate teams for competitions. The algorithm ensures an even distribution of players based on their skills, resulting in balanced competition among teams. The implemented authentication and authorization systems provide different user roles (coach, player, administrator). The work emphasizes the importance of digitalization in sports and demonstrates how ASP.NET Core and React technologies can work together to create functional web applications.

Key words: web application, ASP.NET Core, React, balanced sports teams, sports digitalization, authentication, authorization, balancing algorithm, sports management.